Facultad de Ciencias

INTRODUCCIÓN A LA EVALUACIÓN DE LA ARQUITECTURA SVE EN GEM5

(Introduction to SVE Architecture evaluation in gem5)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Susana Rebolledo Ruiz

Director: José Luis Bosque Orero

Julio – 2022

RESUMEN

La arquitectura SVE, del inglés *Scalable Vector Extension*, es una extensión de la ISA ARM para el procesamiento vectorial que permite escalar el tamaño de los registros vectoriales con flexibilidad. El simulador gem5 posibilita el modelado arquitectónico de computadores mediante la simulación de diferentes configuraciones en diversas ISAs entre las que se encuentra ARM. En este Trabajo Fin de Grado se ha realizado una introducción a la evaluación de la arquitectura SVE en gem5. Para ello, se ha realizado una descripción minuciosa de la metodología necesaria para la realización de simulaciones *Full-System* en el entorno gem5, con las herramientas desarrolladas por el Grupo de Arquitectura y Tecnología de computadores (ATC) de la Universidad de Cantabria. Estas simulaciones permiten la evaluación del rendimiento de diferentes benchmarks tras el escalado de la longitud de vector de SVE y el número de cores. Para dicha evaluación, se han desarrollado dos benchmarks; *matrix*, que realiza la multiplicación de dos matrices, y *gauss*, que propone la aplicación de un filtrado gaussiano a una matriz de píxeles. Los resultados preliminares obtenidos en el proceso para el código vectorizado reflejan un mejor rendimiento al escalar el tamaño de vector antes que el número de cores.

Palabras claves: ARM SVE, gem5, Simulación, longitud de vector

ABSTRACT

The Scalable Vector Extension (SVE) is an ARM ISA architecture extension for vectorization that supports flexible vector length scaling. The gem5 simulator allows computer architectural modelling by simulating different configurations on various ISAs including ARM. In this Final Degree Project, the work to evaluate the SVE architecture in gem5 has been introduced. For this purpose, a detailed description of the necessary methodology to carry out *Full-System* simulations in the gem5 environment, using the tools developed by the Computer Architecture and Technology Group (ATC) of the University of Cantabria, has been provided. These simulations allow the evaluation of the performance of different benchmarks after scaling both SVE vector length and number of cores. Two benchmarks have been developed for such evaluation; *matrix*, which performs the multiplication of two matrices, and *gauss*, which applies a Gaussian filter to a pixel matrix. The preliminary results obtained through the process for the developed vectorized code provide a better performance when scaling the vector length rather than the number of cores.

Keywords: ARM SVE, gem5, Simulation, vector length

AGRADECIMIENTOS

A Iván, por su valioso apoyo.

A mi director José Luis, por toda la orientación, ayuda y paciencia.

A mi hermana Carol, por sus consejos.

Gracias.

Índice del TFG

Capítulo 1. Introducción

En este capítulo se describen los conceptos básicos que han motivado la realización de este trabajo. A continuación, se detallan los objetivos y el plan de trabajo seguido para alcanzarlos. Por último, se proporciona un breve resumen de la estructura del documento.

1.1. Motivación

Las arquitecturas de CPU más modernas, además de estar configuradas con múltiples cores o hilos, admiten instrucciones SIMD con longitudes de vector ampliadas. Sin embargo, las limitaciones de la vectorización automática y la complejidad de la vectorización manual han supuesto un difícil aprovechamiento de las capacidades vectoriales en las implementaciones.

Una solución que resuelve estos problemas y facilita el procesamiento vectorial en las CPUs es la arquitectura SVE introducida por ARM. SVE, del inglés *Scalable Vector Extension*, es una extensión del conjunto de instrucciones SIMD de ARM cuya característica clave es la flexibilidad con la que permite elegir entre diferentes longitudes de vector para cada implementación [1]. Esta arquitectura, específicamente desarrollada para el procesamiento de alto rendimiento, permite realizar varias operaciones en una sola instrucción. En concreto, el número de operaciones depende tanto del tamaño del vector como del tipo de datos de la operación. De este modo, aunque la implementación requiere una cantidad significativa de recursos hardware, escalar la longitud de vector, por lo general, está relacionado con una mejora en el rendimiento.

Además, la arquitectura SVE presenta una serie de ventajas sobre las instrucciones SIMD convencionales como la introducción de nuevas operaciones, como *gather-load* y *scatter-store*, que permiten operar sobre datos no contiguos en memoria, o la incorporación de nuevos registros de predicado, que permiten determinar sobre qué elementos de un vector se deben realizar las operaciones vectoriales. Sin embargo, una de las mayores ventajas características de SVE es su modelo de programación VLA, del inglés *Vector Length Agnostic*, que permite ejecutar un mismo código en cualquier implementación de la arquitectura para cualquier longitud de vector sin necesidad de recompilarlo, es decir, garantiza la compatibilidad binaria posibilitando la compilación de binarios genéricos sin definir la longitud de vector.

Para realizar investigaciones sobre distintas implementaciones de arquitecturas en los procesadores siempre es necesario el uso de simuladores, debido a la no viabilidad del desarrollo de una implementación real de un procesador para cada configuración a estudiar. Los simuladores proporcionan el entorno adecuado para llevar a cabo investigaciones, tanto del rendimiento como de la eficiencia energética que tendrían dichas implementaciones. Uno de los simuladores más usados por la comunidad científica para el modelado arquitectónico de computadores es gem5 [2], una plataforma modular de código abierto que permite la simulación de diversas configuraciones en varias ISAs como ARM.

En este trabajo, se ha realizado una introducción al uso de arquitecturas SVE en el simulador gem5. En concreto, se ha detallado una metodología para la realización de diferentes simulaciones en modo *Full-System* que permiten la obtención de unos resultados preliminares para la evaluación del impacto en el rendimiento de diferentes benchmarks al escalar tanto la longitud de vector de SVE como el número de procesadores fuera de orden utilizando el simulador gem5.

1.2. Objetivos

El objetivo perseguido con el desarrollo de este trabajo ha sido la elaboración de una metodología de simulación de la arquitectura vectorial SVE de ARMv8 que permita dotar a nuevos investigadores de un mecanismo ágil y relativamente sencillo para desarrollar y probar nuevas propuestas arquitecturales en el simulador gem5. La sistematización del trabajo que proporciona la documentación exhaustiva de esta metodología puede ser muy útil y servir de base de trabajo para futuras investigaciones en este campo, además de ahorrar gran cantidad de tiempo a nuevos usuarios que se enfrenten a gem5, dada la gran complejidad que presenta esta herramienta.

Dicha metodología se puede dividir en tres fases; configuración del entorno, realización de simulaciones y análisis de resultados. Para su desarrollo, se han utilizado las herramientas desarrolladas por el grupo de investigación Arquitectura y Tecnología de Computadores (ATC) de la Universidad de Cantabria compuestas tanto por el clúster Triton, que aloja el simulador, como por algunos scripts de Python que facilitan la ejecución de conjuntos de simulaciones o la elaboración de gráficas para un análisis de resultados con mayor comodidad. Además, todo el trabajo se ha realizado de forma remota, sin acceso directo al clúster, por ser el caso de uso más general y la forma de acceso habitual en alumnos de TFG.

1.3. Plan de trabajo

A continuación, se incluyen las fases a seguir para el desarrollo del trabajo.

En la fase inicial, se debe realizar un estudio de las herramientas utilizadas. Para ello, es necesario investigar sobre la arquitectura SVE y el simulador gem5, así como familiarizarse con su estado del arte revisando otros trabajos relacionados.

En la segunda fase, se ha de realizar la configuración del entorno de trabajo a utilizar mediante la instalación y gestión de gem5 en el clúster Triton. Esta fase se realizará en remoto, es decir, sin acceso directo al clúster, puesto que, además de ser la forma de trabajo habitual en alumnos de TFG, es la única posible durante la estancia del Programa de Movilidad SICUE.

En la tercera fase se han de desarrollar los benchmarks a evaluar en las simulaciones. Primero, se han de programar y probar los códigos. Posteriormente, se deben compilar para su vectorización según la arquitectura SVE, lo que implica la gestión de la imagen de disco.

En la cuarta fase, se han realizar las diferentes simulaciones. Para ello, previamente, es necesario generar los *checkpoints* correspondientes y, a continuación, ejecutar cada una de las simulaciones.

Por último, la quinta fase comprende la generación y el análisis de diferentes gráficas para la evaluación de los resultados obtenidos tras las simulaciones.

1.4. Estructura del documento

El contenido de este trabajo se organiza de la siguiente manera:

El *Capítulo 2* describe las herramientas utilizadas; la arquitectura SVE y el simulador gem5, y presenta algunos ejemplos relevantes del estado del arte en el contexto de este trabajo.

El *Capítulo 3* recoge toda la información necesaria y el proceso de desarrollo seguido para la simulación de la arquitectura SVE con el simulador gem5. Este proceso

incluye la configuración del entorno de trabajo de gem5, el desarrollo de los benchmarks de SVE a evaluar y la ejecución de las simulaciones.

El *Capítulo 4* propone el análisis y la evaluación de los resultados obtenidos tras la realización de diferentes simulaciones, así como un resumen de la configuración utilizada.

El *Capítulo 5* resume las conclusiones alcanzadas y proporciona posibles indicaciones para un trabajo futuro.

Capítulo 2. Herramientas y Estado del Arte

En este apartado, se realiza una introducción de las herramientas utilizadas en este trabajo; la arquitectura SVE y el simulador gem5, realizando, además, un repaso del estado del arte sobre ellas.

2.1. Arquitectura SVE

ARM es una familia de arquitecturas de conjunto de instrucciones reducido o RISC, según sus siglas en inglés, para procesadores de computador [3]. Los procesadores RISC están diseñados con un número menor de instrucciones de computador, y más sencillas, que los procesadores de conjunto de instrucciones complejas o CISC. De este modo, consiguen operar a mayor velocidad, aumentando el número de MIPS o millones instrucciones por segundo ejecutadas y, por tanto, el rendimiento.

Las arquitecturas vectoriales emplean instrucciones de tipo vectorial, es decir, poseen una ALU segmentada que permite operar secuencialmente sobre registros vectoriales. Un registro vectorial contiene múltiples datos en vez de uno solo, como en el caso de un registro escalar. De este modo, con una sola instrucción, un procesador de arquitectura vectorial puede manejar vectores de datos.

SIMD, del inglés *Single Instruction Multiple Data*, es una técnica de procesamiento de datos en paralelo de la taxonomía de Flynn [4]. Los computadores SIMD aportan paralelismo a nivel de datos, gracias a numerosas estructuras de datos iguales, y paralelismo de grano fino, al trabajar sobre instrucciones individuales en lugar de sobre procesos completos.

Aunque tanto las arquitecturas vectoriales como las SIMD explotan el paralelismo a nivel de datos, el modo de hacerlo es diferente. Mientras que las arquitecturas SIMD procesan todos los elementos de una vez, es decir, ejecutan una misma instrucción simultáneamente sobre varios datos, las vectoriales lo hacen de forma segmentada sobre bloques de datos [5]. De este modo, una instrucción vectorial especifica una cantidad de trabajo equivalente a ejecutar un bucle, reduciendo de forma significativa el ancho de banda dinámico de las instrucciones y la frecuencia de los riesgos de control propios de los bucles, realizando paradas por cada operación vectorial en vez de por cada elemento del vector. Gracias a ello, las arquitecturas vectoriales aportan mayor eficiencia energética que las arquitecturas SIMD. También proporcionan menor latencia al usar patrones conocidos de acceso a memoria como el entrelazado, para el acceso segmentado a diferentes módulos de memoria paralelos, o el *prefetching*, para la predicción y obtención con antelación de los datos que van a ser solicitados. Además, el procesamiento vectorial es compatible con la transferencia de datos no contiguos, es decir, permite accesos a elementos de un mismo vector separados en memoria. Por tanto, en comparación, las arquitecturas vectoriales son más adecuadas para el paralelismo a nivel de datos, minimizan las dependencias y mejoran el rendimiento frente a las arquitecturas SIMD.

La arquitectura SVE, *Scalable Vector Extension*, es una extensión que incluye instrucciones vectoriales con un tamaño de vector escalable, del conjunto de instrucciones A64 de ARMv8-A, una de las arquitecturas de la familia ARM [6].

SVE se basa en un conjunto de vectores escalables e introduce nuevos registros y características arquitecturales para la vectorización. Entre los registros introducidos, hay 32 registros vectoriales escalables Z, 16 registros P de predicado escalable, un registro FFR de propósito especial, y varios registros ZCR_Elx de control del sistema de vector escalable [7].

Los registros Z son registros de datos con capacidad para almacenar elementos de 8, 16, 32, 64 y 128 bits, que pueden ser valores de tipo entero o de coma flotante en

simple, media o doble precisión. La ventaja principal que proporciona la arquitectura SVE es la libertad con la que permite definir el tamaño de los registros vectoriales Z0-Z31. En lugar de restringirlo a un valor fijo establecido como las arquitecturas SIMD, SVE ofrece un rango de posibles valores que abarca, en múltiplos de 128 bits, desde 128 hasta 2048 bits de longitud. De este modo, la arquitectura proporciona 16 posibles tamaños a elegir para definir la implementación de un procesador.

Los registros P son registros exclusivos de SVE [6]. Su tamaño es un octavo del de los registros vectoriales, ya que contienen 1 bit por cada byte disponible en un registro Z, es decir, son escalables en múltiplos de 16 bits. En estos registros se escriben predicados. Un predicado es un array unidimensional que actúa como máscara de bits para controlar si los elementos de un vector están activos o no. Un elemento, o *lane*, de un vector está *activo* si el bit menos significativo del predicado correspondiente se establece como verdadero, es decir, tiene valor 1. Los registros P0-P7 se encargan del control de la memoria general y las operaciones aritméticas, pero, junto con el resto, todos pueden ser usados tanto por las instrucciones generadoras de predicados como por las instrucciones que solo trabajan con predicados. Las instrucciones generadoras de predicado, como las condiciones vectoriales, utilizan las *flags* de condición *NZCV* con un significado diferente. Los bits *N* y *C* determinan si el primer o último elemento, respectivamente, está activo. El bit *Z* se establece si no hay ningún elemento activo y el bit *V*, al terminar la partición de un bucle vectorizado. Las instrucciones que solo trabajan con predicados, o instrucciones *predicadas*, manipulan los registros de predicado para determinar en qué elementos de un operando vectorial deben operar, ya que solo operan sobre los elementos activos. Esto aporta flexibilidad a las operaciones vectoriales.

El registro FFR, del inglés *First Fault Register*, es un registro de predicado especial usado por las instrucciones dedicadas de carga y almacenamiento de vectores, denominadas *first-fault*, que permiten accesos a memoria especulativos [1]. Al operar con esas instrucciones, el registro FFR se actualiza para indicar el éxito o fallo de la operación en cada elemento activo del vector.

Por último, los registros ZCR_Elx son registros de control de los vectores escalables, que pueden usarse por cada nivel de excepción (ELx) para restringir la longitud de vector en su nivel o en niveles menos privilegiados [1].

Asimismo, a diferencia de otras arquitecturas SIMD, SVE también introduce el concepto de programación agnóstica de longitud de vector o VLA, por sus siglas en inglés [6]. Esta programación tiene por objetivo posibilitar que una misma aplicación se ejecute en cualquier implementación de la arquitectura habilitada para SVE. Para ello, el código debe compilarse sin conocer la longitud vectorial, es decir, para la arquitectura genérica de SVE. Esto supone que los vectores y predicados no puedan ser inicializados en memoria y los contadores e incrementos de los bucles vectoriales sean desconocidos para el compilador. Posteriormente, en tiempo de ejecución, las instrucciones SVE permiten adaptar el código vectorial automáticamente a la longitud de vector definida por la implementación. De este modo, el código puede ejecutarse en cualquier hardware habilitado para SVE con una sola compilación.

Entre las características arquitecturales de SVE se encuentran la vectorización especulativa y la partición de vectores. También se incluyen nuevas instrucciones, como *gather-load* y *scatter-store*, que permiten la carga y el almacenamiento, respectivamente, de elementos de un vector en posiciones no contiguas de memoria [7]. Para evitar dependencias entre iteraciones de bucles dentro de un mismo vector, SVE incluye operaciones vectoriales horizontales. Estas operaciones son instrucciones de reducción horizontal entre las que se incluyen las reducciones de enteros, de coma flotante y lógicas a nivel de bit [7].

La vectorización especulativa es una técnica de división de los bucles en diferentes regiones controladas cada una por un predicado [7]. Estas regiones representan operaciones que siempre son seguras de ejecutar, instrucciones para el cálculo de predicados de las salidas condicionales e instrucciones posteriores a la salida condicional. Para las dos últimas regiones, se utilizan operaciones de partición de vector y de carga *first-fault*. Una partición de vector es una porción del vector definida dinámicamente por un registro de predicado [6]. La técnica de partición de vectores permite que los bucles operen solo sobre particiones seguras hasta procesar todo el vector o terminar el bucle. En concreto, es un mecanismo para la gestión de las condiciones dinámicas de los bucles sin contador de iteraciones explícitos con salidas dependientes de los datos, como en caso de *do-while o break*, al operar sobre vectores. La vectorización especulativa es tolerante a fallos, ya que genera predicados para indicar los elementos del vector cargados con éxito antes del primer fallo de memoria, permitiendo la vectorización segura de bucles con salidas condicionales o contadores desconocidos y manteniendo el mismo comportamiento ante los fallos que en una ejecución secuencial [6]. Esto permite optimizar bucles para el procesamiento de grandes cantidades de datos, favoreciendo un mayor paralelismo, lo que convierte a la arquitectura SVE en idónea para la computación de alto rendimiento (HPC) y el aprendizaje automático (ML) [1].

2.2. Gem5

Gem5 es una plataforma modular de simulación de sistemas informáticos por eventos discretos [2]. Está escrito principalmente en C++ y Python y se proporciona bajo una licencia de software libre basada en BSD, que garantiza el acceso sin restricciones legales. Puede simular diferentes configuraciones y cargas de trabajo en una variedad de ISAs entre las que se encuentran x86, Alpha, MIPS, Power, SPARC y ARM. Por tanto, es compatible con la ISA ARMv-8 y, consecuentemente, con SVE.

El principal objetivo del simulador gem5 es servir de herramienta para el modelado arquitectónico de computadores, ya que está orientado a la simulación y evaluación de una amplia gama de sistemas informáticos [8] [9]. En función de las necesidades, los componentes que la conforman se pueden reorganizar, parametrizar, extender o reemplazar con facilidad para construir el sistema de simulación. Para alcanzar estas prestaciones, se ha sustentado en tres pilares; un modelado flexible, una gran disponibilidad y un alto nivel de interacción entre los desarrolladores [10].

El modelado flexible permite evaluar sistemas con diferentes niveles de detalle, equilibrando entre velocidad y precisión en la simulación. De este modo, el simulador puede ser usado para distintos tipos de experimentos con diferentes requisitos de simulación. Las capacidades de gem5 que posibilitan esta flexibilidad y utilidad se agrupan en tres dimensiones clave; el modelo de CPU, el modo de sistema y el sistema de memoria.

Para el modelo de CPU hay cuatro opciones diferentes [10]:

- *AtomicSimple* es un modelo mínimo de CPU, con una única comunicación entre procesos (IPC), que completa todos los accesos a memoria de manera inmediata, ejecutando una única instrucción en cada ciclo, es decir, con poca sobrecarga.
- *TimingSimple* es un modelo simple similar al anterior que permite una única petición a memoria a la vez. Esta CPU modela los tiempos de acceso a memoria.

- *InOrder* es un modelo de CPU segmentada en orden que enfatiza la sincronización de instrucciones y la precisión de la simulación. Este modelo permite configurar el número de etapas de canalización e hilos hardware.
- *O3* es un modelo de CPU segmentada fuera de orden que simula las dependencias entre instrucciones, unidades funcionales, accesos a memoria y etapas de canalización. En este modelo, la ejecución de instrucciones solo se realiza en la etapa de ejecución tras la resolución de todas las dependencias. *O3* permite parametrizar recursos para simular arquitecturas superescalares con múltiples hilos hardware, o SMT según sus siglas en inglés.

Cada modelo de CPU puede simular en dos modos de sistema [10]. El modo *System-call Emulation (SE)* emula la mayoría de los servicios del sistema sin necesidad de modelar dispositivos o un sistema operativo completo. Por el contrario, el modo *Full-System (FS)* modela el hardware necesario para simular un sistema completo con dispositivos y sistema operativo. *SE* está orientado a simular programas del espacio de usuario, mientras que *FS* se orienta a la ejecución de cualquier tipo de instrucciones, tanto de nivel de usuario como de kernel. Por ello, el modo *Full-System* no solo es indicado para el estudio del comportamiento subyacente de algoritmos o piezas hardware, sino también para la realización de pruebas de nuevos diseños conceptuales de la arquitectura del sistema. Para simular los dispositivos de forma realista en el modo *FS*, es necesario proporcionar una imagen de disco que contenga la estructura y los bytes del dispositivo de almacenamiento del mismo modo en el que aparecerían en un dispositivo hardware real [11]. La imagen de disco y el kernel del sistema operativo se pueden proporcionar por separado. Sin embargo, la simulación en modo *FS* habitualmente exige mucho más tiempo de simulación [11]. Para reducirlo, se pueden utilizar *checkpoints* o puntos de control, que son capturas de la simulación en un momento específico, normalmente después de haber arrancado el sistema operativo, y desde las que se puede reanudar la simulación posteriormente para evitar el proceso de arranque.

El sistema de memoria también consta de dos modelos [10] [12]. El modelo *Classic* proporciona un sistema de memoria rápido y de configuración sencilla que permite la construcción de una jerarquía de memoria sin necesidad de detallar el protocolo de coherencia, ya que implementa un único protocolo de coherencia denominado MOESI , simplificado y que no se puede modificar . Por otro lado, el modelo *Ruby* es más detallado y proporciona una infraestructura flexible capaz de simular con precisión una amplia variedad de sistemas de memoria con coherencia de caché. Para ello, *Ruby* permite definir nuevos protocolos de coherencia de caché y modelarlos con alta fidelidad.

Como se aprecia en la *Figura 2.1*, en función del modelo de CPU y el sistema de memoria elegidos, se obtiene un rendimiento diferente en el espectro velocidad-precisión, siendo estas magnitudes inversamente proporcionales. Además, para un mismo modelo de CPU, se alcanza mayor velocidad con el modo de simulación *SE,* mientras que con el modo *FS* prima una mayor precisión.

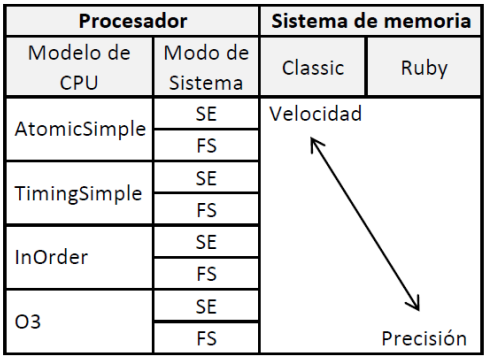


***Figura 2.1**. Espectro velocidad-precisión de gem5.*

La amplia disponibilidad que otorga su licencia ha impulsado el acceso a gem5 a usuarios con distintos objetivos y requisitos, que tanto en el mundo académico y científico como en el industrial y empresarial, pueden hacer uso del simulador sin renunciar a los derechos de autor o al crédito por sus contribuciones o investigaciones. Sin embargo, a pesar de ser el simulador por excelencia y el más usado por la comunidad científica y académica, gem5 cuenta con algunas limitaciones entre las que se incluyen largos tiempos de ejecución en las simulaciones y numerosos errores conocidos y reportados a lo largo del tiempo.

2.3. Trabajos relacionados

En esta sección se recopilan distintas publicaciones de estudios realizados sobre la arquitectura SVE.

Las operaciones *stencil* o de plantilla son patrones regulares para el acceso a grandes volúmenes de datos y son la base de aplicaciones HPC para diferentes ámbitos científicos como la dinámica de fluidos, la mecánica estructural o el procesamiento de imágenes. Armejach *et al.* [13] aprovechan las características principales de SVE para implementar simulaciones con diferentes longitudes de vector para distintas optimizaciones de las operaciones *stencil*, como el desenrollado y la fusión de bucles, el intercambio de cargas y la reutilización de datos, demostrando que se puede conseguir una mejora significativa en el rendimiento de hasta 1,57 veces.

Por su parte, Zhong *et al*. [14] analizan el rendimiento de las implementaciones con SVE para diferentes longitudes de vector de dos optimizaciones de operaciones MPI; el empaquetado y desempaquetado de tipos de datos basado en *gather-scatter* y la reducción MPI. Sus evaluaciones demuestran, en el simulador ARMIE, reducciones de hasta 30 veces el número de instrucciones, y en el procesador A64FX de Fujitsu, el primero y único conocido que utiliza la tecnología SVE, mejoras en el rendimiento de hasta 4 veces mayor velocidad.

DaCe es un marco de programación que permite optimizar el código en una representación intermedia centrada en los datos e independiente de la plataforma antes de exportarlo a una plataforma específica. Scholbe [15] desarrolla DaCe SVE, una extensión de DaCe para el objetivo SVE de ARM capaz de generar código que emite instrucciones que utilizan las capacidades únicas de SVE a partir de una transformación intermedia de vectorización. Los resultados obtenidos demuestran que el rendimiento de la transformación y la generación de código de su contribución están a la par, y superan en algunos casos, la auto-vectorización del compilador C++ de Fujitsu.

SPIRAL es un sistema desarrollado por la Universidad Carnegie Mellon que genera automáticamente implementaciones de alto rendimiento de algoritmos matemáticos. Kitai *et al*. [16] proponen un sistema de autoajuste adaptando SVE para que SPIRAL genere implementaciones de la transformada discreta de Fourier o DFT, según sus siglas en inglés; una transformación matemática de una señal muestreada en los dominios discretos de tiempo y de frecuencia utilizada en el procesamiento de señales. Los códigos DFT aplicados por SVE evaluados alcanzan hasta 1,98 veces mayor rapidez y 3,63 veces mayor ratio de instrucciones SIMD que el código escalar, obteniendo un speedup máximo de 2,32.

Finalmente, como ejemplo de un estudio con un objetivo similar a este trabajo, de evaluación del rendimiento de programas de SVE con distintas configuraciones hardware en el simulador gem5, además del consumo de energía con la herramienta McPAT, está el trabajo de Odajima *et al.* [17]. En él, examinan el efecto de diferentes longitudes de vector de SVE y cantidades de procesadores fuera de orden en el rendimiento y consumo energético empleando dos programas con diferentes características. El programa *N-body* que integra la ecuación de movimiento de Newton basada en las interacciones gravitatorias entre múltiples partículas, tiene considerables cantidades de operaciones de coma flotante en comparación con las de carga y almacenamiento. Los resultados de su evaluación permiten observar que este programa, con una larga cadena de instrucciones, mejora el rendimiento en más de un 30% y el consumo de energía en casi un 21% al aumentar la longitud de vector, en concreto, para 1024 bits, minimizando también el aumento de la cantidad de recursos hardware necesarios, respecto de longitudes más cortas como 512 bits. El segundo programa, *Stream Triad*, mide el rendimiento de la caché o del ancho de banda de la memoria. En este programa de memoria limitada, que posee una mayor proporción de operaciones de carga y almacenamiento respecto de las de coma flotante y cuyo rendimiento está limitado por el ancho de banda de la caché o de la memoria, por el contrario, se observa que la longitud del vector y la configuración del hardware no afecta al rendimiento.

Capítulo 3. Simulación de la Arquitectura SVE con Gem5

En este capítulo se propone una metodología para la realización de simulaciones de la arquitectura ARM SVE con el simulador gem5 y las herramientas desarrolladas en el grupo ATC. La *Figura 3.1* muestra el diagrama de flujo con los pasos realizados, agrupados por tonalidad de color en secciones. Para las diferentes subsecciones se utilizan distintas escalas de color dentro de la misma tonalidad. Además, los pasos están organizados en dos niveles correspondientes a la forma de trabajo, ya que algunos han sido realizados en remoto en Triton y otros, de manera local, en la máquina nativa. En los pasos de la *Sección 3.1*, correspondiente a la tonalidad azulada en la figura, se establece el entorno de trabajo de gem5. A continuación, en la *Sección 3.2*, en tono colorado, se realiza el desarrollo de los diferentes benchmarks de SVE, para, posteriormente, en la *Sección 3.3*, en dorado, realizar los pasos del ciclo de simulación. Finalmente, en el *Capítulo 4*, en color verde, se realiza la evaluación de resultados.

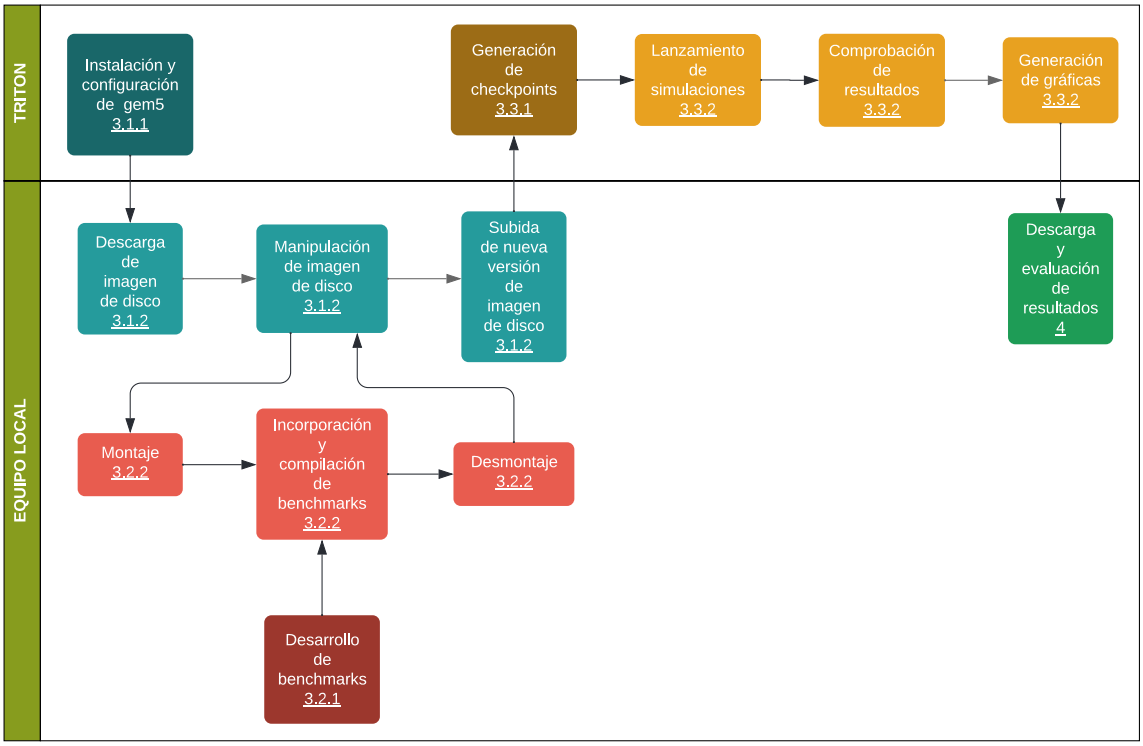


***Figura 3.1**. Diagrama del flujo de pasos seguidos para la simulación de benchmarks en gem5.*

3.1. Entorno de Trabajo Gem5

Para el desarrollo del trabajo se ha utilizado el simulador gem5 instalado en el clúster Triton del Grupo Arquitectura y Tecnología de Computadores (ATC) de la Universidad de Cantabria, cuyo acceso está restringido a la VPN de dicha institución. Por este motivo, todo el trabajo se ha realizado en remoto a través de la conexión a las Aulas de Informática proporcionada por la plataforma Unican Labs.

3.1.1. Instalación y configuración de gem5

Tras la conexión a Tritón por primera vez, ha sido necesaria la instalación de gem5, en el directorio *home* propio de usuario, y de las dependencias a utilizar, en este

caso, *Python* y *SCons*. Para completar la instalación, además de compilar gem5 con *SCons*, ha sido preciso establecer las variables de entorno con las rutas adecuadas en el archivo de configuración *triton.json* del directorio *simple_launcher*. Este archivo JSON contiene cuatro entradas importantes:

- La primera, *system_name*, determina el nombre del sistema, cuyo uso es meramente para depuración.
- La segunda, *hostnames*, define la lista de nombres de *hosts* asignados al sistema que utilizan las variables de entorno.
- Por su parte, *job_class* especifica el tipo de gestor de colas de trabajos, que en caso de Triton es SLURM.
- Finalmente, en la entrada *gem5* se especifican las cuatro variables de entorno de gem5. La variable *M5_PATH* apunta al directorio base con los binarios requeridos por las simulaciones *FS*, así como los gestores de arranque, *kernels* o imágenes de disco. La variable *GEM5_PATH* establece la ruta del directorio base de gem5 con su código fuente, que se puede descargar del repositorio oficial. Por último, *SCRIPTS_PATH* apunta al directorio *simple_launcher* mientras que *WORKING_DIR* al directorio base para las salidas de las simulaciones.

3.1.2. Manipulación de imagen de disco

Una vez instalado y configurado gem5, para probar un benchmark de SVE propio es necesario incluirlo en la imagen de disco. El proceso para manipular una imagen de disco requiere montarla y desmontarla. Sin embargo, esto no es posible en Triton sin privilegios de Supervisor, por lo que todo el trabajo con la imagen de disco se ha realizado en el equipo local. Para esta alternativa, ha sido necesario instalar un entorno de terminal de Ubuntu para el sistema Windows local mediante WSL, el subsistema de Windows para Linux [18]. De este modo, trabajar con la imagen de disco ha implicado descargarla en la máquina anfitriona de usuario, montarla, manipularla y desmontarla desde la PowerShell local, y posteriormente, subir la nueva versión a Triton.

La imagen utilizada para este trabajo ha sido la proporcionada por el gem5 instalado en Triton, en concreto, *ubuntu-21_04.img*, que incluye el sistema de archivos de Ubuntu y contiene versiones recientes de los compiladores GNU necesarios para generar el código SVE.

Para manipular la imagen se ha utilizado QEMU, un emulador de ISAs que permite ejecutar *chroot* para cambiar el directorio raíz a la imagen del disco [19]. Esto funciona incluso cuando las ISAs del sistema anfitrión y de los programas en la imagen de disco son diferentes, al menos con imágenes ARM en sistemas x86-64 [20].

A continuación, se describe el procedimiento seguido para modificar la imagen de disco. Primero, ha sido necesario montar la imagen de disco en un directorio temporal. Para ello, se ha utilizado el script *gem5img.py* proporcionado por gem5 en el directorio *util* para la construcción y manejo de imágenes. Tras el montaje, se han copiado en ella, en la ruta */benchmarks/custom_benchmarks/threads/*, los benchmarks de SVE desarrollados. Posteriormente, desde dentro del directorio de montaje, se ha ejecutado *chroot* para poder manipular el contenido de la imagen de disco. En este caso, se ha realizado la compilación de los benchmarks correspondientes según los pasos detallados en la *Subsección 3.2.2.* Finalmente, una vez realizada la compilación, ha sido necesario cerrar *chroot* y salir del directorio de montaje antes de poder desmontar la nueva versión de la imagen con el mismo script y subirla a Triton.

3.2. Desarrollo de los benchmarks de SVE

En este apartado se describe el proceso de desarrollo de los diferentes benchmarks de SVE sobre los que se han realizado las pruebas en el simulador. Primero, se detalla el objetivo de cada benchmark y se explican los aspectos importantes de la programación de su código. Luego, se especifican los pasos necesarios para su compilación en la imagen de disco y para la comprobación de la correcta vectorización del código.

3.2.1. Objetivos y código de cada benchmark

Para el desarrollo de este trabajo se han implementado dos benchmarks paralelos mediante programación en C++ basada en hilos y memoria compartida. Además, cada benchmark está a su vez vectorizado, con un tamaño de vector que puede variar, según la especificación de la arquitectura SVE explicada en la *Sección 2.1*.

En este apartado, se describe el desarrollo de dichos benchmarks para su posterior evaluación en el simulador. Para la parte común de ambos, se ha seguido una plantilla que hace de estructura. Primero, reciben como parámetros de entrada el número de hilos y el tamaño de las matrices, definidas como vector de vectores, y comprueban que estos valores son compatibles con su problema. A continuación, inicializan los datos, distribuyen la carga en los diferentes hilos y, tras la finalización de todos, comprueban que el resultado obtenido sea correcto. Siguiendo esta estructura, cada benchmark define su propia implementación de la función de los hilos acorde a su problema.

El primer benchmark desarrollado ha sido *matrix*, que realiza la multiplicación repartida en hilos de dos matrices cuadradas; A y B. En este caso, cada hilo recibe una porción de la matriz A correspondiente a las filas sobre las que opera y toda la matriz B para la multiplicación. La *Figura 3.2* muestra un ejemplo del reparto de los datos entre los hilos en la que el primer hilo recibe las *i* primeras filas de la matriz A y toda la matriz B, almacenando, por tanto, el resultado de la multiplicación en las *i* primeras filas de C. El valor *i* de la figura se calcula dividiendo el ancho de la matriz, *n* en la figura, entre el número de hilos.

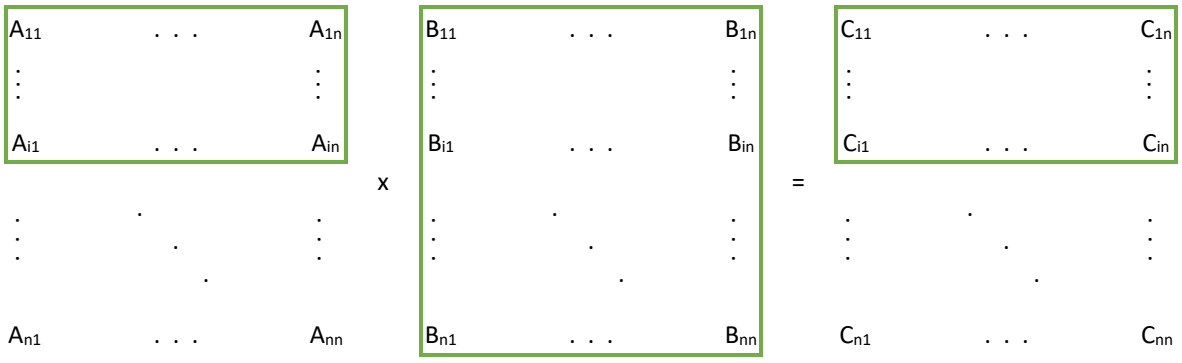


Figura 3.2. *Representación del reparto de los datos entre los hilos en el benchmark matrix.*

El segundo benchmark desarrollado ha sido *gauss*, que propone la aplicación de un filtrado de Gauss bidimensional, repartida en hilos, sobre una imagen, representada por una matriz cuadrada de píxeles. Cada píxel de la matriz es inicializado a un valor aleatorio entre 0 y 255 y recalculado como una media ponderada de sí mismo y de sus 24 vecinos, aplicando el filtro gaussiano que representa la matriz 5x5 de la *Figura 3.3*, cuya suma de valores es 269.

1	4	6	4	1
4	16	26	16	4
6	26	41	26	6
4	16	26	16	4
1	4	6	4	1

Figura 3.3. *Filtro Gaussiano aplicado en el benchmark gauss.*

De esta forma, el filtro solo se aplica sobre los píxeles con 24 vecinos, es decir, los que no pertenecen a las filas y columnas del perímetro de 2 elementos de anchura de la matriz de píxeles, cuyo valor se mantiene sin recalcular. Por tanto, para que cada hilo aplique el filtrado sobre la misma cantidad de filas de elementos, el número de hilos especificado debe ser un divisor del número de elementos de la matriz menos cuatro. La *Figura 3.4* muestra un ejemplo del reparto de los píxeles entre los hilos. En ella se aprecia que las 2 líneas y 2 columnas del perímetro exterior de la imagen I, sombreadas, no se modifican, por lo que solo se opera sobre la matriz interior, es decir, los píxeles sobre los que se puede aplicar el filtro completo, que se reparten entre el número de hilos. De esta forma, el primer hilo recibe las *i-2* primeras filas de la matriz interior de la imagen I, marcadas en la figura. El valor *i-2* se calcula dividiendo el ancho de la matriz interior, *n-4*, siendo *n* el ancho de la imagen I, entre el número de hilos.

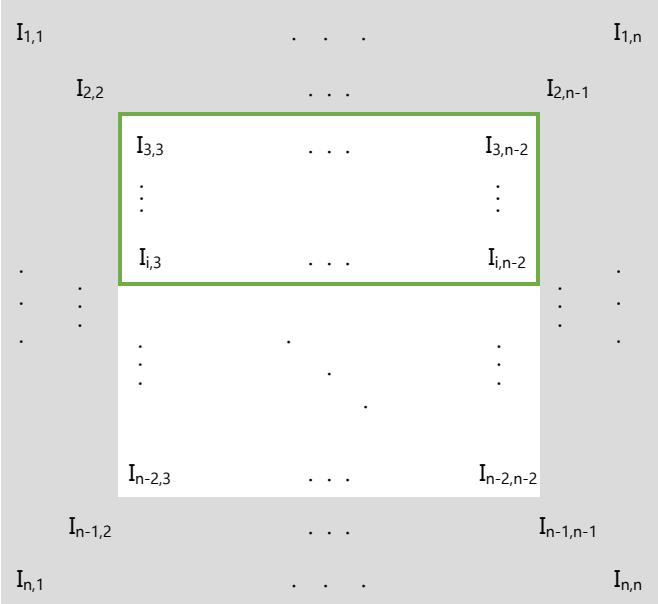


Figura 3.4. *Representación del reparto de los píxeles entre los hilos en el benchmark gauss.*

La función realizada por cada hilo consiste en la aplicación del filtrado gaussiano sobre los píxeles que ha recibido, y está detallada en la *Figura 3.5*.

```
1.  oid gauss(vector<vector<int>> &filter, vector<int> &pixels, vector<int>
    &filtered_gauss, int start, int end, int height) {
2.
```

```
3.      int x,y,dx,dy;
4.      int sum;
5.      for (x = start; x < end; x++) {
6.          for (y = 2; y < (height-2); y++) {
7.              sum = 0;
8.              for (dx = 0; dx < 5; dx++) {
9.                  for (dy = 0; dy < 5; dy++) {
10.                     sum += pixels[(x+dx-2),(y+dy-2)] * filter[dx][dy];
11.                 }
12.             }
13.             filtered_gauss[x*height+y] = (int) sum / 269;
14.         }
15.     }
16.
17.     return;
18. }
```

***Figura 3.5**. Código de la función de los hilos en el benchmark gauss.*

3.2.2. Compilación

En esta sección se describe el proceso de compilación de los benchmarks SVE dentro de la imagen de disco. Cada benchmark se compone de un directorio en el que, además del fichero *main.cpp* con su código fuente, explicado en el apartado anterior, se encuentran los archivos *gem5*, *libm5.a* y *Makefile*.

Las operaciones *m5*, o *m5ops*, son rutinas con instrucciones especiales que solo pueden ser interpretadas por gem5. Gracias a ellas, el benchmark puede interactuar con el simulador para, por ejemplo, la toma de *checkpoints* o el reseteo de estadísticas [20]. El archivo *lib5.a* es el binario de la librería estática para ARMv8 que contiene las rutinas *m5*, mientras que *gem5* es un directorio con las cabeceras de esta librería. Estos archivos permiten el uso de la compilación condicionada para que el mismo código fuente se pueda usar tanto en gem5 y como en una máquina nativa, de forma que sea sencillo depurar cada benchmark antes de incluirlo en el simulador.

Makefile es el archivo de construcción *make*, que realiza la compilación del benchmark en dos versiones; escalar y vectorial.

En ambas versiones se especifica *CFLAGS*, como se muestra en la *Figura 3.6*, que incluye la opción de optimización *-O3*. Esta opción proporciona la máxima optimización posible del compilador GNU para intentar mejorar el rendimiento, el tamaño del código y el tiempo de ejecución [21]. Además, *CFLAGS* añade el directorio actual del benchmark a la lista de inclusión para añadir las cabeceras de gem5 en el *main.cpp*. También añade las invocaciones de las funciones *m5ops*, en concreto, *m5_reset_stats* y *m5_dump_stats*, ubicadas al comienzo y al final de la ROI de cada benchmark. La Región de Interés, o ROI según sus siglas en inglés, delimita la sección relevante del código. En este caso, se ha definido por la parte a tener en cuenta para las estadísticas de gem5, es decir, los cálculos realizados para la creación de hilos y el trabajo realizado por estos hilos, excluyendo de la ROI, por tanto, la inicialización de los datos y la comprobación final de resultados. De este modo, las funciones mencionadas restablecen las estadísticas antes de la ROI y las vuelcan tras esta.

```
1.  CFLAGS=-O3
2.  CFLAGS+=-DGEM5 libm5.a -I./
```

***Figura 3.6.** Opción CFLAGS del archivo Makefile de cada benchmark.*

Para la versión vectorial, además, se utiliza la opción *-march=armv8-a+sve*, que genera el binario del programa que contiene las instrucciones SVE.

Una vez montada la imagen de disco, se ha realizado la compilación con las herramientas de GNU de cada benchmark proporcionada por su *Makefile* en dos fases. En la primera fase, se ha prescindido de las *CFLAGS*, es decir, se ha efectuado la compilación estándar sin gem5. Esto ha permitido la ejecución normal del benchmark desde la terminal con diferentes parámetros para comprobar tanto la funcionalidad como la escalabilidad del código, de forma que se simplifica la fase de programación y depuración. A continuación, en la segunda fase, se ha compilado incorporando la opción *CFLAGS* para la generación del código de SVE a probar en las simulaciones.

Para comprobar el uso de las instrucciones vectoriales, tras la compilación del benchmark, se ha empleado la herramienta *objdump*. Con ella, se ha realizado la búsqueda de instrucciones vectoriales estáticas, es decir, las que aparecen en el código ensamblador que utilizan los registros Z en el binario del programa SVE y se ha comprobado el porcentaje de vectorización del código, comparándolo con el binario en versión escalar sin SVE. En la *Figura 3.7* se reflejan los resultados de vectorización de ambos benchmarks. El código del benchmark *matrix* consigue una vectorización del 5'56% con SVE frente al 4'25% sin SVE, mientras que el del benchmark *gauss*, alcanza hasta un 6'02% frente a un 4'09%. Es decir, en el código *gauss* se usa ligeramente más la vectorización que en *matrix*.

Binario	Métrica		Benchmark	
			matrix	gauss
Sin SVE	Nº instrucciones	Con registros Z	188	187
		Totales	4425	4570
	% vectorización		4,25	4,09
Con SVE	Nº instrucciones	Con registros Z	245	257
		Totales	4410	4272
	% vectorización		5,56	6,02

***Figura 3.7**. Resultados de vectorización de ambos benchmarks.*

3.3. Ciclo de simulación

Tras la incorporación en Triton de la nueva versión de la imagen de disco que incluye los binarios compilados de los benchmarks de SVE desarrollados, se pueden realizar simulaciones. Para este trabajo se han planteado simulaciones *Full-System* para 1, 2, 4, 8 y 16 cores con longitudes de vector de SVE de 128, 256, 512, 1024 y 2048 bits, es decir, 25 instancias de simulación para cada benchmark, una por cada combinación de parejas de número de cores con longitudes de vector.

Debido a que el arranque del sistema operativo es un proceso significativamente largo, para la ejecución de simulaciones en modo *Full-System* con rangos de tiempo más manejables es necesario el uso de *checkpoints*. En concreto, se requiere un *checkpoint* por cada configuración a simular, es decir, por cada combinación diferente de número de cores y longitud de vector de SVE, ya que, como se explicó en la *Sección 2.2*, cada *checkpoint* es una captura de la simulación tras el arranque del sistema operativo con su configuración especificada. De este modo, cada simulación posterior se restaura desde su *checkpoint* correspondiente para realizar únicamente la ejecución de los benchmarks, ahorrando tiempo de simulación. Por tanto, un mismo *checkpoint*, definido para una

longitud de vector y un número de cores determinados, puede utilizarse para diferentes simulaciones en las que, por ejemplo, se seleccione ejecutar distintos benchmarks de los disponibles en la imagen de disco o se cambien los parámetros de entrada.

Debido a que los *checkpoints* guardan el estado del sistema en un momento determinado, se han de generar tras cada modificación de la imagen de disco.

Para la ejecución de las simulaciones, tanto de generación de *checkpoints* como de evaluación de benchmarks, se ha utilizado el script de Python3 *launcher.py* del entorno gem5 de ATC-UC. Este script se encuentra en el directorio *simple_launcher* y permite ejecutar un conjunto de simulaciones desde su configuración de experimento detallada en un archivo JSON, que recibe como parámetro. Estos archivos contienen seis parámetros obligatorios en el nivel superior:

- *sim_dir* indica el directorio de salida para todas las simulaciones derivadas del archivo JSON.
- *name* es el nombre asignado a la simulación utilizado por el gestor de colas de los trabajos y por el script *launcher.py* para denominar al subdirectorio de salida generado dentro de *sim_dir* con los resultados de la simulación.
- *gem5_bin* apunta al binario con la versión de gem5 utilizada. En este trabajo se ha utilizado la versión de construcción *gem5.opt* que incluye optimizaciones.
- *config_file* apunta al script de configuración del sistema de gem5 encargado de la construcción del sistema a simular mediante la instanciación de los modelos hardware. En este trabajo, para la ejecución de simulaciones *Full-System* se ha utilizado el script *fs.py* del repositorio oficial de gem5.
- *params* establece valores para las posibles opciones definidas por *fs.py* en forma de diccionario. La configuración de este trabajo utiliza el conjunto de instrucciones ARMv8 para ejecutar binarios con instrucciones SVE. El sistema simulado arranca la versión 5.15.10 de Linux para ejecutar Ubuntu 21.04 desde la imagen de disco correspondiente. Por tanto, el binario especificado en el parámetro *bootloader* es *boot_v2.arm64*, y en *kernel*, *vmlinux_5.15.10*. El tipo de máquina utilizada es *VExpress_GEM5_V2*, que modela la disposición de 64 bits de ARM.
- *--script* es un parámetro de *fs.py* que apunta a los scripts de Shell *.rcS* a ejecutar por el sistema simulado tras el arranque de Linux.

Los experimentos de las configuraciones a simular se definen en la ruta *unican-util/simple_launcher* del gem5 de Triton. En el subdirectorio *boot* de los experimentos se define el conjunto correspondiente a las simulaciones de generación de *checkpoint*, mientras que en el subdirectorio *benchmark*, el de las simulaciones de evaluación de los benchmarks a partir de los *checkpoints*.

A continuación, se describe el proceso necesario para la generación de *checkpoints* y la ejecución de las simulaciones.

3.3.1. Generación de *checkpoints*

La *Figura 3.8* muestra un ejemplo del contenido de uno de los archivos JSON para la generación de *checkpoints*, en concreto, el de la configuración para 1 core y 128 bits de longitud de vector, *cores_1_vl_128.json*. En sus parámetros destaca la ausencia de caché y la definición del uso del modelo de CPU más simple, *AtomicSimpleCPU*, es decir, se han utilizado los tipos de CPU y memoria más sencillos para agilizar al máximo la generación de los *checkpoints*. Además, se especifica la versión de la imagen de disco

con los benchmarks a simular posteriormente incluidos. El script utilizado es *hack_back_ckpt.rcS*, disponible en la ruta *gem5/rcS* del directorio *simple_launcher*. Este script está diseñado para generar *checkpoints* después de arrancar Linux y permite ejecutar otros scripts tras la restauración de esos *checkpoints*.

```
1.  {
2.      "sim_dir"          : "2022_05_w1",
3.      "name"             : "boot_cpus_1_sve_128",
4.      "gem5_bin"         : "build/ARM/gem5.opt",
5.      "config_file"      : "configs/example/fs.py",
6.
7.      "params" : {
8.          "--num-cpus"        : "1",
9.          "--cpu-type"        : "AtomicSimpleCPU",
10.         "--cpu-clock"         : "2GHz",
11.         "--mem-size"        : "2GB",
12.       "--bootloader"      : "$M5_PATH/binaries/boot_v2.arm64",
13.       "--kernel"          : "$M5_PATH/binaries/vmlinux_5.15.10",
14.       "--disk-image"      : "$M5_PATH/disks/ubuntu-21_04_v1.img",
15.       "--machine-type"    : "VExpress_GEM5_V2",
16.         "--param"           : "'system.sve_vl = 1'"
17.     },
18.
19.     "--script"  : {
20.       "hack_back" : "gem5/rcS/hack_back_ckpt.rcS"
21.     }
22. }
```

Figura 3.8. *Contenido del archivo boot/cores_1_vl_128.json para la generación de checkpoint.*

Tras la correcta definición de todos los archivos JSON de generación de *checkpoints*, se han lanzado los experimentos a la cola de trabajos con *launcher.py*. Una vez finalizadas las simulaciones, se han comprobado los directorios de salida de cada experimento. En ellos, se ha verificado la aparición de los directorios *cpt.<tick>*, la ausencia de errores en el fichero *.err* de la salida de error y la presencia de la línea *"Writing checkpoint"* antes de la llamada *"m5_exit"* del archivo *.out* de la salida estándar. Estas tres condiciones son indicativo del éxito en la creación de los *checkpoints*.

3.3.2. Ejecución de simulaciones

Una vez creados correctamente los *checkpoints* para cada una de las 25 configuraciones, se han lanzado las simulaciones correspondientes a la ejecución de los benchmarks. La *Figura 3.9* muestra un ejemplo del contenido del archivo JSON de la configuración de una de las simulaciones, en concreto, la correspondiente al *checkpoint* de la *Figura 3.8*, para 1 core y 128 bits de longitud de vector, *cores_1_vl_128.json*.

En este archivo, se establece el parámetro *--checkpoint-dir* que apunta al directorio donde se encuentra el *checkpoint* desde el que se retoma la simulación. El tipo de CPU utilizado en estos experimentos ha sido *ex5_big*, un modelo detallado con ejecución fuera de orden que emplea el sistema de memoria *Ruby* con cachés y *Garnet*, como modelo detallado de red en chip, o *Network on chip (NoC)*. Para todas las simulaciones se ha establecido la misma configuración; 1 core de procesamiento, 8 bancos de caché de nivel L3, 8 directorios, 2GHz de frecuencia de reloj para la CPU y 2GB de memoria principal.

```
1.  {
2.      "sim_dir"          : "2022_05_w1",
3.      "name"             : "cpus_1_sve_128",
```

```
4.      "gem5_bin"          : "build/ARM/gem5.opt",
5.      "config_file"       : "configs/example/fs.py",
6.
7.      "params" : {
8.        "--checkpoint-dir"     :
    "$WORKING_DIR/2022_04_w3/boot_cpus_1_sve_128/hack_back/",
9.          "--cpu-type"            : "ex5_big",
10.       "--restore-with-cpu"   : "ex5_big",
11.       "--checkpoint-restore" : "1",
12.         "--num-cpus"            : "1",
13.         "--num-l3caches"        : "8",
14.         "--num-dirs"            : "8",
15.         "--cpu-clock"             : "2GHz",
16.         "--ruby"                : "",
17.         "--network"             : "garnet",
18.         "--topology"            : "Pt2Pt",
19.         "--caches"              : "",
20.         "--mem-size"            : "2GB",
21.       "--bootloader"          : "$M5_PATH/binaries/boot_v2.arm64",
22.       "--kernel"              : "$M5_PATH/binaries/vmlinux_5.15.10",
23.       "--disk-image"          : "$M5_PATH/disks/ubuntu-21_04_v1.img",
24.       "--machine-type"        : "VExpress_GEM5_V2",
25.         "--param"               : "'system.sve_vl = 1'"
26.     },
27.
28.     "--script"  : {
29.         "matrix-sve" : "gem5/rcS/matrix-sve.rcS",
30.         "gauss-sve" : "gem5/rcS/gauss-sve.rcS"
31.     }
32. }
```

Figura 3.9. *Contenido del archivo benchmark/cores_1_vl_128.json para la simulación.*

Al retomar el punto de simulación de los *checkpoints* con Linux ya arrancado, se han ejecutado los programas vectoriales correspondientes a los benchmarks definidos por los scripts *.rcS* especificados; *matrix-sve* y *gauss-sve*, cuyo código se muestra en la *Figura 3.10*. En la primera parte de ambos scripts se ha establecido la longitud de vector de SVE al máximo valor posible, es decir, 256 Bytes o 2048 bits. La segunda parte ejecuta los programas de SVE compilados dentro de la imagen de disco correspondientes a cada benchmark. El número de hilos que reciben estos benchmarks como parámetro se ha definido por el número de procesadores configurado en cada simulación, mientras que el tamaño de las matrices se ha especificado en el script. En concreto, para el benchmark *gauss* se ha utilizado una matriz cuadrada de 516 elementos de longitud. Este valor menos cuatro es múltiplo de todos los posibles números de hilos o cores en las simulaciones; 1, 2, 4, 8 y 16, como se explicó en el desarrollo del código. Por su parte, para el benchmark *matrix* se ha definido la longitud de la matriz al múltiplo 320.

```
1.  #!/bin/bash
2.
3.  echo "Configuring SVE VL"
4.  sysctl -w abi.sve_default_vector_length=256
5.  sysctl -p
6.  sysctl -n abi.sve_default_vector_length
7.
8.  cd
    /benchmarks/custom_benchmarks/threads/gauss
9.  ./gauss_sve `nproc` 516
10. m5 exit
```

```
1.  #!/bin/bash
2.
3.  echo "Configuring SVE VL"
4.  sysctl -w abi.sve_default_vector_length=256
5.  sysctl -p
6.  sysctl -n abi.sve_default_vector_length
7.
8.  cd
    /benchmarks/custom_benchmarks/threads/matrix
9.  ./matrix_sve `nproc` 320
10. m5 exit
```

Figura 3.10*. De izquierda a derecha, contenido de los scripts gauss-sve.rcS y matrix-sve.rcS.*

Tras la correcta definición de todos los archivos JSON y los *.rcS*, se han lanzado las 25 simulaciones mediante el script *launcher.py* del mismo modo que para la generación de los *checkpoints*. Al finalizar las simulaciones, se han comprobado los directorios de salida de cada experimento. Para cada uno de ellos, se ha verificado la ausencia de errores en la salida de error *.err*. Además, se ha comprobado la presencia de la llamada *"m5_exit"* en la salida estándar *.out* y de la línea *"Execution completed correctly"* en el archivo *system.terminal,* que indican la correcta terminación de la simulación y del programa del benchmark, respectivamente. De este modo, se ha confirmado el éxito en todas las simulaciones detalladas de ambos benchmarks.

A continuación, se ha utilizado el script *bulk_plots.py* del directorio *unican-util* del entorno gem5 de ATC_UC que permite generar múltiples gráficos de las estadísticas de los experimentos de las simulaciones de manera sencilla. Para ejecutarlo, se han especificado como parámetros el directorio de las simulaciones de gem5 con los experimentos desde los que se leen los datos, el directorio de salida donde se almacenan las figuras y el fichero de configuración *SVE_analysis.json*. Las figuras resultantes se han descargado en la máquina local para una correcta visualización y análisis del comportamiento de los benchmarks en cada configuración de simulaciones, y se presentan y analizan en el siguiente capítulo.

3.4. Recapitulación de la metodología

A modo de conclusión, se proporciona un resumen de los pasos de la metodología, ordenados en forma de algoritmo, para facilitar una visión global del procedimiento seguido.

Inicio

Paso 1: Configuración inicial del entorno de trabajo
- -Instalación de gem5 y sus dependencias en el clúster Triton
- -Establecimiento de rutas y variables de entorno necesarias

Paso 2: Realización de simulaciones
- -Desarrollo del código fuente de *matrix* y *gauss*
- -Modificación de la imagen de disco mediante la compilación de los benchmarks de SVE
- -Definición de los scripts *.rcS* y archivos JSON para las simulaciones
- -Generación de un *checkpoint* para cada configuración a simular
- -Ejecución de simulaciones
- -Comprobación de ausencia de errores

Paso 3: Análisis de los resultados
- -Generación de gráficas
- -Evaluación de resultados

Fin

Capítulo 4. Evaluación de resultados

En este capítulo, se realiza la evaluación de los resultados obtenidos tras las simulaciones. Para ello, primero, se recoge un sumario de la configuración utilizada para llevar a cabo los experimentos y, posteriormente, se realiza el análisis y la extracción de conclusiones, con ayuda de las gráficas generadas anteriormente, del comportamiento de los benchmarks.

4.1. Metodología

Los experimentos de este trabajo han sido realizados en la versión 21.2.0.0. del simulador gem5 alojada en Triton utilizando los parámetros de configuración recogidos en la *Tabla 4.1*.

El clúster Triton es un sistema de cómputo compuesto por cinco nodos de cálculo y un nodo *frontend*. Cada nodo, o servidor, tiene 32 GB de memoria y 2 procesadores Intel Xeon Silveer 4114. Estos procesadores funcionan a 2’20 GHz y cuentan con 10 cores físicos capaces de ejecutar 2 hilos cada uno. El nodo frontal tiene una unidad SSD de 150GB para el sistema operativo CentOS y 3 discos de 8TB configurados en RAID 5 para los archivos de usuario. Los nodos de cómputo arrancan el sistema del *frontend* a través de la red, por lo que no necesitan disco de sistema, pero cuentan una unidad SSD de 150 GB dedicada al área de intercambio y al almacenamiento de archivos temporales de los procesos de usuario. El arranque de los nodos de cómputo por red permite realizar cambios en su configuración de manera rápida y sencilla, adaptándose así a cambios en los requisitos de los investigadores.

La gestión de tareas de usuario en Triton se realiza con el gestor de colas Slurm, que permite lanzar las simulaciones en paralelo usando todos los cores disponibles.

Parámetro	Valor
ISA	ARMv8
Binario de gem5	gem5.opt
Modo de simulación	Full-System
Fichero de configuración	fs.py
Tipo de CPU	ex5_big
Núm. de cores	1, 2, 4, 8 y 16
Núm. de bancos de caché L3	8
Núm. de controladores de memoria	8
Frecuencia de reloj de CPU	2 GHz
Sistema de memoria	Ruby con caché
Red	Garnet
Topología	Pt2Pt
Tamaño de memoria	2 GB
Gestor de arranque	boot_v2.arm64
Kernel	vmlinux_5.15.10
Imagen de disco	ubuntu-21_04.img (versión con benchmarks)
Tipo de máquina	VExpress_GEM5_V2
Tamaño de vector de SVE	128, 256, 512, 1024 y 2048 bits
Benchmarks	matrix y gauss
Otros	Por defecto (a fecha de mayo 2022)

***Tabla 4.1**. Parámetros de configuración utilizados en los experimentos.*

Los tamaños de las matrices especificados en cada benchmark han sido, como se ha comentado previamente, para *gauss* 516 x 516 elementos y para *matrix*, 320 x 320. Adicionalmente, se han realizado experimentos con tamaños de matrices más grandes, en concreto, de 1032 x 1032 elementos en *gauss* y 640 x 640 en *matrix*. Sin embargo, los resultados obtenidos al modificar los tamaños han sido muy similares. Por este motivo, en la memoria se presentan únicamente los resultados para los tamaños iniciales.

En la sección siguiente se incluyen las gráficas generadas tras las simulaciones de ambos benchmarks sobre las que se analizan y comparan los resultados. Estas gráficas bidimensionales representan en el eje X la longitud en bits de vector de SVE utilizada (desde 128 hasta 2048). En ellas, aparecen 5 líneas de colores; una por cada número de cores utilizados, como indica la leyenda. Los valores representados en el eje Y dependen de la métrica mostrada en cada caso. Para comparar los resultados con facilidad, se muestran las dos gráficas de cada métrica juntas; a la izquierda la de *gauss* y a la derecha la de *matrix*.

Al analizar el código ensamblador de ambos benchmarks, se ha comprobado que el compilador no ha conseguido realizar una correcta vectorización de la función que realiza la multiplicación de las matrices en *matrix*. Para intentar resolverlo, se han realizado varias implementaciones diferentes, variando tanto la definición de los tipos de dato de las matrices como la estructura de los hilos, pero en ningún caso se ha conseguido la vectorización de dicha función. A pesar de que sí se han logrado vectorizar otras partes del código, estas generan un menor número de instrucciones dinámicas y tienen menos peso en el tiempo de ejecución del programa .Por ello, los resultados de *matrix*, a diferencia de los de *gauss*, muestran un caso en el que el compilador no es capaz de vectorizar el benchmark.

4.2. Resultados experimentales

Las gráficas de la *Figura 4.1* y *Figura 4.2* muestran el *speedup* obtenido en valores del eje Y comprendidos entre 1 y 8. Para el cálculo de este *speedup* se toma como base el tiempo de ejecución del procesador con un solo core y con el vector más pequeño (128 bits).

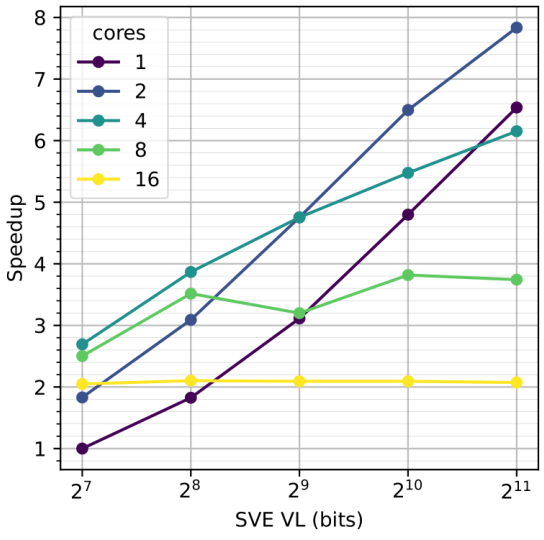


Figura 4.1. *Speedup de gauss.*

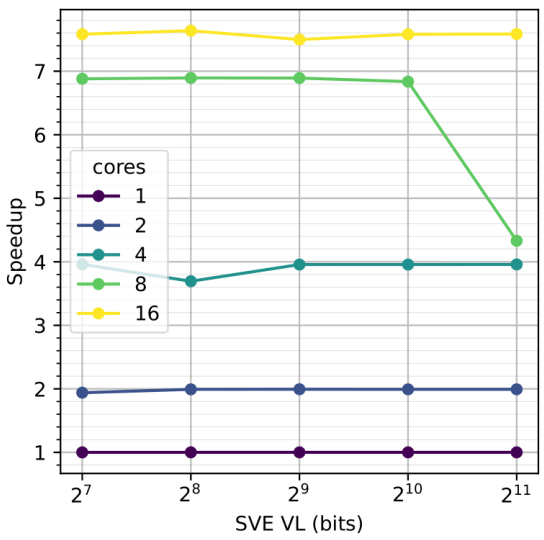


Figura 4.2*. Speedup de matrix.*

Los valores para *gauss* demuestran mejoras de rendimiento para 1, 2 y 4 cores a medida que aumenta la longitud de vector utilizada. Con 8 cores, para longitudes a partir de 256 bits, además de apenas mejorar, destaca que, en algunos casos, se llegan a obtener peores resultados al duplicar el tamaño de vector, como se muestra para vectores de 512 bits. Para 16 cores, el *speedup* obtenido es estable independientemente de la longitud de vector utilizada, consiguiendo peores resultados que para 4 y 8 cores y para la mayoría de casos con 1 y 2 cores. A la vista de la gráfica, también destaca que la ejecución con 1 core llega a proporcionar mejor resultado para 2048 bits que con 4 cores. Una posible explicación sería que el tamaño de la matriz, 516 x 516 elementos, es demasiado pequeño como para que la cantidad de cálculos a realizar compense el esfuerzo de gestión para más de 2 hilos. Este resultado también se puede deber a que al aumentar el número de hilos se produzca contención en el acceso a memoria que sature el ancho de banda de acceso a memoria y limite la mejora que se puede obtener.

Los valores para *matrix*, sin embargo, no son nada parecidos a los de *gauss*. Mientras que en el primer benchmark, por lo general, aumenta el *speedup* al aumentar la longitud de vector, en este lo hace al aumentar el número de cores, es decir, *matrix* no aprovecha la vectorización, debido a la no vectorización del código comentada anteriormente. En concreto, al duplicar los cores, se duplica el valor obtenido, menos para 8 y 16 cores, donde, aunque aumenta, deja de mantenerse la proporción x2. Sin embargo, destacan 2 casos extraños en los que para un mismo número de cores empeora notablemente el rendimiento; para 4 cores con 256 bits y, aún más impactante, 8 cores con 2048 bits, empeorando hasta casi 3 puntos respecto del valor alcanzado para el resto de longitudes.

En las gráficas de la *Figura 4.3* y la *Figura 4.4* se muestra el número de instrucciones ejecutadas en cada benchmark. Es importante notar que el eje Y de *gauss* está en escala 10^6, mientras que el de *matrix*, en escala 10^8. Esta diferencia puede ser debida al uso de dos matrices en el segundo benchmark frente al uso de a una matriz y un filtro pequeño en el primero.

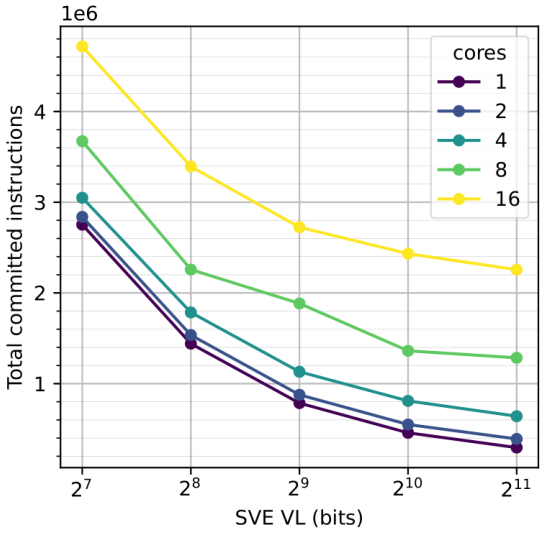


Figura 4.3. *Instrucciones ejecutadas de gauss.*

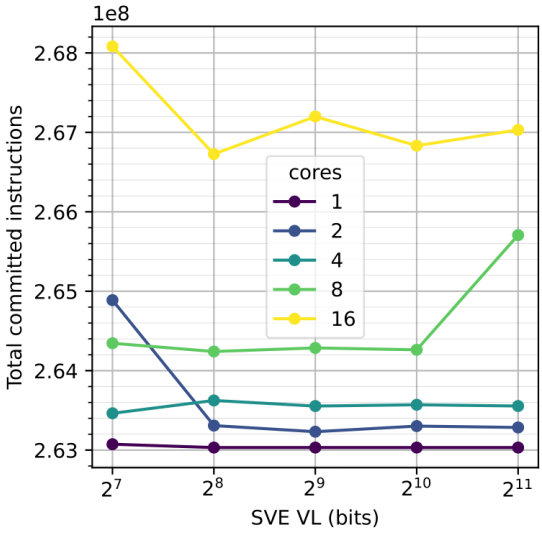


Figura 4.4*. Instrucciones ejecutadas de matrix.*

En ambos benchmarks se observa, para mismas longitudes de vector, un aumento del número de instrucciones relacionado con el aumento del número de cores.

En el benchmark de *gauss* se muestra una tendencia clara de disminución del número de instrucciones a medida que aumenta la longitud de vector. Este es un resultado esperable puesto que un mayor tamaño de vector implica registros más grandes, por lo que en una misma instrucción vectorial se manejan más datos, es decir, se necesitan menos instrucciones para abarcar todos los elementos del problema. Esto supone que los bucles reducen su número de iteraciones, por lo que, además de necesitarse menos instrucciones vectoriales, también se necesitan menos instrucciones escalares para el control de los bucles.

Sin embargo, en *matrix*, a diferencia de lo comentado en *gauss*, el comportamiento vuelve a ser independiente de la longitud de vector, ya que, por lo general, la tendencia en la segunda gráfica es estable para cada línea o número de cores, aunque vuelve a haber casos excepcionales, como la línea de 16 cores, o como los llamativos casos para 2 cores con 128 bits y, de nuevo, 8 cores con 2048 bits. En este último caso, a pesar de tratarse de una excepción, al incrementar el número de instrucciones, se comprende la disminución significativa del *speedup* visto anteriormente.

A continuación, se muestra el número de accesos por instrucción de memoria a la caché de primer nivel de datos en la *Figura 4.5*, con una escala entre 0 y 225, y en la *Figura 4.6*, con la escala entre 1’053 y 1’068.

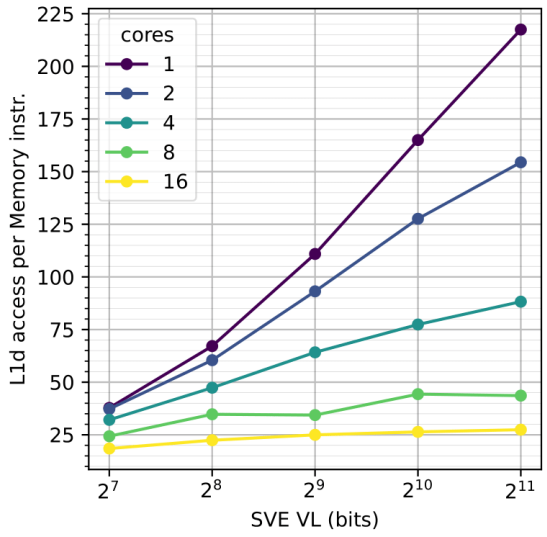


Figura 4.5. *Accesos a la caché de Datos L1 por instrucción de memoria de gauss.*

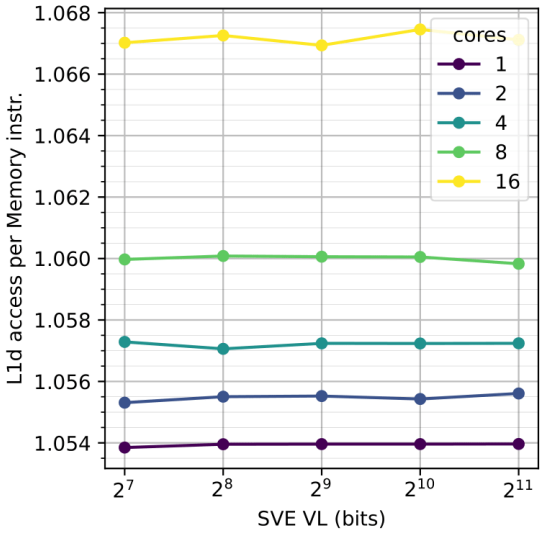


Figura 4.6*. Accesos a la caché de Datos L1 por instrucción de memoria de matrix.*

En la primera gráfica se aprecia que longitudes de vector más largas implican un aumento en el número de accesos a la primera caché de datos por instrucción de memoria. Este comportamiento tiene sentido puesto que las mismas instrucciones de memoria operan sobre registros más grandes, es decir, aumenta la cantidad de datos que maneja cada una. Por lo tanto, disminuye el número de instrucciones de memoria, como se observó en la métrica anterior, pero cada una realiza más accesos. Además, la proporción de accesos disminuye a medida que aumenta el número de cores, lo cual también es predecible, puesto que, al estar los datos más repartidos, a cada core le llegan menos

elementos sobre los que operar, y, por tanto, las instrucciones de memoria en cada uno deben realizar menos accesos.

Por el contrario, en la segunda gráfica, las instrucciones de memoria realizan prácticamente el mismo número de accesos en todos los casos. Además, este número de accesos es muy cercano a 1, ya que se trabaja realmente con instrucciones escalares en la parte del código que realiza la operación de multiplicación de matrices, que es la que más instrucciones ejecuta. En concreto, para cada número de cores, el valor se mantiene estable, sin que influya el tamaño de vector. Lo que sí influye es el número de cores, ya que al aumentar este, aumenta ligeramente el promedio de accesos por instrucción de memoria. Sin embargo, destaca la escala del eje Y, ya que prácticamente cada instrucción de memoria genera un acceso a la caché de datos L1. El valor decimal, que aumenta con el número de cores, podría deberse a una falta de alineación de los vectores dentro del bloque de caché que provoque un segundo acceso en determinadas operaciones.

En la *Figura 4.7* y la *Figura 4.8* se muestran las gráficas que recogen el número de veces que las ALUs SIMD de cada core estuvieron ocupadas en cada uno de los benchmarks. En esta ocasión, ambas gráficas son idénticas, puesto que en ningún caso se llegaron a bloquear las ALUs, es decir, nunca provocaron un cuello de botella para el sistema. Esto puede deberse a que el número de operaciones simultáneas que ambos benchmarks requieren no sobrepasa el límite de la ALU y, por tanto, aún se podrían realizar ejecuciones especificando matrices de más elementos.

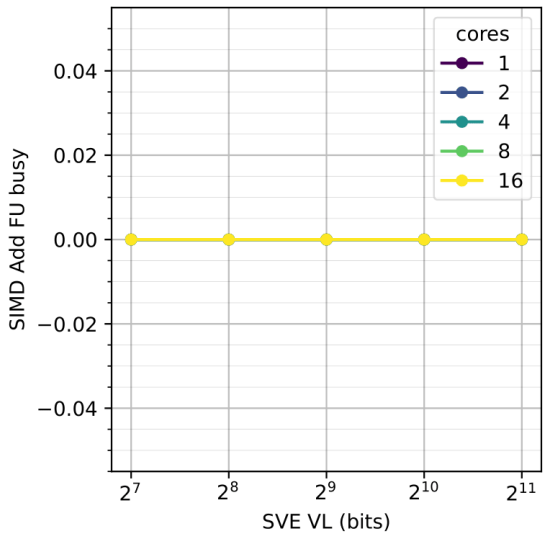


Figura 4.7. *Número de veces que las ALUs SIMD de cada core estaban ocupadas de gauss.*

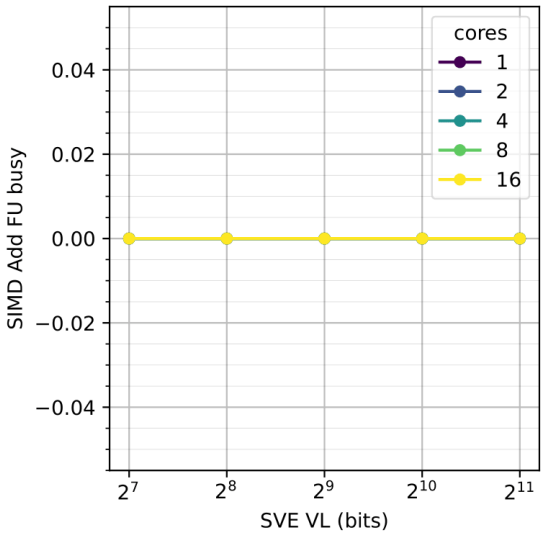


Figura 4.8*. Número de veces que las ALUs SIMD de cada core estaban ocupadas de matrix.*

La métrica recogida en la *Figura 4.9* y la *Figura 4.10* muestra el número de veces que la cola de carga y almacenamiento, o LSQ según sus siglas en inglés, estuvo totalmente ocupada en cada caso. En el benchmark *gauss*, la escala del eje Y abarca el rango entre 0 y 140 000 mientras que en *matrix*, lo hace entre 0 y 17 500 000.

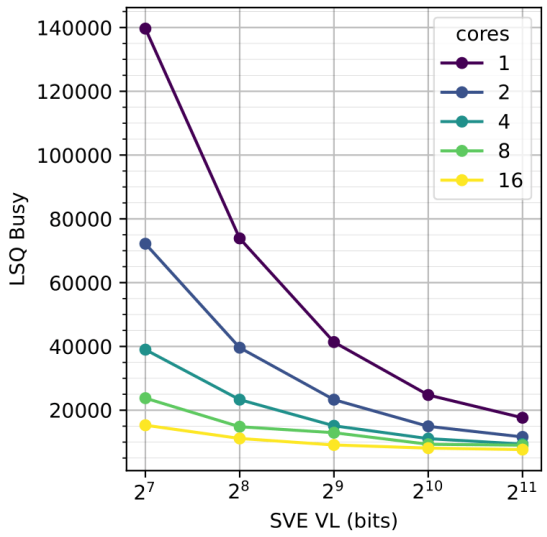


Figura 4.9. *Número de veces que la cola de carga y almacenamiento estaba ocupada de gauss.*

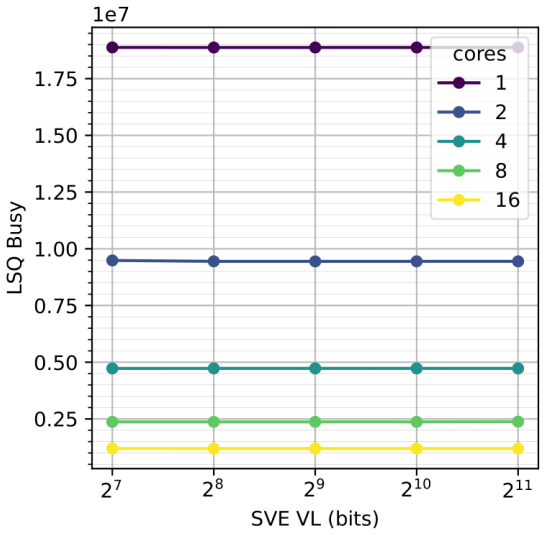


Figura 4.10*. Número de veces que la cola de carga y almacenamiento estaba ocupada de matrix.*

Por un lado, en la primera gráfica, se observa que la cantidad disminuye progresivamente a medida que aumenta la longitud de vector. Este fenómeno podría deberse a la disminución del número de operaciones de memoria debida al incremento del tamaño de los registros, es decir, se realizan menos peticiones de memoria, aunque cada una conlleva más carga. En la segunda gráfica, por el contrario, se vuelve a observar que el recuento es estable para cada número de cores, manteniendo un valor constante independientemente del tamaño de los vectores. Esto sugiere, una vez más, que el benchmark *matrix* no aprovecha la vectorización, puesto que realiza las mismas peticiones a memoria que de tratarse de un código escalar, debido a que no vectoriza la función principal que realiza la multiplicación de matrices. Atendiendo a la escala del eje Y se aprecia, además, que la cola de acceso a memoria en el caso de *matrix* está muy saturada en comparación con el de *gauss*, es decir, parece indicar que hay una gran cantidad de fallos de caché y se ha de acceder a memoria principal continuamente, lo que explica el comportamiento del *speedup* visto en el primer par de gráficas.

En cuanto al número de cores, en ambas gráficas coincide que, a medida que se duplica, el número de veces que la cola está ocupada disminuye en prácticamente la mitad. De este resultado se intuye que cada core posee su propia cola, por lo que el número de veces que cada una está ocupada disminuye al repartir las peticiones. Esto es razonable, puesto que, en efecto, cada core tiene su propia cola, además de su propia memoria caché L1 privada.

En la *Figura 4.11* y *Figura 4.12* se reúne el número medio de peticiones pendientes. Esta métrica, también se puede entender como el promedio de registros *MSHR*, del inglés *Miss Status Holding Register*, en uso por cada secuenciador o interfaz entre el core y la caché L1. En esta ocasión, la escala del eje Y de *gauss* contempla el rango desde 1 hasta 2’8 mientras que la de *matrix*, entre 2’2925 y 2’3105.

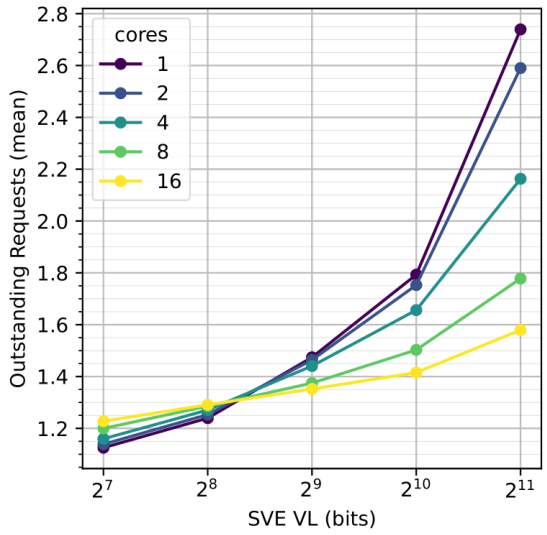


Figura 4.11. *Promedio de solicitudes pendientes de gauss.*

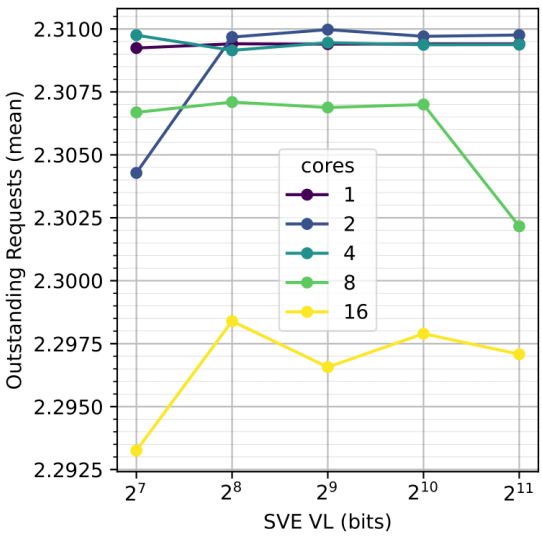
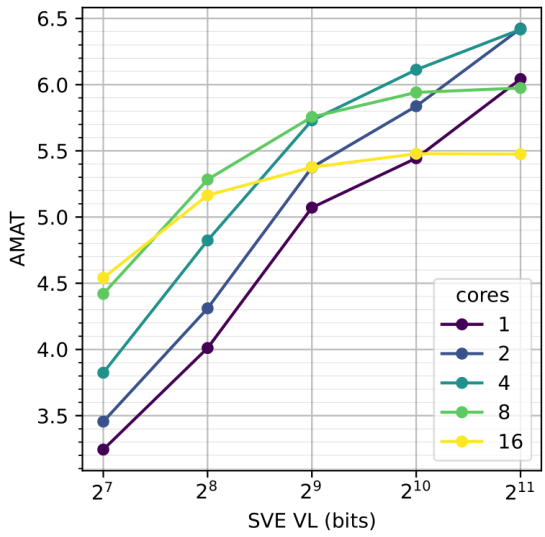


Figura 4.12*. Promedio de solicitudes pendientes de matrix.*

En *gauss*, para mayores longitudes de vector se observan mayores cantidades de peticiones pendientes. Sin embargo, a la vista de la escala, no se puede considerar que se crea ningún cuello de botella significativo, ya que, en promedio, ni siquiera se llegan a alcanzar 3 peticiones pendientes simultáneas. En *matrix*, aunque la longitud de vector, generalmente, no afecta el valor, el resultado es similar; de media, tampoco se llegan a quedar pendientes simultáneamente tres peticiones. De nuevo, en este benchmark, vuelven a darse las excepciones para 2 cores con 128 bits y 8 cores con 2048 bits en los que el valor disminuye más respecto de la tendencia para cada línea de cores, pero, dada la escala, sin ser relevante. Además, también se aprecia una inestabilidad en la línea de los 16 cores cuya excepción parece estar relacionada con la correspondiente a la gráfica del número de instrucciones ejecutadas analizada anteriormente.

Por último, las métricas siguientes muestran el rendimiento del subsistema de memoria. Primero, en las gráficas de la *Figura 4.13* y *Figura 4.14*, se especifica el tiempo medio de acceso a memoria, o AMAT por sus siglas en inglés. En *gauss*, la escala del eje Y abarca el rango [3, 6’5] y en *matrix*, más reducido, el rango [4’143, 4’173].



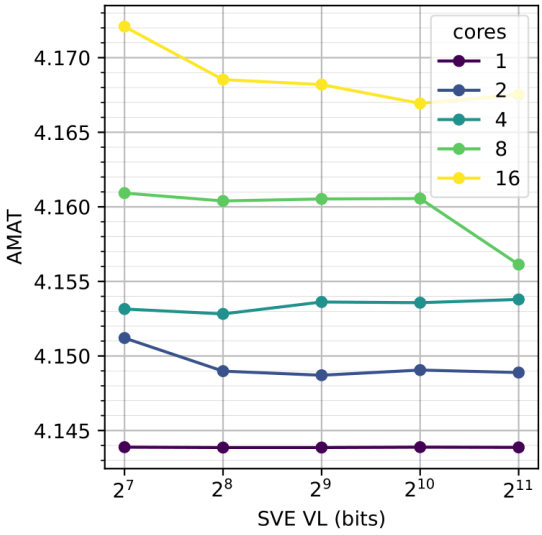


Figura 4.13. *Tiempo medio de acceso a memoria de gauss.*

Figura 4.14*. Tiempo medio de acceso a memoria de matrix.*

De la primera gráfica, donde el tiempo crece con la longitud de vector, se deduce lo comentado anteriormente; en *gauss*, al aumentar el tamaño de los registros, aunque se realizan menos peticiones a memoria, estas tienen más carga, es decir, operan sobre más datos, y, por tanto, conllevan más tiempo. Por lo tanto, se puede concluir que el ancho de banda de acceso a memoria debería aumentarse con el tamaño del vector, ya que en caso contrario supone un fuerte impacto en el tiempo de acceso a memoria, como muestran las gráficas. En este caso, el número de cores influye en un mayor tiempo de acceso hasta alcanzar 8 y 16 cores, donde sorprendentemente empieza a disminuir para tamaños a partir de 512 y 256 bits, respectivamente.

En la segunda gráfica, de nuevo, se entiende que en *matrix* las instrucciones vectoriales no tienen efecto en el rendimiento, es decir, al comportarse como un código escalar, el tiempo de acceso es solamente proporcional al número de datos. En este caso, el número de cores estrictamente influye de manera negativa en el tiempo medio de acceso a memoria, pues en ningún caso se obtienen mejores tiempos con mayores cores, ni siquiera con las excepciones habituales. No obstante, la diferencia de tiempo incrementada por el aumento de cores no es significativa dada la escala.

La métrica de las gráficas en la *Figura 4.15* y la *Figura 4.16* muestra la latencia media de los fallos, o promedio de ciclos de penalización por fallo. En el eje Y, la escala de *gauss* abarca desde 34 hasta 52’5 ciclos mientras que la de *matrix*, desde 31’72 hasta 32’24 ciclos.

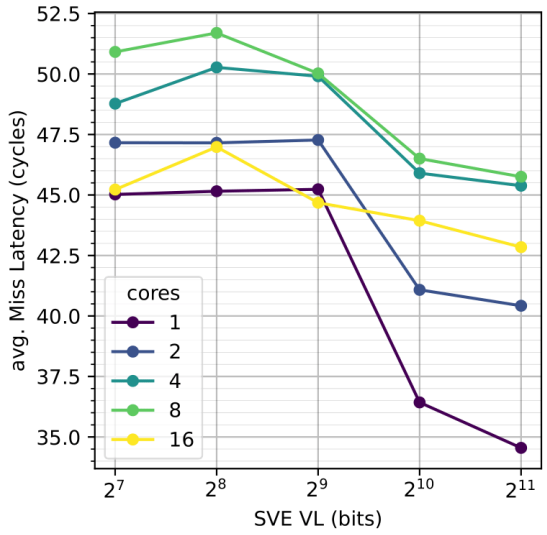


Figura 4.15. *Latencia media de los fallos, o penalización por fallo, de gauss.*

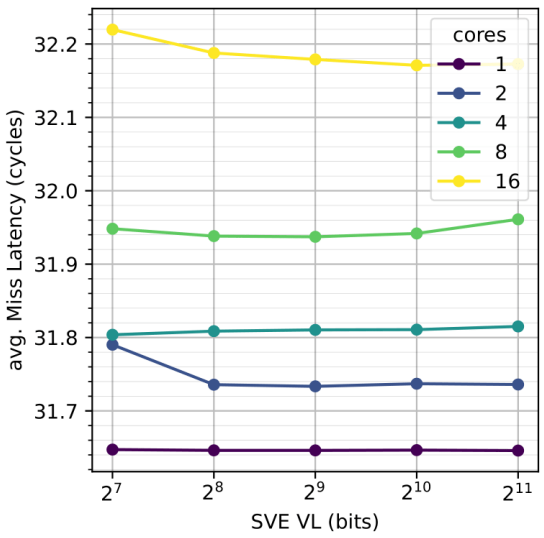


Figura 4.16*. Latencia media de los fallos, o penalización por fallo, de matrix.*

Los resultados en el primer benchmark son extraños. Para 1 y 2 cores, hasta 512 bits, el número medio de ciclos de penalización por fallo se mantiene casi constante, pero para las longitudes de vector más largas disminuye considerablemente, alcanzando hasta 10 ciclos menos. Por otro lado, para 4, 8 y 16 cores, inicialmente aumenta al pasar de 128 a 256 bits, mientras que para el resto de tamaños disminuye, esta vez menos acentuadamente. Este comportamiento resulta bastante extraño, ya que para las primeras longitudes de vector el código se comporta de una forma y, tras un punto de inflexión, pasa a comportarse de otra. Sin embargo, es evidente que las mayores longitudes de SVE perjudican menos por los fallos, puesto que consiguen disminuir notablemente la latencia. En este caso, el número de cores juega un papel contrario al tamaño de vector; a medida que aumenta, empeora el resultado, exceptuando el caso de 16 cores que, insólitamente, para 512 bits, incluso consigue mejorar la latencia de 1 core.

En contraste, el segundo benchmark demuestra resultados ya esperados; la latencia promedio de los fallos es independiente de la longitud de vector utilizada, sin considerar los casos de excepción ya habituales. Nuevamente, el rango de la escala es muy pequeño y, aunque el aumento de cores incrementa el número medio de ciclos de penalización por fallo, quizá debido a que se genera algo de contención en el acceso a memoria, la diferencia no es para nada significativa.

En el *Anexo I* se recogen otros pares de gráficas con métricas de evaluación adicionales.

4.3. Discusión de los resultados

A pesar de que ambos benchmarks presentan un porcentaje de vectorización similar, no logran vectorizar de la misma manera. Mientras que en *gauss* el compilador consigue realizar con éxito la vectorización de la función que más peso tiene en la ejecución del programa, es decir, la que realiza el filtrado, en *matrix*, por el contrario, no lo consigue con la función equivalente; la que realiza la multiplicación de matrices. Esta diferencia resulta muy evidente al comparar las gráficas de los resultados. El comportamiento de *matrix* es prácticamente el de un código escalar, puesto que los resultados obtenidos son casi siempre independientes de la longitud de vector de SVE

utilizada. Sin embargo, en *gauss*, se observa el comportamiento propio de un código vectorizado en el que el aumento del tamaño de vector influye considerablemente en el rendimiento.

Por otro lado, en ambos benchmarks sí influye la variación del número de cores, lo que era de esperar dado el código, ya que en *matrix* se divide la matriz A en hilos, que en la simulación corresponden a los cores, de la misma forma en que se divide la matriz de píxeles en *gauss*.

El análisis de diferentes métricas ha confirmado que el uso de mayores longitudes de vector de SVE, en *gauss*, mejora notablemente el rendimiento. En concreto, para el tamaño máximo, 2048 bits, siempre se consigue el máximo *speedup* para cada número de cores, a pesar de que aumenten los tiempos de acceso a memoria al incrementarse el número de accesos por instrucción. Un resultado muy interesante es que el *speedup* conseguido con 2 cores usando vectores de 2048 bits es muy superior al del resto de configuraciones. Esto determina que, en el procesador simulado, la mejor configuración desde el punto de vista del rendimiento es la que utiliza solamente 2 cores, pero con el máximo tamaño del vector de SVE. Sin embargo, si, por ejemplo, solo se utilizaran vectores de 512 bits, la mejor configuración posible a utilizar sería con 4 cores.

En comparación, mientras que en el benchmark de *matrix*, que se comporta como un código escalar, el número mayor de cores, 16, proporciona el mayor rendimiento, en *gauss*, cuyo comportamiento sí es el de un código vectorizado, apenas merece la pena pasar de 1 o 2 cores, ya que no solo no mejora, sino que incluso puede llegar a empeorar, el resultado. Por tanto, se podría afirmar que, mientras se consiga una vectorización exitosa, para optimizar el código, resulta preferible, por ser más efectivo, ampliar la longitud vectorial de SVE, a poder ser al máximo, antes que el número de cores para distribuir la carga. Por otra parte, además, el número de cores utilizado también tendría un fuerte impacto en el consumo energético, aunque este estudio queda fuera del alcance de este trabajo. Pero sí se puede intuir que usando 2 cores el consumo energético será sensiblemente menor que con 16 cores, máxime si se reduce notablemente el tiempo de ejecución, como se ha demostrado al analizar el rendimiento.

Otro aspecto importante que destacar es la relación entre las prestaciones de la memoria y el tamaño del vector. De los resultados presentados se puede deducir que el incremento del tamaño del vector debe venir acompañado de un incremento en el ancho de banda de acceso a memoria del sistema. En caso contrario se producirán incrementos en los tiempos de acceso a memoria que pueden llegar a lastrar notablemente el rendimiento de la aplicación.

Capítulo 5. Conclusiones y trabajos futuros

En este último capítulo se recopila, a modo de conclusión, un resumen de los pasos realizados y los aspectos claves de las conclusiones alcanzadas en el desarrollo de este trabajo. Además, se realizan diferentes propuestas sobre posibles líneas de investigación futuras cuyas aportaciones estén relacionadas con el trabajo reflejado en esta memoria.

5.1. Conclusiones

En este Trabajo de Fin de Grado se ha definido una metodología detallada en tres fases para la evaluación de diferentes configuraciones de la arquitectura SVE de ARM en el simulador gem5 con las herramientas desarrolladas por el Grupo ATC de la Universidad de Cantabria.

En la primera fase, se han especificado los pasos necesarios para la configuración del entorno de simulación del gem5 alojado en el clúster Triton, cuyo acceso ha sido realizado en remoto.

En la segunda fase, se ha simulado el comportamiento de dos benchmarks en diferentes configuraciones del sistema. Estas configuraciones han sido definidas mediante la combinación de cinco implementaciones de la arquitectura de SVE, en concreto, para las longitudes de vector de 128, 256, 512, 1024 y 2048 bits, con cinco definiciones del número de cores entre los que distribuir la carga de trabajo; 1, 2, 4, 8 y 16. Para ello, primero, se han desarrollado los benchmarks; *gauss*, que aplica un filtro gaussiano bidimensional a una matriz de píxeles, y *matrix*, que realiza la multiplicación de dos matrices. Ambos utilizan matrices cuadradas y reparten el trabajo en hilos distribuidos en los diferentes cores. Para realizar evaluaciones de la arquitectura del sistema de forma realista, es necesario utilizar el modo de simulación *Full-System*, que modela el sistema completo con sistema operativo. Por ello, ha sido necesario incluir los benchmarks compilados en la imagen de disco a utilizar, lo que requiere de un montaje cuyos privilegios en Triton solo poseen supervisores. Esto ha condicionado la realización de este paso en el equipo local, mediante la instalación de un terminal Ubuntu con WSL en el que se ha utilizado QEMU para manipular la imagen de disco en la que se han compilado los binarios de los benchmarks con instrucciones de SVE. La versión de la imagen de disco resultante ha sido subida a Triton y utilizada en los archivos JSON de cada simulación. A continuación, ha sido necesaria la generación de un *checkpoint*, o captura del sistema simulado para cada configuración especificada, desde el que posteriormente se ha retomado cada simulación. Las simulaciones de *gauss* han sido especificadas para matrices cuadradas de 516 elementos, y las de *matrix*, para 320. Por último, se ha comprobado la correcta ejecución de todas las simulaciones.

En la tercera fase se han generado distintas gráficas con los resultados obtenidos tras las simulaciones y se ha comparado el comportamiento de cada benchmark evaluando el rendimiento demostrado tras el escalado de la longitud de vector y el número de cores.

En los resultados de *matrix*, cuya función principal no ha conseguido ser vectorizada con éxito, se refleja el comportamiento propio de un código escalar; el rendimiento mejora al aumentar el número de cores y la longitud de vector de SVE apenas tiene efecto. Por el contrario, de los resultados obtenidos de gauss, cuya vectorización sí se ha logrado correctamente, se deduce que, por lo general, se consigue mejor rendimiento a medida que se incrementa la longitud de vector de SVE, preferiblemente al máximo, y se disminuye el número de cores. Sin embargo, dicho incremento parece tener cierto impacto negativo en los tiempos de acceso a memoria. Esto es, probablemente, debido al

desequilibrio con el ancho de banda de memoria, que ha permanecido inalterable durante las simulaciones. A pesar de esto, destaca que el mayor *speedup* se ha alcanzado con la configuración para la máxima longitud de vector de SVE, 2048 bits, y solo 2 cores.

5.2. Trabajos futuros

Para finalizar, en esta sección se proponen posibles líneas de trabajo adicional a seguir en un futuro para la continuación de lo desarrollado en este Trabajo de Fin de Grado.

Primeramente, podría ser interesante realizar una investigación adicional sobre la razón por la cual ciertos benchmarks no consiguen ser vectorizados con éxito según la arquitectura de SVE por el compilador GNU. Esto podría arrojar luz sobre el caso concreto del benchmark *matrix* que, a pesar de haber sido tratado de la misma manera que el de *gauss*, no ha conseguido la misma vectorización, pues, aunque sí se ha logrado vectorizar parte del código, no se ha hecho en la función principal que realizan los hilos, es decir, la de la multiplicación de matrices. Para esta propuesta, es necesario realizar una búsqueda minuciosa de las diferencias entre ambos benchmarks, dado que ni el tipo de datos ni la estructura de los hilos ha resultado ser la causa. Además, esta búsqueda se podría extender mediante el desarrollo de nuevos tipos de benchmarks, probablemente ayudando a encontrar un patrón común en los códigos que consiguen o no vectorizar con éxito la función principal de sus hilos.

Como segunda línea de trabajo, se propone la inclusión de la evaluación del consumo energético de los benchmarks en las simulaciones de gem5. Para ello, podría utilizarse la ayuda de la herramienta McPAT, un marco integrado para el modelado de potencia, área y tiempo para procesadores fuera de orden. A pesar de que McPAT originalmente no es compatible con las instrucciones SVE, Odajima *et al.* [17] contribuyeron a su extensión mediante una modificación que permite calcular el consumo de energía a partir del número de instrucciones de SVE ejecutadas. Esto permitiría evaluaciones más exhaustivas de los experimentos realizados, atendiendo tanto al rendimiento como a la eficiencia energética.

Otra línea de trabajo relacionada puede ser una investigación que permita analizar qué posibles cambios en las prestaciones de memoria se ajustan mejor al tamaño de los vectores. Para ello, podría realizarse, por ejemplo, un estudio que compruebe si la ampliación del ancho de banda para vectores más grandes produce o no mejores resultados, observando las diferencias de comportamiento en las gráficas de tiempo medio de acceso a memoria (AMAT).

Finalmente, se propone una línea de trabajo adicional en la que se realice un estudio más detallado de los accesos a memoria. Esto permitiría la observación de los distintos tipos de fallos que se producen en cada caso, lo que sería útil para intentar mejorar sus parámetros mediante la realización de diferentes modificaciones en las implementaciones de los benchmarks.

Bibliografía

[1] «Arm Developer. Introduction to SVE,» [En línea]. Disponible: https://developer.arm.com/documentation/102476/0001.

[2] «Gem5,» [En línea]. Disponible: https://www.gem5.org.

[3] «Arm. RISC,» [En línea]. Disponible: https://www.arm.com/glossary/risc.

[4] M. Flynn, «Flynn's Taxonomy,» *Encyclopedia of Parallel Computing.*, p. 689–697, 2011.

[5] M. Stanic, «An Integrated Vector-Scalar Design on an In-Order ARM Core,» *ACM Trans. Archit. Code Optim.*, Mayo 2017.

[6] «Arm Developer. Learn about de Scalable Vector Extension (SVE),» [En línea]. Disponible: https://developer.arm.com/documentation/101726/0400/Learn-about-the-Scalable-Vector-Extension--SVE-/What-is-the-Scalable-Vector-Extension-.

[7] N. Stephens, «The ARM Salable Vector Extension,» *IEEE Micro*, Mar-Abr. 2017.

[8] E. Castillo, M. Moretó, M. Casas, L. Álvarez, E. Vallejo, K. Chronaki, R. M. Badia, J. L. Bosque, R. Beivide, E. Ayguadé, J. Labarta y M. Valero, «CATA: Criticality Aware Task Acceleration for Multicore Processors,» *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 413-422, 2016.

[9] E. Castillo, M. Moretó, M. Casas, L. Álvarez, E. Vallejo, J. L. Bosque, R. Beivide y M. Valero, «Architectural Support for Task Dependence Management with Flexible Software Scheduling,» *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 283-295, 2018.

[10] N. Binkert, «The gem5 simulator,» *SIGARCH Comput. Archit. News*, nº 39.2, pp. 1-7, 2011.

[11] A. Tousi y C. Zhu, «Arm Research Starter Kit: System Modeling using gem5.,» *Arm*, Julio 2017.

[12] J. Lowe-Power et al., «The gem5 Simulator: Version 20.0+.,» *ArXiv*, 2020.

[13] A. Armejach et al., «Using Arm's Scalable Vector Extension on Stencil Codes,» *The Journal of Supercomputing*, vol. 76, nº 3, pp. 2039-2062, 2020.

[14] D. Zhong et al., «Using Arm Scalable Vector Extension to Optimize OPEN MPI,» de *The 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid20)*, 2020.

[15] S. Scholbe, «Extending DaCe to support the Scalable Vector Extension,» Zurich, 2021.

[16] N. Kitai et al., «An Auto-tuning with Adaptation of A64 Scalable Vector Extension for SPIRAL,» *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 789-797, 2021.

[17] T. Odajima, Y. Kodama y M. Sato, «Performance and power consumption analysis of Arm Scalable Vector Extension,» *The Journal of Supercomputing*, vol. 77, p. 5757–5778, Junio 2021.

[18] «Microsoft Docs. Windows Subsystem for Linux Documentation,» 2021. [En línea]. Disponible: https://docs.microsoft.com/en-us/windows/wsl/.

[19] «QEMU,» [En línea]. Disponible: https://www.qemu.org/.

[20] I. Pérez Gallardo, «Instructions to use the ATC-UC environment,» 11 Marzo 2022.

[21] «GCC, the GNU Compiler Collection. GCC Optimize Options,» [En línea]. Disponible: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[22] Arm Architecture Reference Manual Supplement, The Scalable Vector Extension, 24 de Mayo, 2021.

[23] A. Rico, «Vector Architecture for HPC and ML. BSC Severo Ochoa Research Seminar,» 2018.

[24] A. Rico, «ARM HPC Ecosystem and the Reemergence of Vectors,» *Association for Computing Machinery*, Mayo 2017.

[25] N. Stephens, «ARMv8-A Next-Generation Vector Architecture for HPC,» *IEEE Hot Chips*, pp. 1-31, 22 Agosto 2016.

[26] N. Stephens, «The Scalable Vector Extension for Armv8-A,» *Arm Community*, 22 Agosto 2016.

[27] A. Barredo Ferreira, «Novel techniques to improve the performance and the energy of vector architectures,» 2021.

[28] K. Asanov, «Vector Processor's in More Depth,» de *Computer Architecture: A Quantitative Approach*, 2012.

[29] M. J. Flynn, «Very High-speed Computing Systems,» *Proceedings of the IEEE* , vol. 54, Diciembre 1966.

[30] B. Bramas, «A fast vectorized sorting implementation based on the ARM scalable vector extension (SVE),» *PeerJ Computer Science*, 19 Noviembre 2021.

Anexo I. Métricas adicionales de evaluación de los benchmarks

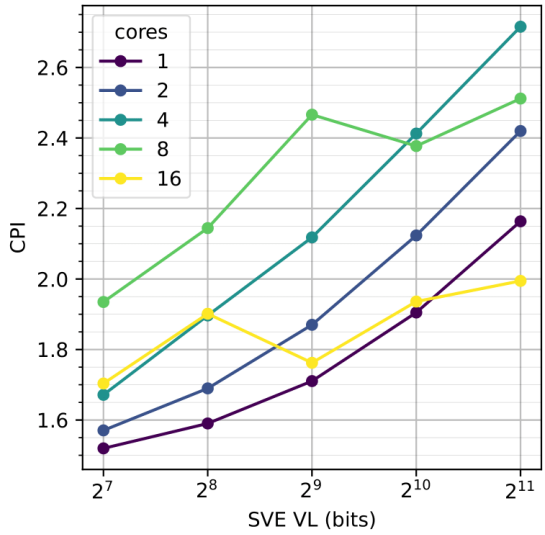


Figura I.1. *CPI medio de gauss.*

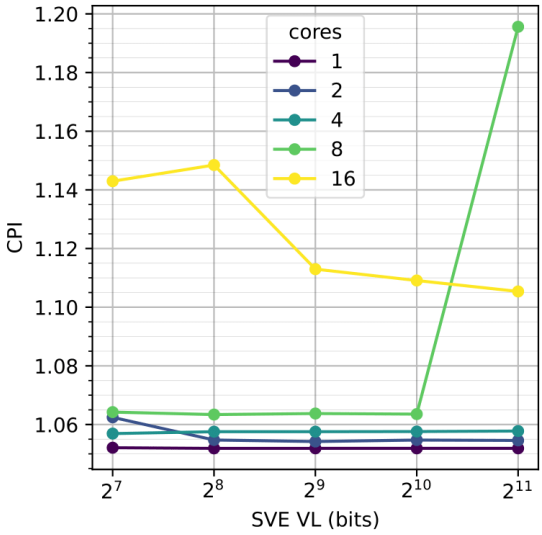


Figura I.2*. CPI medio de matrix.*

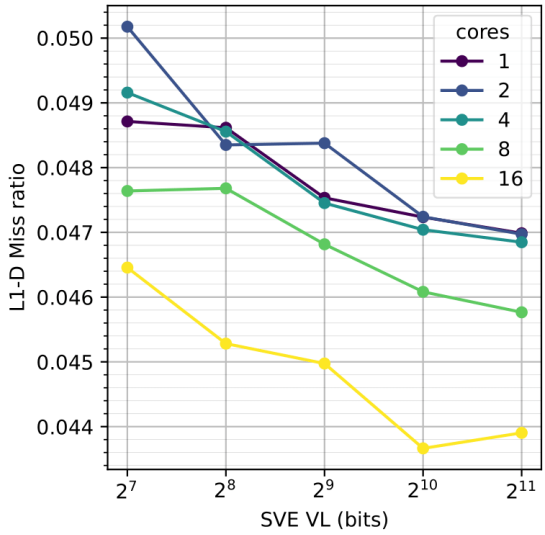


Figura I.3. *Tasa total de fallos de la caché L1-D de gauss.*

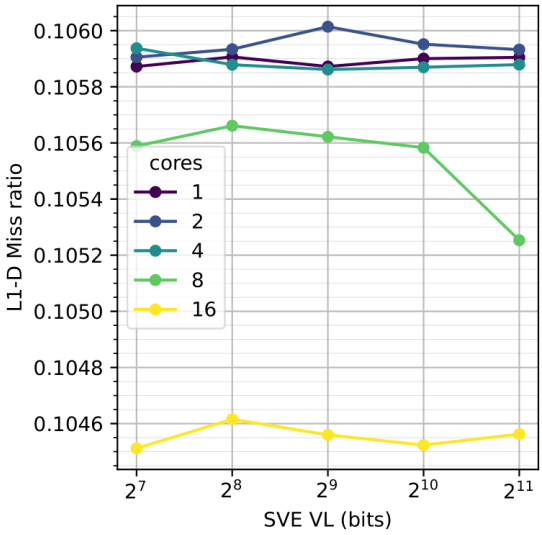


Figura I.4 *Tasa total de fallos de la caché L1-D de matrix.*

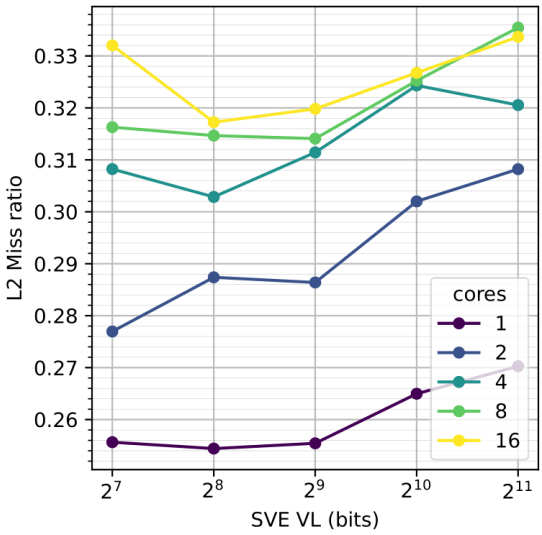


Figura I.5. *Tasa total de fallos de la caché L2 de gauss.*

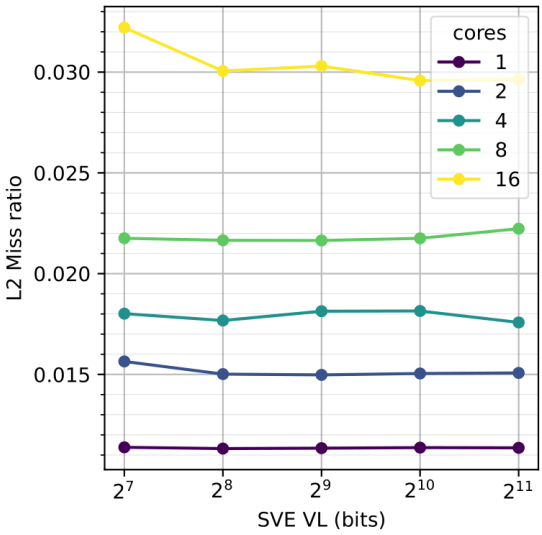


Figura I.6*. Tasa total de fallos de la caché L2 de matrix.*

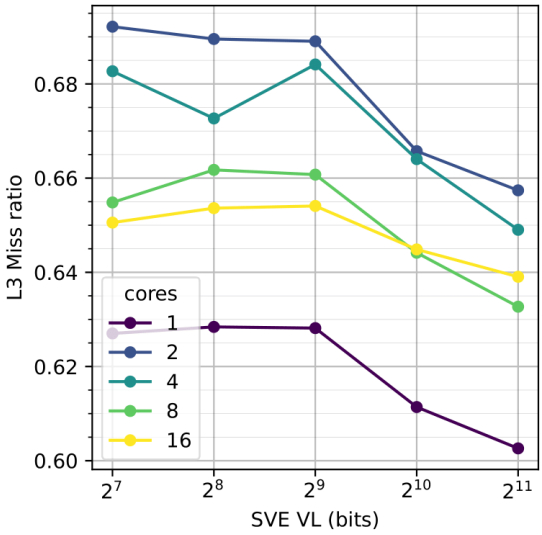


Figura I.7. *Tasa total de fallos de la caché L3 de gauss.*

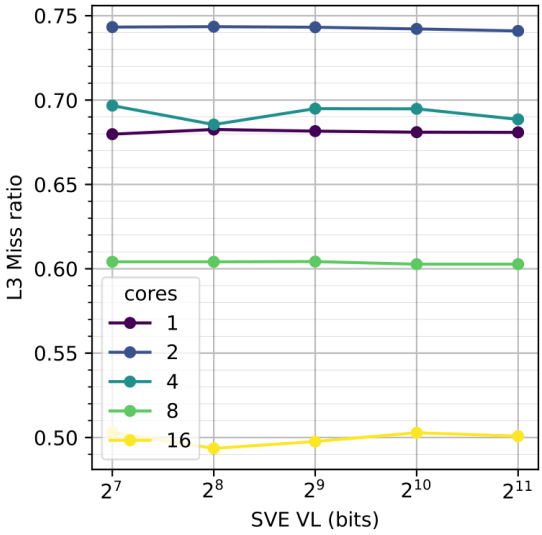


Figura I.8*. Tasa total de fallos de la caché L3 de matrix.*