Adaptación de un framework de IA en procesadores vectoriales para el espacio

Aleix Valdivieso González

Resumen—En este trabajo se ha evaluado el uso de instrucciones vectoriales con intrínsecos y auto-vectorización en los compresores V2F y JPEG-TURBO. La evaluación se ha realizado en el hardware Kendryte MV-K230 con RVV 1.0. Además, se ha analizado la eficacia de la extensión vectorial (RVV) en un entorno más controlado, implementando las funciones REDUCE, SAXPY, VVADD, MATMUL y la CONVOLUCIÓN2D. Para estas funciones, se ha encontrado el LMUL óptimo y se ha determinado la mejor implementación de VL para tamaños de array no múltiplos del registro vectorial. Todo esto, con el objetivo final de compararlo con la ejecución escalar para CHAR, INT, FLOAT y DOUBLE.

Palabras clave— RISC-V, RVV, V2F, compresor, JPEG-Turbo, auto-vectorización, GCC-14, Clang-17, Xuantie-C908.

Abstract— This paper evaluates the use of vector instructions through intrinsics and auto-vectorization in the V2F and JPEG-TURBO. The evaluation was performed by the Kendryte MV-K230 hardware with RVV 1.0. Additionally, the efficiency of the vector extension (RVV) was analyzed in a more controlled environment by implementing the REDUCE, SAXPY, VVADD, MATMUL and CONVOLUCIÓN2D. For these functions, the optimal LMUL was determined, and the best implementation of VL for array sizes that are not multiples of the vector register was identified. All of this was conducted with the goal of comparing it to scalar execution for CHAR, INT, FLOAT, and DOUBLE.

Index Terms— RISC-V, RVV, V2F, compressor, JPEG-Turbo, auto-vectorization, GCC-14, Clang-17, Xuantie-C908.

1 Introducción

RISC-V es la arquitectura del momento [1], su ideología open-source y su gran flexibilidad utilizando extensiones le permite instaurarse en gran cantidad de aplicaciones de todo tipo [2]. Una aplicación es el espacio, por ello instituciones como el IEEC (Institut d'Estudis Espacials de Catalunya) y más concretamente el equipo de Vibria, tiene el objetivo de estudiar y mejorar la ejecución de código en esta arquitectura mediante el uso instrucciones vectoriales y de esta manera ahorrar el uso de una GPU (Unidad de Procesamiento Gráfico). Este trabajo es una continuación de "*Implementació d'algorismes vectorials per acceleració HW en sistemas RISC-V per espai.*" [3] y tiene los principales objetivos de evaluar los rendimientos, adaptación y finalmente implementación de las operaciones SAXPY, REDUCE, VVADD, MATMUL y CONVOLUCIÓN2D pasando de la versión de RVV (RISC-V Vectorial extension) en su versión preliminar 0.7.1 a la primera versión estable 1.0. Y, finalmente, evaluar el uso de RVV 1.0 en dos casos de uso, concretamente el algoritmo de compresión de datos V2F (Variable-to-Fixed) desarrollado por el DEIC (Department of Information and Communications Engineering) perteneciente a la Universitat Autònoma de Barcelona y el compresor JPEG. Sus evaluaciones consistirán en pruebas de rendimiento entre la ejecución optimizada vectorialmente y la ejecución escalar en el mismo entorno RISC-V. Como objetivos secundarios o relacionados con los principales, este trabajo aspira a proporcionar un entendimiento mayor en la programación en el lenguaje C en bajo nivel mediante el uso de intrínsecos, la familiarización con una tecnología emergente como es RISC-V y su extensión vectorial, así como del funcionamiento de compresores de datos con distintas filosofías, como el V2F y el JPEG.

- *E-mail de contacte: aleix.valdivieso@autonoma.cat*
- *Menció realitzada: Tecnologies de la Informació*
- *Treball tutoritzat per: Màrius Montón Macian*
- *Curs 2023/24*

2 Marco Teórico

Este apartado permite al lector comprender conceptos relevantes para un mejor entendimiento de este trabajo. Entre estos conceptos, se remarca explicar que es RISC-V, centrando el foco en su extensión vectorial, así como el tipo hardware utilizado y una breve introducción a las funciones de muestra a evaluar.

2.1 Estado del Arte

Las instrucciones vectoriales son una herramienta fundamental para mejorar el rendimiento en aplicaciones de procesamiento paralelo sin necesidad de utilizar componentes externos al procesador. Para utilizar las instrucciones vectoriales, existen dos métodos: los **intrínsecos** y la **auto-vectorización**. Los intrínsecos son funciones de bajo nivel, las cuales adaptan el código máquina para poder ser utilizado en lenguaje C. La auto-vectorización es el proceso que realiza el compilador a un binario para aprovechar situaciones concretas en las que poder aprovechar el uso de instrucciones vectoriales. RISC-V está en continuo desarrollo, ello implica que trabajos recientes quedan rápidamente desfasados. La versión **0.7.1** de la extensión vectorial fue la primera en ser lanzada al mercado como una versión

preliminar y solo era compatible con el uso de intrínsecos, además, de ser soportada únicamente por una versión experimental de **Clang** desarrollado por el BSC, por otro lado, la versión **1.0** es la primera versión estandarizada, proporcionando una mayor estabilidad para versiones futuras, permitiendo retrocompatibilidad entre ellas [4]. Los compiladores al uso como **Clang** o **GCC**, permitieron soporte de esta extensión en la versión **16** y la **13** (respectivamente). Todo este soporte fue solo dado a los intrínsecos, mientras que la auto-vectorización (tema clave en este estudio), está siendo desarrollado y estará disponible en las versiones **Clang 17** y **GCC 14**.

2.2 RISC-V

RISC-V es una arquitectura de código abierto, diseñada por la Universidad de Berkeley, California. Esta arquitectura utiliza instrucciones de tamaño fijo de 32 bits y contiene 32 registros (los cuales soportan variables de 32 y 64 bits). Siendo una arquitectura de tipo RISC, su ISA es simple y mínima contando con las operaciones: LOAD/STORE, operaciones atómicas, instrucciones de sistema, instrucciones de salto y operaciones aritmético/lógicas básicas. Gracias a su diseño flexible, esta ISA puede ser aumentada mediante el uso de extensiones y de esta manera, permitir que el procesador se adecue a aplicaciones concretas. Algunas de las estandarizadas son la extensión de operaciones como la multiplicación y división (Extensión M), operaciones con punto flotante simples (Extensión F) y dobles (Extensión D) y la más relevante para este estudio, la extensión vectorial (Extensión V). [5]

2.2.1 Vectorització a RISC-V (RVV)

La ISA de esta extensión es relativamente grande en comparación a la ISA básica de RISC-V debido a la necesidad de implementar todas las instrucciones escalares con diferentes configuraciones, así como tener instrucciones específicas para acceder a memoria y manipular los elementos de los registros vectoriales.

Esta extensión vectorial cuenta con gran flexibilidad permitiendo un uso similar a las instrucciones vectoriales en arquitecturas ya existentes como SIMD. No por ello, RVV incorpora novedades como el uso de tamaños de registros fijos o agnósticos. Los vectores de tamaño fijo (**VLS**) permiten que la instrucción vectorial trabaje con un tamaño fijo de registro vectorial (**Vlmax**) y por tanto operará las posiciones correspondientes a este. Por otro lado, el vector de tamaño agnóstico (**VLA**) permite que este varie dinámicamente y que de esta manera se pueda operar con las posiciones únicamente necesarias.

A la hora de utilizar o implementar la extensión vectorial, se deben tener en cuenta una serie de aspectos, uno concierne al diseñador del chip, mientras que el otro, al programador de más alto nivel.

Aspectos no configurables en tiempo de ejecución:

El diseñador debe tener en cuenta dos parámetros para implementar la extensión en el procesador: Primero se deben implementar 32 registros vectoriales de un tamaño fijo **VLEN** (Vector Register Size o tamaño en bits de cada registro). Estos registros estan divididos en elementos (**ELEN**) y con la restricción de ser potencia de dos y 8≤ELEN≤VLEN.

Aspectos configurables en tiempo de ejecución:

Hay ciertos parámetros que se definen en tiempo de ejecución o compilación, estos pueden venir dados ya bien por la auto vectorización realizada por el compilador como por el propio desarrollador mediante el uso de intrínsecos. Estos parámetros son el **vtype** y el **vl**.

El **vtype** permite describir el tipo de operador con el cual se operará: el **SEW** es el tamaño de la variable a operar (float = 32 bits) y tiene como restricción 8≤SEW≤ELEN. Por otro lado, el **LMUL**, es el multiplicador de registros vectoriales. Este parámetro es de especial relevancia para conseguir la mejor optimización posible ya que permite aprovechar la cantidad de registros vectoriales implementados. Sus posibles valores son:

- = 1: Operar con todos los elementos del registro vectorial.
- < 1: Operar con una fracción de los elementos del registro vectorial.
- > 1: Incrementar el tamaño de registros mediante la agrupación de registros vectoriales, esto reduce el número total de registros vectoriales disponibles.

Por otro lado, tal como muestra Figura 2-1, el **vl** describe cuantos elementos del vector se utilizarán para operar. Su valor máximo es **Vlmax** y viene dado por $LMUL * VLEN/SEW$.

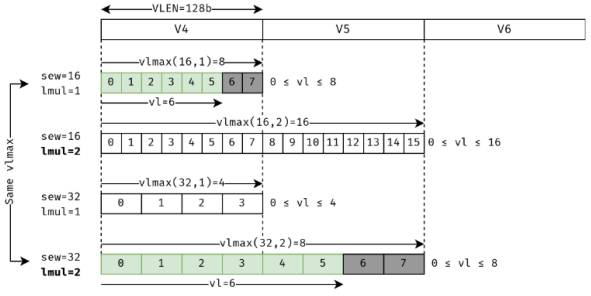

Figura 2-1 – Ejemplos de uso con distintos parámetros VL, LMUL y SEW.

Instrucciones vectoriales enfocadas a los accesos a memoria y máscaras

La extensión vectorial cuenta con instrucciones para acceder a memoria, y aunque lo ideal es realizar accesos a memoria contiguos para aprovechar al máximo posible las operaciones vectoriales, esta extensión también soporta:

- **Accesos a memoria separados por una misma distancia arbitraria**, esta opción es interesante para situaciones en las que recorrer matrices por columnas.
- **Accesos a memoria utilizando un vector de índices (gathering)**, esta opción resulta útil cuando se quiere realizar accesos a memoria basado en el offset proporcionado por el contenido de otro vector.

Todas las instrucciones vectoriales soportan el uso de **máscaras**, las cuales permite controlar el flujo de datos con instrucciones vectoriales. Las máscaras son vectores en los que sus elementos son bits simples.

Por ejemplo, si se quiere vectorizar un bucle con instrucciones condicionales, se puede realizar su conversión vectorial utilizando máscaras y luego aplicando la máscara a la instrucción deseada, permitiendo operar solo las posiciones del registro vectorial que haya cumplido la condición. En el caso de la Figura 2-2, se utiliza v0 como registro de máscaras y se emplea en la instrucción vvadd para controlar que operaciones se realizaran y cuáles no.

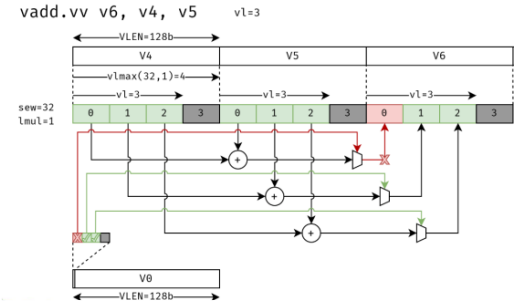

Figura 2-2 – Instrucción vvadd con máscaras.

2.3 Kendryte MV-K230

La Canaan Kendryte MV-K230 es una placa basada en el SoC Kendryte-230 contando con 512 MB de RAM y su característica más importante: con un procesador T-Head Xuantie C908. Este procesador es dual-core, siendo el principal a 1.6 GHz con soporte 1.0 para la extensión vectorial y otro secundario más básico de 800 MHz con soporte de extensiones G, C y B. A diferencia de otras placas en el mercado con arquitectura RISC-V, esta permite ejecutar el sistema operativo Linux, en este caso la versión sid de Debian adaptada por el propio IEEC.

2.4 Funciones a evaluar

En este apartado, se explicarán las funciones de muestra a evaluar y por qué son métricas relevantes para su uso en entornos vectoriales.

2.4.1 SAXPY / DAXPY

$$y[i] = x[i] * a + y[i]$$

La función **SAXPY/DAXPY** combina operaciones multiplicativas sobre datos escalares y multiplicativas/aditivas sobre arrays. A pesar de ser un código simple, corto y con poca carga computacional, su capacidad de paralelización puede aprovechar eficientemente las características del uso de instrucciones vectoriales.

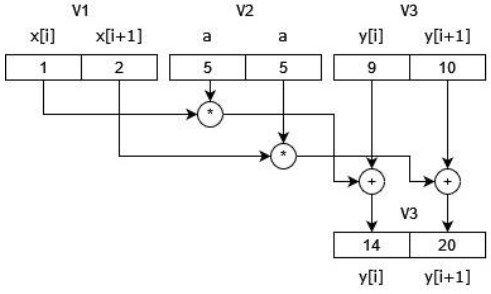

Figura 2-3 – Diagrama de la operación SAXPY.

2.4.2 VVADD

$$z[i] = x[i] + y[i]$$

La operación **VVADD** es la operación más sencilla de todas las propuestas, ya que consiste en la suma de dos posiciones de memoria de dos arrays guardando su resultado en otro array. Esta operación presenta gran capacidad de vectorización y permitirá comprobar cuellos de botella en los accesos a memoria y encontrar el límite de la MV-K230.

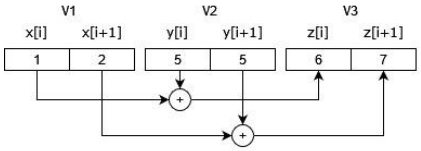

Figura 2-4 – Diagrama de la operación VVADD.

2.4.3 Reduce

$$s += x[i] * y[i]$$

La función **REDUCE** es ampliamente utilizada en computación y tratamiento de grandes datos. Consiste en realizar una operación entre dos arrays, guardando la suma de todas estas en una única variable.

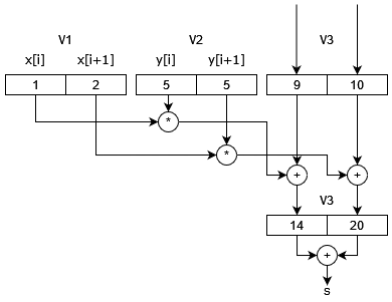

Figura 2-5 – Diagrama de la operación Reduce. V3 es el registro resultante de las operaciones acumulativas.

2.4.4 Multiplicación de Matrices

$$A[n][o] * B[n][o] = C[n][n]$$

La función **MATMUL** realiza la multiplicación de dos matrices obteniendo una tercera como resultante. Esta es una de las operaciones más exigentes para la optimización vectorial y mostrará como actúa bajo condiciones no idóneas. Las optimizaciones que se le aplicarán a parte de la vectorial, será la transposición de la segunda matriz, permitiendo de esta manera realizar los accesos a memoria de forma contigua.

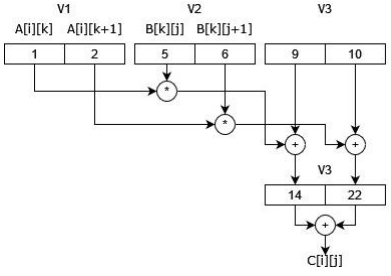

Figura 2-6 – Diagrama de la operación MATMUL. Operación en la última iteración, donde V3 tiene los valores previos y desde donde se sumarán para guardarlos en C[i][j].

2.4.5 Convolución 2D

$$y[i] += x[j] * h[i-j]$$

La **convolución 2D** es ampliamente utilizada en redes neuronales y compresores de imagen, por ello, estudiar el efecto que las instrucciones vectoriales producen en ella, será de gran relevancia.

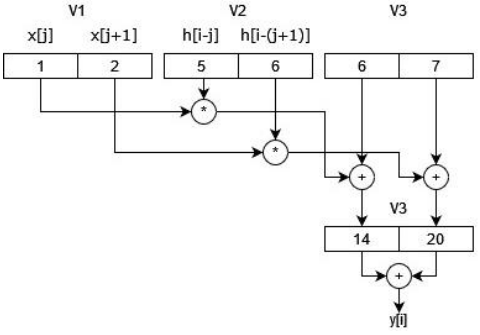

Figura 2-7 – Diagrama de la última iteración en operación Convolución 2D.

2.5 Variable to Fixed (V2F)

El Variable to Fixed es un sistema de compresión desarrollado por el GICI (Group on Interactive Coding of Images) de la Universitat Autònoma de Barcelona (UAB). Su objetivo es permitir la compresión y descompresión de datos sin perdidas en entornos espaciales. Este compresor está basado en la codificación Tunstall, el cual mapea los símbolos de entrada en un número fijo de bits. Para ello se construye un árbol de longitud variable en el que cada hoja tiene asociada una palabra codificada en función de los símbolos de entrada que se han iterado para llegar hasta ahí. De esta manera, y conociendo la palabra codificada se conoce la secuencia de símbolos de entrada. Este método proporciona una codificación y decodificación simple y computacionalmente liviana, siendo útil en aplicaciones de transmisión de datos en tiempo real como puede ser la comunicación satelital.

2.6 Herramientas de medida

Este estudio requiere de herramientas precisas que nos puedan proporcionar datos fiables, por ello es importante mostrar que herramientas se utilizarán.

- **gprof**: Herramienta de análisis de rendimiento que permita perfilar la ejecución de binarios.
- **clock_gettime():** Función perteneciente a la librería time.h de C. Permite medir el tiempo de las ejecuciones de funciones dentro del propio código. Se utilizará con el valor clk_id=CLOCK_MONOTONIC.
- **time**: Herramienta de medición de tiempo para binarios en ejecución.

3 Metodología y Planificación

Este trabajo está desarrollado sobre tres etapas principales. La primera etapa es teórica y se centrará en el desarrollo del estado del arte y el entorno sobre el que se trabajará. Las dos siguientes etapas son prácticas: La implementación y optimización de las operaciones de muestra y la optimización vectorial de los algoritmos de compresión.

La primera etapa servirá de fase preparatoria para obtener una mayor comprensión de **RVV 1.0**. Se estudiará el hardware sobre el que se implementaran tanto las operaciones como la optimización de **V2F** [6]. Finalmente se investigará sobre el estado del arte y las compatibilidades de los compiladores sobre esta extensión tanto en su vertiente de intrínsecos como auto-vectorización.

La segunda etapa consistirá en la programación de las operaciones básicas para su final evaluación.

Finalmente, la última etapa consistirá en la optimización del algoritmo **V2F** junto con el **JPEG-Turbo**. En el caso de **V2F** se deberá perfilar el código para obtener las funciones de mayor cómputo y así poder aplicarles la optimización vectorial para finalmente evaluar el tiempo de ejecución en un marco de pruebas controlado junto a la ejecución escalar. Mientras que con **JPEG-Turbo** únicamente se evaluará el tiempo de ejecución en base a la ejecución escalar.

Para realizar este trabajo se empleará una metodología agile la cual permita un flujo de trabajo flexible y rápido [7]. Como metodología agile se utilizará **KanBan**, esta consiste en la división del trabajo en tareas las cuales tendrán asociadas un estado TO DO (por hacer), DOING (en proceso), REVIEW (revisión), DONE (completado). Con este método se podrá realizar el trabajo de una forma más fluida junto con el equipo de Vibria, el cual utiliza el entorno de repositorios GitLab como tablero.

Para la planificación, se utilizará de apoyo la Figura 0-1 del apéndice. A medida que avance el trabajo siguiendo la metodología **KanBan**, se irá siguiendo el diagrama de Gantt para intentar garantizar una carga de trabajo bien repartida.

3.1 Primera fase: Estado del arte

Inicialmente, se deberá estudiar y comprender la nueva versión 1.0 de RVV para poder utilizar sus intrínsecos, esta fase no será extensa ya que se aprovecharán los conocimientos adquiridos en trabajos previos enfocados en la versión 0.7.1 de RVV [3]. Para realizar esta tarea, se deberá estudiar los cambios experimentados entre cada una de las versiones, se deberán comprender las características del hardware que se utilizara y finalmente analizar el soporte de herramientas de compilación y estado del arte en este entorno.

3.2 Segunda fase: Operaciones básicas

Esta fase consiste en la programación e implementación de las funciones MATMUL, REDUCE, SAXPY, VVADD, CONVOLUCIÓN2D. Para ello, se utilizará el lenguaje de programación C en bajo nivel, haciendo uso de los intrínsecos de RVV 1.0 y la auto-vectorización proporcionada por compiladores. Todas las operaciones serán implementadas para los tipos de dato: punto flotante de precisión doble (fp64), precisión simple (fp32), enteros de 32 bits (int32) y enteros de 8 bits (int8).

Cada operación sigue la misma estructura de trabajo, primero, se programarán las funciones aprovechando la estructura realizada por trabajos previos sobre las mismas

operaciones [3]. En esta estructura, se podrán instaurar e implementar las operaciones con todos los tipos de dato a evaluar, estas implementaciones tendrán tres características: una versión escalar, una versión auto-vectorizada, una versión vectorizada mediante intrínsecos y una versión auto-vectorizada mediante el uso de *#pragma*. A continuación, se evaluará la sintonización de la variable LMUL de RVV. Seguidamente se ejecutará y se comparará el tiempo de ejecución para cada tipo de dato entre las operaciones optimizadas vectorialmente y su versión escalar. Una vez el cómputo esté realizado, se procederá a evaluar los resultados obtenidos a raíz de los distintos compiladores utilizados para cada una de las optimizaciones.

3.3 Tercera fase: Optimización del compresor V2F y JPEG

Esta fase contendrá dos principales tareas, optimizar vectorialmente el códec de compresión V2F y optimizar vectorialmente el códec JPEG-turbo. El objetivo principal de este trabajo es optimizar vectorialmente el V2F, pero también se ha escogido el compresor JPEG ya que será interesante comprobar como las instrucciones vectoriales actúan con respecto a un códec de compresión sin pérdidas (V2F) y otro con pérdidas (JPEG).
Pasando a explicar el flujo de trabajo que tendrá V2F, primero de todo se deberá suplir y asegurar que las dependencias y el algoritmo funciona correctamente en una máquina Linux con arquitectura x86, esto permitirá realizar la prueba en un entorno bien conocido y familiarizado. Una vez conseguido, se realizará el mismo proceso en el hardware RISC-V. Asegurado el funcionamiento del algoritmo, se pasará a la siguiente etapa, optimizar el compresor. Para ello, este se deberá perfilar con el objetivo de buscar la función que más tiempo de ejecución consume en su cómputo y finalmente, optimizarla. En este punto, se deberá evaluar los resultados obtenidos con la ejecución del compresor de forma escalar, con la ejecución del V2F optimizado vectorialmente mediante el uso de intrínsecos y mediante el uso de auto-vectorización por parte del compilador.
El flujo de trabajo del compresor JPEG [8] será más sencillo debido al gran soporte que hay por parte de la comunidad. Una de estas versiones es JPEG-Turbo [9], el cual está preparado para utilizar instrucciones vectoriales. Primero de todo, se compilará el compresor de tres formas distintas: Compilación escalar, compilación auto-vectorizada y compilación con soporte nativo de RVV por parte de JPEG-Turbo. Una vez esta etapa finalizada, se evaluarán los tiempos de ejecución de cada una de las compilaciones.

3.4 Documentación

Finalmente, se redactará el informe final junto con una presentación la cual incluirá todo el trabajo y resultados obtenidos del desarrollo de este, finalizando con una conclusión de los resultados mostrados además de un posible trabajo futuro realizable en este entorno.

4 Desarrollo

En este capítulo se realizarán las implementaciones y evaluaciones de las operaciones de muestra, así como la optimización del compresor V2F y el JPEG-Turbo según la metodología previamente explicada.

4.1 Implementación de las funciones

En esta fase del desarrollo, se especificará el entorno de pruebas, se considerarán unos aspectos previos para la implementación de las operaciones de muestra y se evaluarán los tiempos de ejecución obtenidos para cada uno de los tipos de ejecución.

Consideraciones preliminares y entorno de prueba

Antes de comenzar a ejecutar los binarios y evaluar los resultados obtenidos, es necesario establecer un entorno de pruebas óptimo, para ello se utilizará la operación *Multiplicación de Matrices* con la que se configurará y conocerá: qué compilador es el que mejor resultado proporciona en base a la optimización vectorial, el LMUL óptimo de la placa de desarrollo kv230 y la mejor vía para afrontar un tamaño de array no múltiple de VL.

Elección del compilador

La ejecución de las operaciones de muestra y de los compresores se realizarán con un único compilador, por ello, es necesario comprobar cuál proporciona un mayor desempeño con respecto a la optimización vectorial. Este apartado constará de dos pruebas: Comparativa de tiempo y número de instrucciones máquina con la auto-vectorización y comparativa del tiempo de ejecución utilizando intrínsecos.
Para ejecutar los archivos C de la operación se ha creado un archivo Makefile en el que poder configurar la compilación, permitiendo realizar una compilación con y sin auto-vectorización y permitiendo escoger uno de los tres compiladores disponibles: BSC-Clang19git, GCC-14, Clang-16. Para finalmente generar el archivo ejecutable, el archivo objeto (.o) y el archivo ensamblador (.asm).

Comparativa de tiempo de ejecución en intrínsecos

Esta comparación no debería reflejar cambios excesivos en la ejecución, ya que estas funciones son simplemente "traducciones" a instrucciones máquina, pero es interesante ver el comportamiento de cada compilador de cara a estas optimizaciones, así como el tiempo de referencia que ofrecen. Hay "flags" que son comunes para los tres compiladores, estos son:

- **-O2:** Optimización de segundo nivel por parte del compilador.
- **-ffast-math:** Permite al compilador aplicar optimizaciones a variables de punto flotante.
- **-march=rv64gcv1p0:** Arquitectura con la que se ejecutará el binario compilado: riscv 64 bits con la extensión C, G y V [5]. Siendo esta última a versión 1.0.

BSC-clang19git:

- **-mepi**: Flag únicamente utilizada en el compilador del BSC para utilizar su propia toolchain de intrínsecos.

GCC-14:

- **-misa-spec=2.2:** Versión de la ISA que utiliza el procesador.

La Figura 4-1 muestra que el compilador que ofrece un menor tiempo de ejecución es GCC-14 tanto en tiempo de referencia como vectorial.

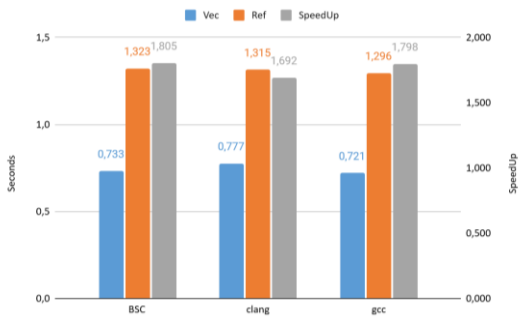

Figura 4-1 – Tiempos de ejecución intrínsecos de MATMUL en compiladores Clang-BSC, Clang 16, GCC-14

Comparativa de tiempo de ejecución y número de ASM auto-vectorización

Esta comparativa nos va a proporcionar resultados muy útiles para conocer que compilador es capaz de realizar la mejor auto-vectorización, así como reducir al máximo el número de instrucciones máquina dentro del bucle más interno y de esta manera reducir el número de ciclos en la ejecución. Debido a que la prueba de LMUL sobre la operación Matmul no se ha realizado todavía, se utilizará un valor de LMUL=2 ya que en anteriores trabajos se reflejó que este valor era el óptimo [3]. Para establecer estos nuevos flags se utiliza:

BSC-clang19git y clang-16:

- **-mllvm -riscv-v-register-bit-width-lmul=2**: Flag que establece LMUL=2

GCC-14:

- ***-mrvv-max-lmul=m2:*** Flag que establece el valor LMUL=2 para la auto-vectorización.

La Figura 4-2 muestra que la mejor optimización vectorial viene dada por el compilador del BSC, obteniendo un tiempo de ejecución de 0,713s con respecto a 0,75s de gcc-14.

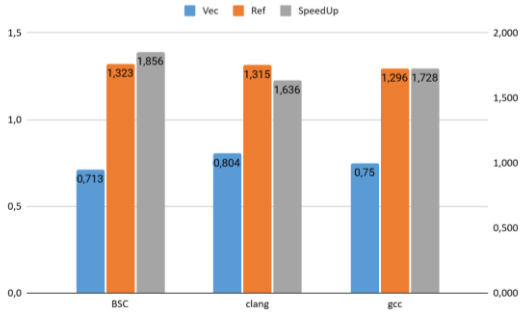

Figura 4-2 – Tiempos de ejecución auto-vectorial de MATMUL en compiladores Clang-BSC, Clang 16, GCC-14

La Figura 4-3 muestra que el número de instrucciones máquina en el bucle más interno es el mismo para todos los compiladores a excepción de GCC, esto se debe a que es capaz de extraer del bucle más interno la instrucción que configura el VL (cosa obvia si se tiene en cuenta que la cantidad de posiciones que se debe cargar y operar en cada iteración es el mismo).

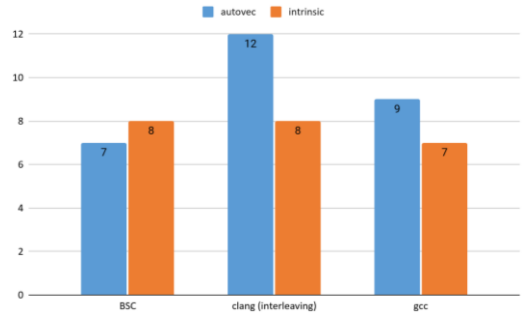

Figura 4-3 – Número de instrucciones máquina del bucle más interno en MATMUL en compiladores Clang-BSC, Clang 16, GCC-14.

Por otro lado, el compilador que a priori utiliza la menor cantidad de instrucciones utilizando a la auto-vectorización es el compilador del BSC, pero esto es debido a que clang-16 utiliza por defecto interleaving junto con la auto-vectorización y, por tanto, estaría realizando dos operaciones vectoriales en el mismo bucle. En el caso de GCC, es el que más instrucciones utiliza.

A pesar de que el compilador del BSC ofrece un mejor resultado, y teniendo en cuenta el enfoque de Vibria se basa en utilizar recursos accesibles para el mercado general, se utilizará el compilador GCC-14 para el resto del desarrollo de este proyecto.

Sintonización de LMUL

La sintonización de la variable LMUL es importante ya que gracias a ella podremos aprovechar al máximo la optimización que las instrucciones vectoriales proporcionan, para ello, se establecerá una configuración común para todas las pruebas donde lo único que varie son los valores disponibles para LMUL (1, 2, 4, 8). Este valor es puramente arbitrario y dependiente del diseño del procesador, por tanto, la única manera de sintonizarla es mediante la experimentación.

El tamaño de array utilizado es N=32 (matriz de 32x32), valor de reiteración es NLOOPS=10000. Con este tamaño, nos aseguramos de no sobrepasar la caché y el valor de reiteración tan alto permite evitar el cache-warmup.

Los resultados de la Figura 4-4 muestran que utilizar un LMUL=2 es el que mejor rendimiento proporciona con lo que respecta a utilizar optimizaciones vectoriales. Lógicamente las configuraciones que peores resultados proporcionan son 1 y 8. En el caso de LMUL=1 no se estan aprovechando los 32 registros vectoriales y con ello la ejecución se queda limitada a un tamaño de registro vectorial de 128 bits. Por otro lado, con un LMUL=8, se realizan agrupaciones de registros demasiado grandes (proporcionando un tamaño virtual de 1024 bits con un número total de 4 registros vectoriales accesibles), esto crea problemas debido a la poca cantidad de registros vectoriales disponibles y el

cuello de botella que se ocasiona debido al pequeño tamaño de las memorias caché, así como el lento acceso a los datos.

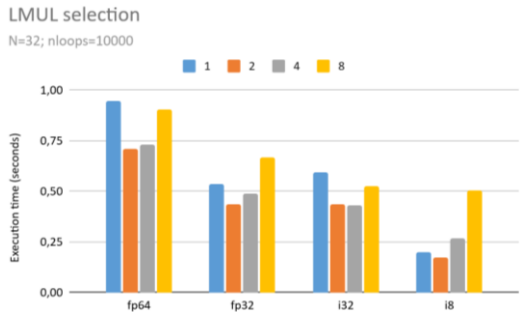

Figura 4-4 – Selección LMUL en MATMUL en compiladores Clang-BSC, Clang 16, GCC-14.

Array no múltiple de VL

La extensión vectorial dispone del parámetro VL para indicar las posiciones de memoria que se cargaran en el registro vectorial, si este no se modifica cuando el tamaño del array no es múltiplo de este número provocará problemas de overflow. Esta prueba se realizó con anterioridad sobre la versión 0.7.1 [3], presentando tres posibles soluciones: Utilizar la función Golden para suplir el número de iteraciones restantes, transformar el tamaño de los arrays a uno múltiplo de VL y utilizar la instrucción que configura el VL para poder establecerlo a necesidad.

A diferencia de los resultados obtenidos en la versión 0.7.1 de RVV en el que utilizar el zero-padding era más rápido [3], en esta versión RVV 1.0, la Figura 4-6 demuestra que configurar en tiempo de ejecución el valor VL aporta el mejor rendimiento, lo cual, encaja con la filosofía y diseño de la extensión vectorial [10].

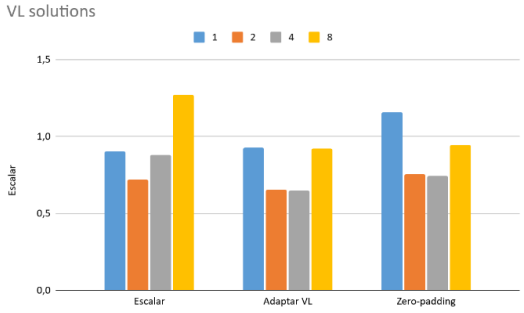

Figura 4-6 – Tiempos de ejecución escalar, adaptación vl y zero-padding en función de LMUL.

Evaluación de las funciones de muestra

En este apartado se mostrarán los resultados en forma de SpeedUp al aplicar la optimización vectorial sobre las operaciones de muestra, tanto utilizando intrínsecos, como utilizando auto-vectorización. Todas las ejecuciones se han realizado utilizando los siguientes *flags*: *-O2 -ffast-math -march=rv64gcv1p0 -misa-spec=2.2 -mrvv-max-lmul=m2*.

Reduce

La Figura 4-5 muestra una mejora en todas las situaciones tanto en la optimización mediante intrínsecos como la auto-vectorizada, con especial mención al uso de int8, el cual, al ser un tipo de dato pequeño, permite cargar mayor cantidad de valores y de esta manera aprovechar las instrucciones vectoriales. Otro aspecto reflejado en los resultados es la variación de SpeedUp obtenido entre el uso de intrínsecos y el uso auto-vectorial. La teoría sugiere que los intrínsecos deberían ofrecer mayor resultado ya que es el programador el que aplica la optimización de manera específica, esto es cierto para los tipos de datos de punto flotante, pero no para los enteros en los que se puede apreciar que utilizando auto-vectorización en (por ejemplo) int8 y N=1024 se obtiene un SpeedUp de 29,7x mientras que, con intrínsecos de 26,65x. Por tanto, demuestra que la auto-vectorización proporcionada por los compiladores es más que capaz de suplir con los requisitos de calidad que se podría tener a la hora de optimizar un código con estas características.

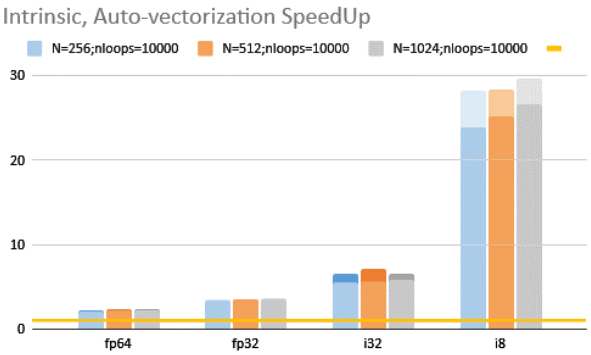

Figura 4-5 – SpeedUp operación REDUCE. Los colores más claros indican SpeedUp de auto-vectorización, lo más sólidos SpeedUp de intrínsecos.

SAXPY

La Figura 4-7 muestra una mejora en todas las situaciones tanto en la optimización mediante intrínsecos como en la auto-vectorizada. El mejor SpeedUp conseguido por parte de la optimización vectorial es de 15x. Esta reducción en la mejora respecto a REDUCE se debe a la baja carga computacional que esta operación tiene, reflejando que los accesos a memoria, (imprescindibles en la optimización vectorial) lastran la optimización. En este caso, la comparativa entre intrínsecos y auto-vectorización muestra un resultado parecido, obteniendo mejores resultados con el uso de intrínsecos.

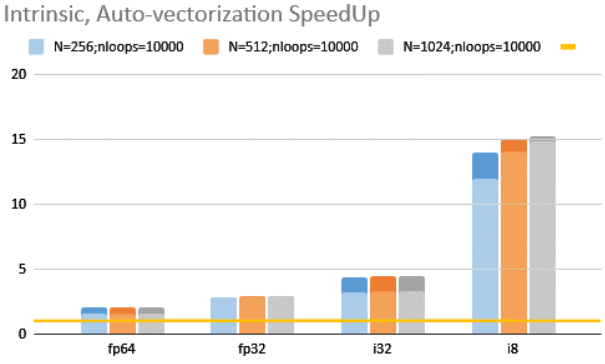

Figura 4-7 – SpeedUp operación SAXPY. Los colores más claros indican SpeedUp de auto-vectorización, lo más sólidos SpeedUp de intrínsecos.

VVADD

La Figura 4-9 muestra un SpeedUp superior a uno con la optimización vectorial aplicada, pero en este caso en menor magnitud, esta operación tiene grandes similitudes con SAXPY ya que ambas comportan una carga computacional baja, incrementándose este aspecto en esta operación. Por tanto, la optimización vectorial se ve limitada por los accesos a memoria y el pequeño tamaño de las memorias caché implicando que en el caso del tipo de dato punto flotante de doble precisión, no se consiga ni 2x de speedup.

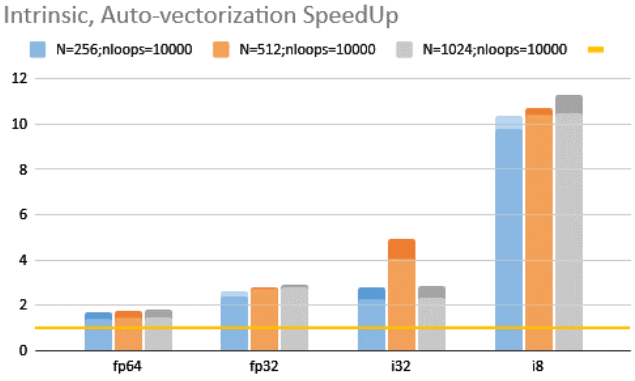

Figura 4-9 – SpeedUp operación VVADD. Los colores más claros indican SpeedUp de auto-vectorización, lo más sólidos SpeedUp de intrínsecos.

MATMUL

Los resultados de la Figura 4-10 reflejan una mejora en todos los casos, mostrando un rendimiento similar entre los intrínsecos y la auto-vectorización.

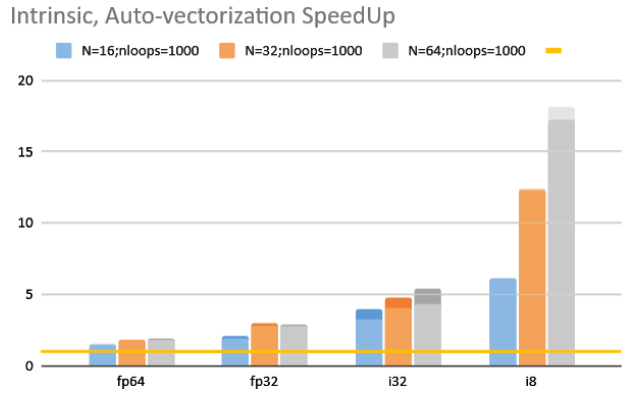

Figura 4-10 – SpeedUp operación MATMUL. Los colores más claros indican SpeedUp de auto-vectorización, lo más sólidos SpeedUp de intrínsecos.

Convolución 2D

La optimización de la convolución 2D ha sido satisfactoria en todos los casos, obteniendo SpeedUps superiores a uno. A pesar de ello, existen grandes diferencias entre tipos de datos entre la optimización mediante intrínsecos y auto-vectorización, llegando a doblar el SpeedUp obtenido utilizando intrínsecos con respecto a la optimización proporcionada por el compilador. Centrándose en el tipo de dato de punto flotante de precisión media, se obtienen SpeedUps limitados tanto por los accesos a memoria como por la computación más compleja que supone este tipo de dato.

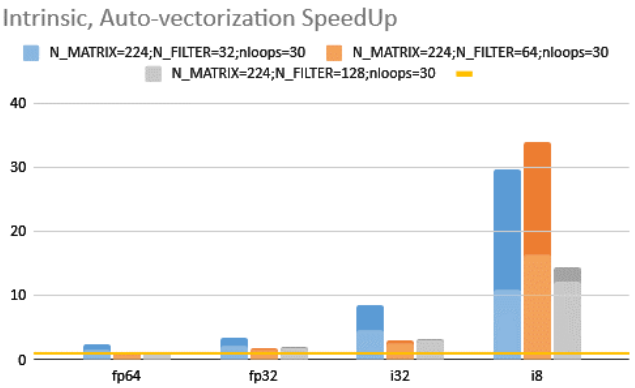

Figura 4-8 – SpeedUp operación Conv2D. Los colores más claros indican SpeedUp de auto-vectorización, lo más sólidos SpeedUp de intrínsecos.

4.2 Optimización vectorial del compresor V2F

Una vez computado y analizado los resultados con las funciones de muestra, da comienzo la tercera etapa de este estudio.

Entorno de pruebas

Es importante dejar constancia del entorno de pruebas utilizado para su futura replicación:

- Periférico: MV-K230, con arquitectura RISC-V
- Sistema operativo sid Debian.
- Compilador: Versión preliminar GCC 14.0.1:20240330
- Archivo de entrada: Fotografía en formato RAW proporcionada por el GICI. Figura 0-3.
- Cuantificador: 5
- Número de árboles: 1
- Predictor: W

Perfilado

El perfilado es una etapa crítica en cualquier optimización de código, esta permite al desarrollador intuir y conocer qué funciones son las que mayor tiempo de cómputo requieren y, por ende, sobre las que el desarrollador debe centrar sus esfuerzos. La perfilación se ha realizado empleando la herramienta gprof y con todas las optimizaciones desactivadas.

Compresor

El perfilado mostrado en la muestra que la función *v2f_entropy_coder_compress_block* consume el 48% del tiempo, por tanto, los esfuerzos se deben centrar en optimizar esa función. La función *v2f_entropy_coder_compress_block* se encarga de ir recorriendo el árbol en función de las muestras de entrada y en el caso de encontrar una hoja, guardar la palabra codificada en buffer.

Descompresor

El perfilado muestra que la función *v2f_entropy_decoder_decode_next_index* consume el 31% del tiempo, por tanto, los esfuerzos se deben centrar en optimizar esa función.
La función *v2f_entropy_decoder_decode_next_index* se encarga de ir cogiendo de los datos de entrada las palabras codificadas, y transformándolas en los símbolos originales.

Optimización vectorial

Una vez conocidas las funciones a optimizar, se procede a optimizar vectorialmente las funciones. Primero de todo se permitirá al compilador auto vectorizar cualquier bucle del código que detecte (y sea vectorizable), seguidamente se comprobará manualmente si se puede mejorar la optimización vectorial de la función perfilada.

Auto-vectorización

Tal como se ha comentado con anterioridad en este escrito, el compilador solo es capaz de auto vectorizar bajo situaciones muy específicas, en este caso, el compilador GCC-14 solo ha sido capaz de optimizar vectorialmente la función *v2f_entropy_coder_sample_to_buffer*. Esta función es la segunda con mayor tiempo de ejecución de la perfilación Codi 0-2, por tanto, es interesante ver si esta optimización aporta una mejora en el tiempo final.

La Figura 4-11 muestra cuatro ejecuciones, la ejecución escalar, la ejecución con un LMUL=1, LMUL=2, LMUL=4. Desgraciadamente la carga computacional del bucle y su profundidad no ha sido suficiente para paliar el overhead introducido al utilizar las instrucciones vectoriales y por ello, el tiempo escalar ofrece un mejor rendimiento.

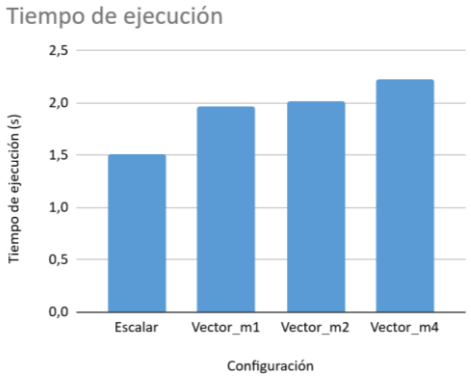

Figura 4-11– Resultados auto-vectorización v2f_entropy_coder_sample_to_buffer.

Intrínsecos

Implementar la optimización mediante intrínsecos es una tarea mucho más complicada y profunda que la auto vectorización, ya que se debe entender el código para asegurar su integridad una vez aplicada la optimización vectorial. En este caso, el compresor V2F es un compresor con estructura de árbol y la función perfilada más costosa es la encargada de recorrer el árbol, esta es una tarea incompatible con la vectorización ya que como se ha comentado con anterioridad las iteraciones de los valores a optimizar deben ser independientes entre ellas. Esto no quita que se puedan tomar otros enfoques para vectorizar la función.

El enfoque que se ha tomado para la vectorización es realizar operaciones de "gathering". En la función perfilada, no hay ninguna instrucción con gran carga computacional, por ello, se vectorizará el recorrido de los símbolos de entrada por el árbol hasta encontrar las palabras codificadas. Es decir, se irán cargando los símbolos de entrada en cada una de las posiciones de un registro vectorial y de esta manera poder realizar una especie de paralelismo con threads.

Esto implica que la salida generada vectorialmente ya no será la misma que la salida escalar, por ello, no será posible decodificar los datos una vez codificados. Pero en cambio, si nos puede dar una idea sobre si este enfoque puede aportar una mejora del rendimiento considerable y, por tanto, comprobar si merece emplear mayor esfuerzo en adaptar el decodificador.

Una vez realizada la implementación y analizando los resultados de la Figura 4-12, la optimización vectorial tiene un SpeedUp de 0,93x con respecto a la versión escalar. Por tanto, no se adaptará el decodificador, ya que los resultados demuestran que no hay una mejora en el tiempo de ejecución. Estos resultados, surgen a raíz del planteamiento utilizado, tal y como se demostró en otros trabajos [3], el uso de instrucciones vectoriales para el control del flujo merma la optimización vectorial y en este caso, toda la optimización tiene ese enfoque y por tanto, no hay carga computacional, aspecto que tal y como se ha demostrado con las funciones de muestra como SAXPY o VVADD, es uno de los casi requisitos para emplear instrucciones vectoriales en una función.

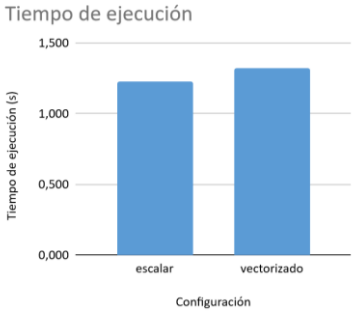

Figura 4-12 – Tiempos de ejecución escalar y vectorizada con intrínsecos.

4.3 Optimización vectorial de JPEG

Como última etapa de este trabajo, se evaluará el rendimiento de las instrucciones vectoriales en el compresor JPEG. Concretamente se utilizará el JPEG-Turbo el cual es un compresor de datos idéntico a JPEG salvo por la importante diferencia de que tiene soporte para operaciones SIMD. El soporte de RVV todavía no ha sido implementado de manera oficial en JPEG-Turbo, pero Zhiyuan Tan, estudiante de la University of Chinese Academy of Sciences lo implementó en su trabajo de graduación [11]. Por tanto, la aportación de este estudio será con la ejecución auto-vectorizada de este compresor. Con el objetivo de poder comparar la versión JPEG-Turbo escalar, la implementada por Zhiyuan Tan y la auto-vectorizada.

Entorno de pruebas

- Periférico: MV-K230, con arquitectura RISC-V
- Sistema operativo sid Debian.
- Compilador: Versión preliminar GCC 14.0.1:20240330
- Archivo de entrada: Fotografía en formato BMP proporcionada por el GICI. Figura 0-3

Comando: *tjbench-static {image} 95 -rgb -qq -nowrite -warmup 10*

Optimización vectorial

Los resultados de la Figura 4-13 muestran que la implementación con intrínsecos consigue un mayor rendimiento con respecto a la ejecución escalar en 1,937x en la compresión y 1,485x en la descompresión. Por otro lado, la auto-vectorización no ha superado en ningún caso el rendimiento de la ejecución escalar. Lo que demuestra que, para aprovechar las capacidades vectoriales, estas se deben tener en cuenta a la hora de desarrollar el código.

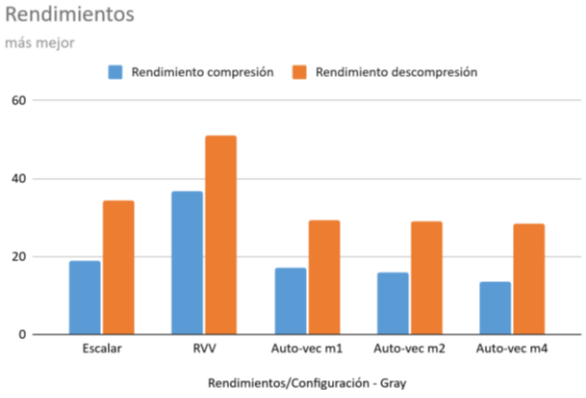

Figura 4-13 – Rendimientos de compresión y descompresión para ejecución escalar, intrínsecos y auto-vectorización con lmul=1,2,4.

5 Conclusión

Como conclusión a este trabajo, se puede afirmar que, con una buena implementación, las instrucciones vectoriales aportan una mejora de rendimiento en todas las situaciones como bien se ha demostrado en esta primera fase del desarrollo con las funciones de muestra. En ellas se ha obtenido un SpeedUp superior a 1 en todos los casos. Por otro lado, cabe señalar que las instrucciones vectoriales no aportan un beneficio en el tiempo de ejecución de manera facil y se requiere de una dedicación expresa para adaptar el código en concreto y que, de esta manera, se puedan aprovechar las instrucciones vectoriales, lo cual puede no ser posible en muchas situaciones. Esto se demuestra con la implementación del compresor V2F en el que la idea inicial era comprobar el rendimiento otorgado por la auto-vectorización pero que en la realidad y debido a la estructura de árbol en la que está basado, ha supuesto adaptar la función resultante de la perfilación mediante el uso de intrínsecos y concretamente con un enfoque al control del flujo de datos. Con JPEG-Turbo se han obtenido resultados similares al V2F, donde la auto-vectorización no ha otorgado ninguna mejora sobre la ejecución escalar, pero al realizar la implementación mediante intrínsecos, el rendimiento mejora 2x sobre la ejecución escalar, demostrando lo redactado previamente.

En el futuro sería adecuado realizar este mismo trabajo en placas más capaces, es decir, que implementen multi-core, o tamaños de registros vectoriales mayores a 128 bits. Así como evaluar otros compresores basados en árboles y comprobar si las instrucciones vectoriales pueden llegar a tener cabida en este tipo de estructuras. Finalmente, sería oportuno realizar el análisis de rendimiento de JPEG-turbo con una implementación oficial de este para la extensión vectorial de RISC-V.

Bibliografía

[1] J. Marquez, «Xataka,» 15 diciembre 2023. [En línea]. Available: https://www.xataka.com/componentes/rics-v-esta-creciendo-rapido-que-algunas-consultoras-creen-que-se-aduenara-25-mercado-soc-2030. [Último acceso: marzo 2024].

[2] R. Jones, «Red Hat Research,» noviembre 2023. [En línea]. Available: https://research.redhat.com/blog/article/risc-v-extensions-whats-available-and-how-to-find-it/. [Último acceso: marzo 2024].

[3] A. Valdivieso, «Implementació d'algorismes vectorials per acceleració HW en sistemes RISC-V per espai,» Bellaterra, 2024.

[4] riscv-v-spec, «GitHub: Vector Extension 1.0, frozen for public review,» 20 Septiembre 2021. [En línea]. Available: https://github.com/riscv/riscv-v-spec/releases/tag/v1.0.

[5] dominiksalvet, «GitHub: RISC-V standard extensions,» [En línea]. Available: https://gist.github.com/dominiksalvet/2a982235957012c51453139668e21fce.

[6] M. H.-C. e. al., 1 Mayo 2022. [En línea]. Available: https://raw.githubusercontent.com/gici-uab/v2f_codec/master/doc/user_manual.pdf. [Último acceso: Marzo 2024].

[7] J. Martins, «ASANA,» 19 Enero 2024. [En línea]. [Último acceso: Marzo 2024].

[8] J. P. E. Group, «JPEG,» [En línea]. Available: https://jpeg.org/.

[9] D. C., «libjpeg-turbo,» [En línea]. Available: https://libjpeg-turbo.org/.

[10] R. Espasa, «The RISC-V Vector ISA,» de *Conferencia de Esperanto Technologies*, California, 2017.

[11] «Add RISC-V vectors support,» [En línea]. Available: https://github.com/libjpeg-turbo/libjpeg-turbo/issues/620.

[12] Elecard Company, «Vector Instrucions. Part I,» *Medium*, 2022.

[13] A. Raveendran, V. B. Patil, D. Selvakumar y V. Desalphine, «A RISC-V instruction set processor-micro-architecture design and analysis,» *2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA)*, nº doi: 10.1109/VLSI-SATA.2016.7593047., pp. 1-7, 2016.

[14] R. Denis-Courmont, «remlab,» [En línea]. Available: https://www.remlab.net/op/riscv-v-draft-1.shtml.

[15] riscv-non-isa, «GitHub: rvv-intrinsic-doc,» [En línea]. Available: https://github.com/riscv-non-isa/rvv-intrinsic-doc.

APÉNDICE

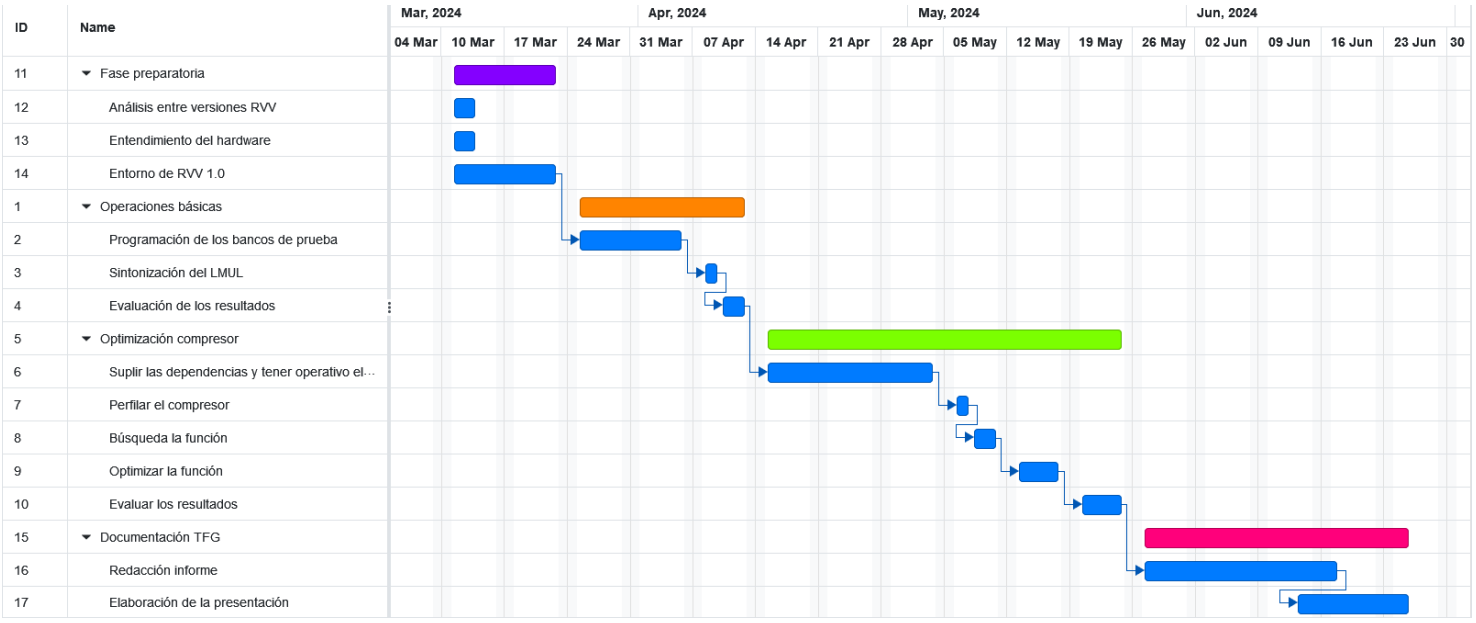

Figura 0-2 - Planificación inicial del proyecto en forma de Diagrama de Gantt

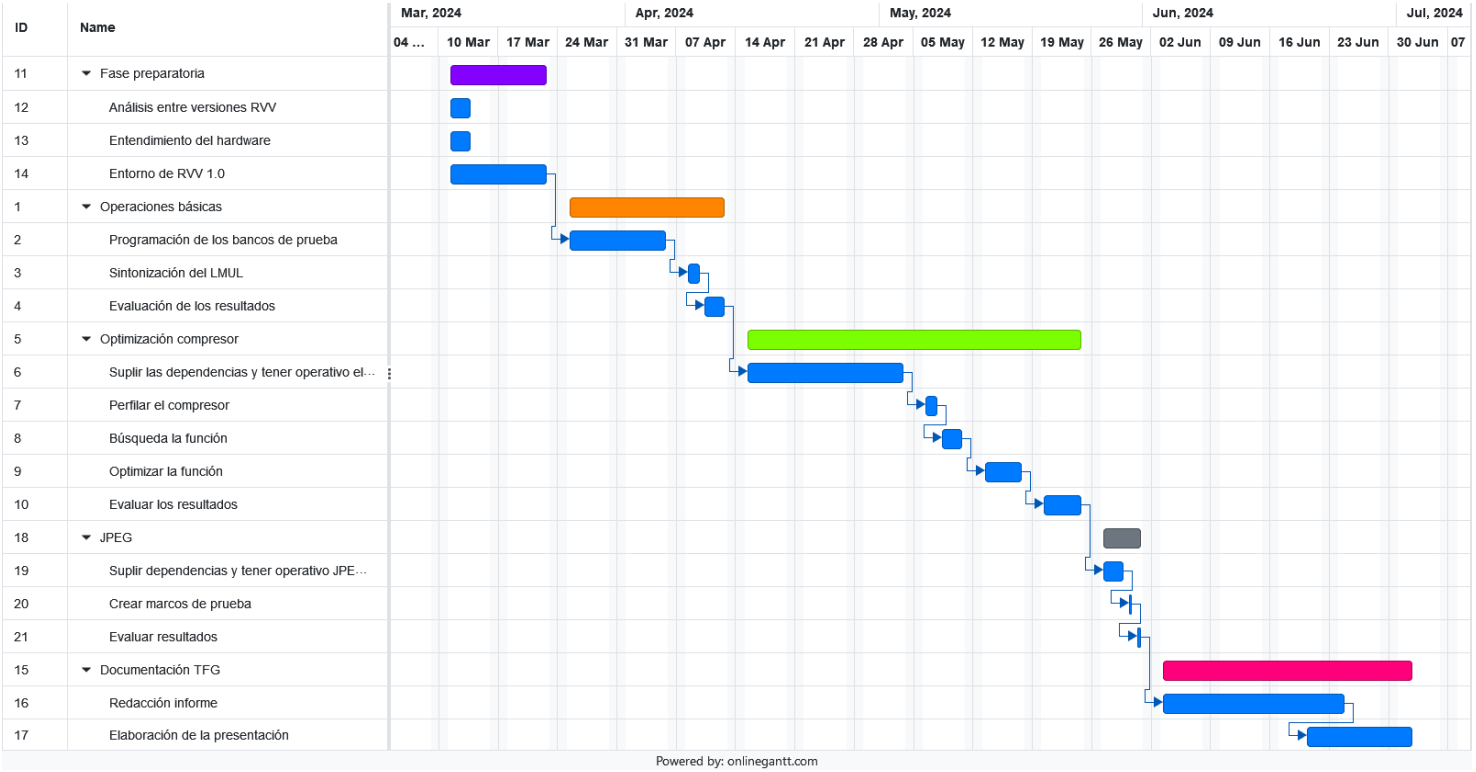

Figura 0-1 - Planificación final del proyecto en forma de Diagrama de Gantt

% time	cumulative seconds	self seconds	self calls	Ts/call	total Ts/call	name
48.46	1.42	1.42				v2f_entropy_coder_compress_block
16.55	1.91	0.48				v2f_decorrelator_map_predicted_sample
8.87	2.17	0.26				v2f_entropy_coder_buffer_to_sample
8.87	2.42	0.26				v2f_file_read_big_endian
6.83	2.62	0.20				v2f_decorrelator_apply_left_prediction
4.78	2.77	0.14				v2f_quantizer_apply_uniform_division
3.07	2.85	0.09				v2f_file_read_forest
0.34	2.87	0.01				v2f_decorrelator_unmap_sample
0.34	2.88	0.01				v2f_entropy_coder_fill_entry
0.34	2.88	0.01				v2f_entropy_coder_sample_to_buffer
0.34	2.90	0.01				v2f_file_destroy_read_forest
0.17	2.90	0.01				v2f_decorrelator_create

Codi 0-1 – Perfilado del compresor

% time	cumulative seconds	self seconds	self calls	Ts/call	total Ts/call	name
30.94	0.73	0.73				v2f_entropy_decoder_decode_next_index
17.17	1.14	0.41				v2f_entropy_coder_sample_to_buffer
10.60	1.39	0.25				v2f_decorrelator_unmap_sample
8.05	1.58	0.19				v2f_quantizer_inverse_uniform
8.05	1.77	0.19				v2f_entropy_decoder_decompress_block
7.21	1.94	0.17				v2f_file_write_big_endian
6.15	2.08	0.15				v2f_entropy_coder_buffer_to_sample
5.93	2.22	0.14				v2f_decorrelator_inverse_left_prediction
2.54	2.28	0.06				v2f_file_read_forest
2.12	2.33	0.05				v2f_file_read_big_endian
0.85	2.35	0.02				v2f_decorrelator_map_predicted_sample
0.42	2.36	0.01				v2f_file_destroy_read_forest

Codi 0-2 – Perfilado del descompresor

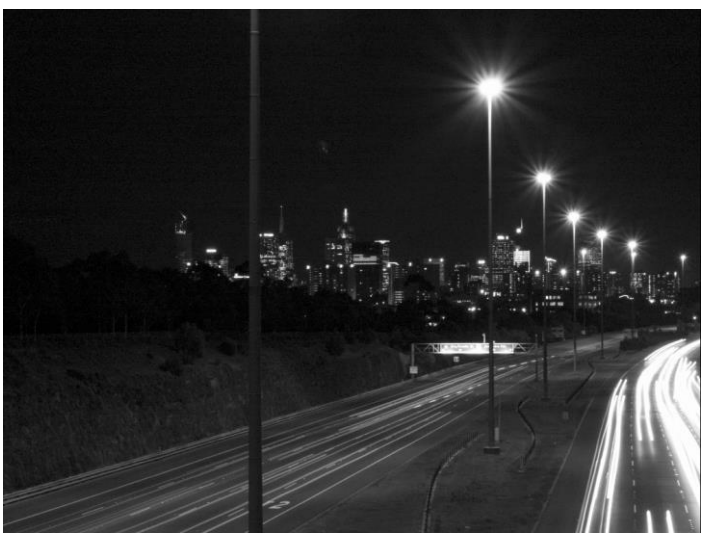

Figura 0-3 – Imagen proporcionada por el GICI.