Tecnológico de Costa Rica

Escuela de Ingeniería en Computadores
(Computer Engineering School)

Programa de Licenciatura en Ingeniería en Computadores
(Licentiate Degree Program in Computer Engineering)

Compilador para el acelerador de aprendizaje profundo ArPaReUV-AP

(Compiler for the ArPaReUV-AP deep learning accelerator)

Informe de Trabajo de Graduación para optar por el título de Ingeniero en Computadores con grado académico de Licenciatura

(Report of Graduation Work in fulfillment of the requirements for the degree of Licentiate in Computer Engineering)

Steven Badilla Soto

Cartago, junio de 2025
(Cartago, June, 2025)

Declaro que el presente Proyecto de Graduación ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos propios.

En los casos en que he utilizado bibliografía he procedido a indicar las fuentes mediante las respectivas citas bibliográficas. En consecuencia, asumo la responsabilidad total por el trabajo de graduación realizado y por el contenido del correspondiente informe final.

Steven Badilla Soto

Cartago, 29 de marzo de 2025

Céd: 208080680

Instituto Tecnológico de Costa Rica
Área Académica de Ingeniería en Computadores
Proyecto de Graduación
Tribunal Evaluador

Proyecto de Graduación defendido ante el presente Tribunal Evaluador como requisito para optar por el título de Ingeniero en Computadores con el grado académico de Licenciatura, del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal

Profesora Lector

Profesor Lector

Jason Leitón Jiménez
Profesor Asesor

Los miembros de este Tribunal dan fe de que el presente trabajo de graduación ha sido aprobado y cumple con las normas establecidas por el Área Académica de Ingeniería en Computadores.

Cartago, mayo de 2025

Instituto Tecnológico de Costa Rica
Área Académica de Ingeniería en Computadores
Proyecto de Graduación
Tribunal Evaluador
Acta de Evaluación

Proyecto de Graduación defendido ante el presente Tribunal Evaluador como requisito para optar por el título de Ingeniero en Computadores con el grado académico de Licenciatura, del Instituto Tecnológico de Costa Rica.

Estudiante: *Steven Badilla Soto*

Nombre del Proyecto: *Compilador para el acelerador de aprendizaje profundo ArPaReUV-AP*

Miembros del Tribunal

_____ _____

Profesora Lector Profesor Lector

_____

Jason Leitón Jiménez
Profesor Asesor

Los miembros de este Tribunal dan fe de que el presente trabajo de graduación ha sido aprobado y cumple con las normas establecidas por el Área Académica de Ingeniería en Computadores.

Nota final del Proyecto de Graduación: _____

Cartago, mayo de 2025

Resumen

Palabras clave: Compiladores, aprendizaje profundo, aceleradores, LLVM, Clang, SIMD, RISCV.

El desarrollo de aceleradores de *hardware* especializados para tareas de aprendizaje profundo ha tomado popularidad y la que seguirá tomando por las ventajas que pueden ofrecer sobre equipos comerciales como GPUs en ciertos contextos. El desarrollo de estos viene acompañado del desarrollo de herramientas de *software*, como compiladores, que faciliten el uso del acelerador. Este proyecto se enfoca en el desarrollo de un compilador para el acelerador SIMD ArPaReUV-AP, que permite compílar código C a código RISCV, para esto se extendió el compilador LLVM y su *frontend* Clang, en este documento se evidencia el proceso mediante el que se agregaron nuevas instrucciones al compilador, en forma de *instrinsics*, con las que se pueda interactuar con el acelerador, estas instrucciones se agregaron al *backend* de RISCV aprovechando la extensión de RVV. Una de las tareas que se desean optimizar utilizando el acelerador es la convolución, para el proyecto se implementó el algoritmo de convolución que se espera ejecutar con el acelerador, se realizó una implementación SIMD utilizando las instrucciones que se agregaron y una SISD utilizando únicamente C, con esto se compararon ciertas características de los códigos generados y así se observaron los resultados que tiene el uso de estas instrucciones. De estos resultados se concluyó que para el algoritmo de la convolución el uso de las instrucciones agregada representa una mejora en la cantidad total de instrucciones ensamblador, de usos de memoria y registros, además de generar archivos más ligeros, con el costo de necesitar conocimiento de la arquitectura al realizar el código en C y de un mayor tiempo para compilar.

Abstract

Keywords: Compilers, deep learning, hardware accelerators, LLVM, Clang, SIMD, RISCV.

The development of specialized hardware accelerators for deep learning tasks has gained popularity and will continue to gain popularity due to the advantages they can offer over commercial equipment such as GPUs in certain contexts. Their development is accompanied by the development of software tools, such as compilers, that facilitate the use of the accelerator. This project focuses on the development of a compiler for the ArPaReUV-AP SIMD accelerator, which allows C code to be compiled to RISCV code. For this, the LLVM compiler and the Clang frontend were extended. This document shows the process by which new instructions were added to the compiler, in the form of intrinsics, that can be used to interact with the accelerator. These instructions were added to the RISCV backend taking advantage of the RVV extension. One of the tasks that is intended to be optimized using the accelerator is convolution, for the project the convolution algorithm that is expected to be executed with the accelerator was implemented, a SIMD implementation was made using the added instructions and also a SISD implementation using only C. With this, certain characteristics of the generated codes were compared and thus the results of using these instructions were observed. From these results, it is concluded that for the convolution algorithm, the use of the added instructions represents an improvement in the total number of assembler instructions, memory and register usage, in addition to generating lighter files, at the cost of needing knowledge of the architecture when writing the code in C and a longer compilation time.

Dedico este proyecto a mis padres, Jenny Solo Ulate y Braulio Badilla Murillo, quienes me han apoyado y brindado herramientas durante toda mi vida para tener la posibilidad de estudiar y llegar hasta la última etapa de mi educación superior.

Agradecimientos

A mi madre Jenny Soto Ulate, quien durante mi infancia impulsó mi gusto por las ciencias y la matemática que luego me llevó a querer estudiar ingeniera. Quien además durante toda mi vida me ha dado apoyo y herramientas que me han permitido el poder estudiar durante todo este tiempo y llegar hasta este punto en mi carrera.

A mi padre Braulio Badilla Murillo, quien al igual que mi madre me ha dado apoyo y herramientas durante toda mi vida para llegar hasta aquí y que siempre me impulso a que estudiará para poder hacer trabajos más cómodos de los que él tuvo que hacer durante su vida.

Ambos han sido una inspiración para mí, me han demostrado el valor del cariño como forma de motivar y ayudar a las personas a crecer, y que el esfuerzo es necesario para cumplir nuestras metas y salir adelante. Me mostraron amor y apoyo incondicional toda mi vida y fueron mi mayor inspiración para querer ser ingeniero, y sin ellos nada de esto habría sido posible para mí.

Agradezco también a mi hermana Shannen, a mis abuelas Angela y Flory, que siempre mostraron interés, apoyo y orgullo por lo que estaba estudiando, y que de muchas formas me acompañaron en el proceso, mediante palabras de apoyo y muestras de cariño, que fueron importantes durante los días difíciles cuando tenía dudas de querer seguir.

Agradezco a mi pareja Natalia quien durante el desarrollo de este proyecto me ha acompañado cuando me sentía abrumado y me ha brindado su cariño para mejorar mis ánimos. Quien además me ayudó con su compañía en sesiones de estudio que me permitieron avanzar en el proyecto de forma más eficaz. Sin duda durante este proceso su ayuda fue de gran valor para poder finalizar.

Finalmente quiero agradecer a los profesores que me han enseñado durante la carrera y gracias a los cuales sé que tengo una formación profesional completa y de calidad. Especialmente al profesor Jeison Leiton quien me asesoró y acompañó en el desarrollo del proyecto, para poder tener certeza que el informe cumple con los requisitos que se esperan de un trabajo de graduación.

Steven Badilla Soto

Cartago, 10 de junio de 2025

Índice general

Índice de figuras

Índice de tablas

Lista de símbolos y abreviaciones

Abreviaciones

SIGLAS SIGNIFICADO

Capítulo 1

Introducción

En esta sección del informe se habla de forma general sobre el proyecto que se está realizando, brindando una descripción de este, ademas de información, relacionada a la organización con la que se desarrolla el proyecto, otros proyectos existentes similares, y cual es el problema especifico que se busca solucionar, para así poder justificar la realización de este trabajo.

Además de esto se presentan los objetivos del proyecto y se define el alcance que este tendrá, posibles limitaciones y que es lo que se planea entregar.

1.1 Descripción general del proyecto

Este proyecto forma parte de otro proyecto desarrollado por Allan Navarro, quien trabaja en el diseño de una arquitectura SIMD para un acelerador enfocado en optimizar operaciones comunes en tareas de aprendizaje profundo. Una de estas tareas, por ejemplo, es la convolución una operación matemática muy importante en la mayoría de modelos de redes neuronales y de modelos de visión por computadora en general.

El acelerador que se está desarrollando es llamado ArPaReUV-AP, y tiene como una de sus características ser reconfigurable, permitiendo versatilidad en el tamaño y cantidad de registros que tienen los elementos de procesamiento y en la cantidad de elementos de procesamiento. Como es común a la hora de diseñar una arquitectura esta viene acompañada de un ISA (Instruction Set Architechture) propio, es decir, un conjunto de instrucciones definidas específicamente para poder utilizar y ejecutar órdenes en el acelerador. El acelerador se va a ejecutar en una FPGA Kria KV260.

El proyecto para el que se desarrolla este informe consiste en un compilador para este acelerador, que sea capaz de generar código en lenguaje máquina, es decir secuencias binarias, que puedan ser entendidas por el acelerador. Estas secuencias binarias deben ser generadas a partir de un código en lenguaje de alto nivel, más específicamente desde un código escrito en el lenguaje de programación C.

La idea con este compilador es que se pueda hacer uso de instrucciones especiales en el código en lenguaje de alto nivel con las cuales poder interactuar específicamente con el acelerador, para así hacer uso eficiente de la memoria de la FPGA a la hora de cargar la información a procesar, ademas de poder asignar las tareas especificas que se quieren ejecutar para ser procesadas por el acelerador.

El proceso de compilación cuenta con varias etapas, para analizar el código fuente en busca de errores léxicos o sintácticos, y así asegurarse que se cumplen las restricciones del lenguaje, en especial en las instrucciones especializadas que se esperan implementar, ademas de estas etapas vara verificar la estructura es importante la etapa de optimización, buscando aprovechar al máximo los recursos del hardware, y generar un código maquina eficiente.

1.2 Antecedentes

En esta sección hablará sobre la organización para la que se va a desarrollar el proyecto, la cual está conformada por dos universidades públicas, además se va a especificar las áreas de conocimiento que abarca este proyecto relacionadas con la ingeniera en computadores, y por último se va a comentar y analizar brevemente algunos trabajos similares a este proyecto.

1.2.1 Descripción de la empresa

El proyecto es desarrollado con el grupo de investigación ECASLab, este grupo está compuesto tanto por estudiantes como por profesores del Instituto Tecnológico de Costa Rica y de la Universidad de Trieste.

El Instituto Tecnológico de Costa Rica [2] es una de las universidades publicas del país, fue fundada en 1971 y desde este año se dedica a la preparación de profesionales e investigación relacionadas con las áreas de tecnología e ingeniara principalmente, su cede principal está ubicada en la Provincia de Cartago pero cuenta con sedes en San Carlos de Alajuela, en Limon y en San José, en total el Tecnológico cuenta con unos 10.000 estudiantes activos divididos en las distintas escuelas y áreas académicas.

La Universidad de Trieste [1] es también una universidad publica, esta se encuentra en la ciudad de su mismo nombre en Italia y fue fundada en 1924, en ella existen 10 departamentos principales abarcando una gran diversidad de áreas, desde tecnología e ingeniera hasta ciencias sociales y ciencias políticas. Esta universidad destaca en el área de investigación y es muy prestigiosa, siendo recocida de forma internacional como una de las mejores universidades de Italia.

1.2.2 Áreas de conocimiento

El proyecto abarca principalmente el área de compiladores y traductores, ya que lo que se planea desarrollar es un compilador para un acelerador de aprendizaje profundo que sea capaz de compilar código desde un lenguaje de alto nivel como C a lenguaje de máquina que el acelerador pueda utilizar.

Además, de esto se involucra el área de arquitectura y sistema empotrados, ya que el acelerador para el que se desarrolla el compilador tiene el objetivo de optimizar las tareas para trabajar en sistemas empotrados, y para el funcionamiento óptimo del compilador se debe entender el ISA y la arquitectura del acelerador, para realizar las optimizaciones del código necearías ademas de aprovechar los recursos de hardware al máximo mediante la compilación. También es necesario entender la arquitectura ya que esta al ser reconfigurable será un reto a la hora de desarrollar el compilador que este pueda adaptarse, al generar el código, a la cantidad y tamaño variable de los registros y la cantidad de elementos de procesamiento.

1.2.3 Trabajos similares

En esta sección se mencionan algunos trabajos similares al que se desarrolla en este proyecto, mediante una descripción breve de estos proyectos que permita observar algunas similitudes y relaciones, y poder también recalcar como se diferencian respecto a este proyecto.

CUDA es una arquitectura para GPUs introducida por NVIDIA en 2006, con la que se buscaba ampliar la capacidad de las GPUs de trabajar en tareas de propósito general y no solo de procesamiento de gráficos, junto a estas mejoras de hardware NVIDIA lanzaría poco después CUDA C un compilador que permitía crear programas que interactuaban con la GPU desde el lenguaje de programación C, con el fin de que los programadores no tuvieran que hacer sus programas en lenguajes orientados a gráficos como OpenGL GLSL o Microsoft HLSL [23], a lo largo de los años CUDA ha evolucionado existiendo ahora distintas versiones de la herramienta y el compilador que permiten su uso en otros lenguajes de alto nivel.

El proyecto desarrollado se relaciona ya que también busca el desarrollo de un compilador que permita a los futuros desarrolladores que quieran utilizar el acelerador poder interactuar con este mediante un lenguaje de alto nivel popular como lo es C. A diferencia de CUDA el compilador diseñado esta hecho para trabajar con un acelerador de aprendizaje profundo que va a ser ejecutado en una FPGA para así poder ser reconfigurable y no diseñado para trabajar con GPUs.

El proyecto ONNC (Open Neural Network Compiler) es un framework que busca conectar de forma rápida modelos en formato ONNX con aceleradores de aprendizaje profundo de distintos desarrolladores, esto con el fin de apoyar a la comunidad de investigación en el área de desarrollo de aceleradores [19]. ONNX (Open Neural Network Exchange) es

un ecosistema que permite a los desarrolladores de IA escoger las mejores herramientas para su proyecto, al brindar un formato de código abierto para representar modelos de IA [12]. Con esto facilitando el migrar modelos entre herramientas comunes en el desarrollo y permitiendo combinar estas para tener resultados óptimos [19].

ONNC también busca el desarrollo de compiladores para aceleradores de aprendizaje profundo, pero a diferencia de ONNC el proyecto desarrollado, primero se limita a un compilador para un único acelerador y no busca crear un framework para el desarrollo de compiladores como tal. Ademas el compilador desarrollado va a partir de código en C que permita instrucciones SIMD especializadas para interactuar con el acelerador, pero este código no representa un modelo de aprendizaje profundo, mientras que ONNC se enfoca específicamente en compilar modelos de aprendizaje profundo que estén en el formato ONNX para poder utilizarse en aceleradores.

Otro proyecto similar es Apache TVM [11], este proyecto desarrolla un compilador optimizado para distribuir cargas de trabajo de aplicaciones de aprendizaje profundo de manera óptima en diferentes tipos de hardwares, al separar la descripción del algoritmo, de la calendarización y la interfaz de hardware.Apache TVM puede tomar modelos descritos en formatos de distintos frameworks populares (como pytorch, MXNet, caffe2 etc) realizar optimizaciones de alto y bajo nivel y generar código para hardwares específicos (como Raspberry Pi, GPUs y aceleradores que trabajan en FPGA).

Similar que ONNC, Apache TVM es un proyecto donde se desarrolla un compilador que trabaje directamente con representaciones de modelos de aprendizaje profundo, en este caso en varios formatos de frameworks populares, además de ser capaz de compilar a distintos tipos de hardwares, así que es similar al proyecto desarrollado en el sentido de implementar un compilador para poder trabajar con aceleradores en FPGA, pero igual que con ONNC, el proyecto que se está desarrollando se diferencia de Apache TVM en que este busca compilar código más general escrito en un lenguaje de alto nivel, C en este caso, para ser utilizado por un único acelerador, el ArPaReUV-AP.

1.3 Planteamiento del problema

En esta sección se va a presentar el contexto del problema que se quiere trabajar mediante el desarrollo de este proyecto, hablando de las tendencias actuales y de los últimos años en el desarrollo de hardware especializado para aprendizaje profundo y las herramientas que los complementan, con esto se espera justificar la importancia que tiene el desarrollar este proyecto y poder definir de forma concreta que es lo que se espera lograr.

1.3.1 Contexto del problema

Las tendencias modernas indican que el uso de sistemas heterogéneos va a ser parte fundamental en la evolución de sistemas computacionales [23], desde laptops hasta su-

percomputadoras, estos sistemas van a estar compuestos por CPUs de múltiples núcleos, hardware de propósito especial y aceleradores paralelos, pero esto viene acompañado de desafíos nuevos, como la necesidad de herramientas y métodos que faciliten el desarrollo de software para ser utilizado en el hardware de propósito especial. Hace no mucho tiempo la computación en paralelo era un área de nicho, pero en los últimos años se ha vuelto una necesidad para cualquier programador tener entendimientos sobre la computación paralela.

En los inicios del uso de GPUs estas estaban diseñadas estrictamente para el procesamiento de gráficos, y la interacción con estas restringida a APIs gráficas como OpenGL y DirectX, aun así algunos programadores vieron la oportunidad de utilizar la alta capacidad de procesamiento en paralelo de las GPUs para tareas de propósito general, mediante trucos de software que les permitieran representar otros problemas como problemas de procesamiento gráfico, si bien esto funcionaba por el contexto era sumamente limitado y complejo lo que limitada la comunidad de desarrolladores interesados. Como respuesta a estas necesidades es que Nvidia lanza la arquitectura CUDA permitiendo a sus GPUs ser usadas para propósitos generales y poco tiempo después lanza el compilador CUDA C que permitía interactuar con sus GPUs desde un lenguaje de alto nivel como lo es C facilitando el desarrollo de código que aprovechase las GPUs [23].

En la actualidad el uso de IA ha aumentado mucho su popularidad siendo una herramienta que provee muchas ventajas para distintas áreas, las redes neuronales profundas, una forma muy popular de aprendizaje profundo, son uno de los modelos más populares en el desarrollo de aplicaciones de IA como la visión por computadora, robótica o procesamiento de lenguaje natural [26]. Estos avances requieren de mucho poder computacional y la mayoria de implementaciones se han venido apoyando en hardware como CPUs y GPUs, sin embargo, en los últimos años se ha popularizado el diseño de aceleradores especializados, que mejoran la eficiencia computacional, energética y de espacio. A pesar de la creciente popularidad de aceleradores especializados, aún existen muchos retos para estas alternativas, entre ellas la dificultad para generar representaciones optimizadas de los modelos de redes neuronales para estos hardwares especializados.

Es por eso que recientemente han surgido diferentes frameworks de compilación que buscan el generar compiladores capaces de compilar y optimizar las representaciones y operaciones comunes de redes neuronales para que puedan ser ejecutadas en aceleradores personalizados.

1.3.2 Justificación del problema

El desarrollo de aceleradores personalizados para aprendizaje profundo es cada vez más popular, entre algunos de los métodos que se utilizan es el hacer aceleradores diseñados para ejecutarse en una FPGA y así aprovechar la versatilidad que brindan estas ademas de sus ventajas computacionales.

La necesidad de utilizar aceleradores especializados nace en la búsqueda de reducir el

consumo energético, y reducir el espacio que requiere, y así poder mejorar el rendimiento de aplicaciones de aprendizaje profundo en contextos donde los recursos son más limitados, por ejemplo, en sistemas embebidos que no puedan utilizar procesamiento en la nube por restricciones de latencia o conectividad así que requieren de opciones de hardware que se adapten a estas necesidades.

Además, de mejorar el funcionamiento de aplicaciones de aprendizaje profundo en sistemas embebidos o con recursos limitados, el optar por alternativas más eficientes puede tener un impacto beneficioso en contextos medioambientales, ya que por los altos costos computacionales y energéticos de los modelos de aprendizaje profundo su entrenamiento y uso puede afectar negativamente al medio ambiente. Pero al estas aplicaciones ser de tanta utilidad y popularidad la solución no debe ser no aprovecharlas pero buscar formas de mejorar su consumo e impacto ambiental.

El desarrollo de estos aceleradores especializados viene de la mano con el desarrollo de herramientas de software que faciliten su uso, por eso es importante que se desarrolle en forma paralela un compilador que permita a futuros desarrolladores que quieran hacer aplicaciones que aprovechen este hardware poder hacerlo en lenguajes de alto nivel que ya conozcan para que no se desmotiven o se vean obligados a aprender y programar en código ensamblador del acelerador. La importancia de estas herramientas de software se pudo observar en el inicio del uso de GPUs donde antes de tener herramientas que facilitaran la interacción con estas la comunidad de desarrolladores que querían aprender a programar y trabajar con GPUs era limitada ya que requería primero aprender herramientas muy especificas que eran las únicas que permitían interactuar con estas.

1.3.3 Definición concreta del problema

El desarrollo de aceleradores de hardware especializados para aprendizaje profundo es cada vez más necesario, y para que los programadores puedan usar el acelerador desarrollado de forma cómoda, rápida y sin tener que profundizar demasiado en su arquitectura es necesario el desarrollo de un compilador que permita interactuar con el acelerador mediante lenguajes de programación de alto nivel que sean ya conocidos.

1.4 Objetivos del proyecto

1.4.1 Objetivo general

Desarrollar un compilador con el cual se genere lenguaje de maquina optimizado y entendible para el acelerador ArPaReUV-AP a partir de lenguaje natural utilizando herramientas que facilitan la aplicación de las distintas fases de la compilación al código en lenguaje natural.

1.4.2 Objetivos específicos

1. Investigar sobre las mejores herramientas de compilación en búsqueda de que sea un proceso eficiente según las características de la arquitectura mediante el análisis del estado del arte en esta área.

2. Implementar las fases de la compilación necesarias para que se pueda hacer la transformación del lenguaje natural al lenguaje máquina, utilizando las herramientas encontradas durante la investigación.

3. Comparar códigos C y ensamblador que utilizan la extensión de instrucciones del compilador con versiones que no la utilizan para verificar que ventajas o desventajas genera el uso de la extensión, esto mediante la implementación de diferentes algoritmos en C y compilandolos a ensamblador.

4. Detallar mediante distintas documentaciones el proceso de desarrollo del proyecto, de forma que las personas interesadas puedan darle un seguimiento y validación, esto utilizando herramientas de edición de texto y repositorios de código virtuales.

1.5 Alcances, entregables y limitaciones del proyecto.

1.5.1 Alcances

Los alcances definidos para el proyecto según los objetivos y lo establecido con la organización son.

- Implementar un compilador capaz de compilar código de alto nivel en C o C++ a código en lenguaje maquina según el ISA establecido para el acelerador. Este código en lenguaje de alto debe permitir el uso de instrucciones definidas para poder interactuar con el acelerador.

- El compilador debe ser capaz de realizar un análisis léxico y sintáctico del código para asegurar que cumple con la estructura necesaria.

- El compilador debe aplicar técnicas de optimización, para generar código en lenguaje máquina eficiente y aprovechar al máximo los recursos de la FPGA utilizada y la arquitectura del acelerador.

- Debe hacerse un código en el lenguaje de alto nivel definido con el cual poder verificar el funcionamiento del compilador, este se debe probar utilizando el acelerador ejecutado en la FPGA. El código no requiere una alta complejidad ni ser un programa de aprendizaje profundo específicamente, solo debe ser suficiente para poder verificar profundamente el funcionamiento del compilador.

1.5.2 Entregables

En la Tabla 1 se mencionan los diferentes entregables esperados para el proyecto, cada uno con un identificador único, una descripción y forma para verificar su validez y completitud.

ID	Entregable	Descripción	Verificación
E.1	Informe escrito del proyecto	Documento que define el proyecto, abarcando desde una introducción al proyecto, información referente a la investigación de herramientas y conceptos importantes relacionados con el proyecto, descripción del proceso de diseño y justificación de las decisiones tomadas, análisis de los resultados obtenidos y de las conclusiones finales del trabajo.	Revisión y aprobación del representante de la organización.
E.2	Código fuente del compilador	Código documentado y modular de la implementación de las diferentes etapas de compilación que permitan llevar código C con instrucciones especiales a código en el lenguaje máquina que consume el acelerador	Pruebas utilizando el acelerador en la FPGA Kria KV260 para observar que se obtienen los resultados esperados según el código C brindado.
E.3	Código C de prueba	Código de alto nivel escrito en C que permita probar el correcto funcionamiento del compilador, que abarque la prueba de distintos casos de uso y de todas las instrucciones que soporta el acelerador, mediante instrucciones especializadas que representan estas operaciones.	Revisión por parte del desarrollador del acelerador, donde pueda verificar que se prueban todas las operaciones que soporta el hardware.
E.4	Repositorio	Enlace para el acceso a un repositorio en linea donde se pueda acceder a el código generado, tanto del compilador como códigos de pruebas, y se brinde información sobre dependencias y el proceso de instalación y configuración del entorno necesario para el correcto funcionamiento del compilador, además de información sobre el autor y el proyecto	Revisión por parte del representante de la organización para verificar que se encuentra lo necesario en el repositorio y que la información es suficiente para enteneder y utilizar el proyecto de forma clara.

Tabla 1: Entregables para cada objetivo

1.5.3 Limitaciones del proyecto

- El código de alto nivel a compilar debe estar escrito en el lenguaje de programación C o en C++.
- Para la verificación el acelerador debe ser ejecutado en una FPGA Kria KV260.
- Ya que el presupuesto de la organización es limitado se debe optar para el desarrollo del compilador por herramientas de código abierto, en caso de necesitar usar alguna herramienta que no lo sea se debe consultar con la organización o buscar una alternativa que lo sea.
- El compilador debe ser versátil respecto al tamaño y cantidad de registros, además de la cantidad de elementos de procesamiento, ya que la arquitectura es reconfigurable y estos pueden variar en cada ejecución.

Capítulo 2

Marco teórico

En esta sección se procederá a describir los conceptos necesarios para entender el proyecto en su totalidad, abarcando información sobre el concepto y operaciones relacionadas al aprendizaje profundo, información relacionada a los aceleradores de aprendizaje profundo, especialmente a los enfocados para trabajar en FPGA, y por ultimo los conceptos relacionados al proceso de compilación.

2.1 Aprendizaje Profundo

El aprendizaje profundo (AP) es un área de la inteligencia artificial (IA) que nace a partir del uso de modelos en forma de redes neuronales, el AP es caracterizado por el uso de muchas capas entre la entrada y la salida que permiten muchas etapas de procesamiento no lineal [4]. El objetivo de la IA es generar dispositivos capaces de percibir su entorno y así ser capaces de actuar de una forma que maximice su probabilidad de tener éxito en las tareas para las que se desarrolla[24].

Esta área ha visto un gran interés al mostrar resultados muy éxitos en muchas áreas de trabajo distintas[4]. Algunos ejemplos de aplicaciones para el AP son, la visión por computadora, descubrimiento de fármacos, análisis de imágenes médicas, análisis semántico y de lenguaje natural, conducción automática y muchos más campos, en los que se trabaja ya, o que están esperando para ser explorados[24]. Gracias a esta versatilidad la investigación en las áreas de la IA especialmente en AP crece cada vez más con el paso del tiempo.

2.1.1 Redes Neuronales Convolucionales

Las Redes Neuronales Convolucionales (CNN, siglas del inglés) son una de las formas de redes neuronales más representativas del AP, y sus aplicaciones en la visión por computadora han permitido avances que parecían imposibles hace algunos años [18]. Es tanto el impacto que han tenido los modelos de AP, que la investigación en detección de objetos

se considera separada en la era de los modelos tradicionales, antes del 2014, y la era de los modelos de AP con modelos como las redes neuronales profundas (DNN, siglas del inglés) y las CNNs [5].

Las CNN suelen estar conformadas por combinaciones de tres tipos de capas, las capas convolucionales, las capas de pooling, y las capas completamente conectadas [21]. La capa más característica es la capa de convolución, donde los parámetros a optimizar son conocidos como kernels.

En el contexto de CNNs la operación utilizada es la convolución entre matrices, esta es una operación que se realiza entre dos matrices, supongamos la matriz A de tamaño n x m y la matriz B de tamaño p x q, el resultado va a ser la matriz C de tamaño (m + p - 1) x (n + q - 1) y va a estar dada por la siguiente ecuación [22].

$$C(i,j) = [A * B](i,j) = \sum_r \sum_s A(r,s)B(i-r+1, j-s+1) \tag{2.1}$$

donde

$$r \in \{\max(1, i-p+1), \ldots, \min(i,m)\} \tag{2.2}$$

y

$$s \in \{\max(1, j-q+1), \ldots, \min(j,n)\} \tag{2.3}$$

A la matriz B se le conoce como filtro o kernel y suele ser de tamaño impar [22]. Lo que nos dice la ecuación 2.1 es que B se voltea tanto horizontal como verticalmente, luego se superpone el centro de B en la primera celda de A, se multiplican todos los valores de las celdas de B con los valores de las celdas de este subconjunto de A y se suman los resultados, esto genera la primera celda de C, después se mueve el centro de B a la siguiente celda de A y se repite este proceso hasta llegar a la última celda.

2.2 Aceleradores

La necesidad de procesar grandes cantidades de información es una característica creciente en los problemas computacionales modernos, lo que ha llevado a el uso de la IA en muchas áreas que la requieren para manejar estos datos. Para conseguir estos resultados los métodos de AP requieren de unidades de procesamiento de alto rendimientos como unidades de procesamiento gráfico (GPU, siglas del inglés) o unidades de procesamiento de tensores (TPU, siglas del inglés), además de consumir mucha energía y requerir mucho almacenamiento [3].

Debido a estas restricciones la implementación de aplicaciones de AP en entornos con pocos recursos es muy limitada, sin embargo, es importante el avance de la IA en dispositivos al borde, permitiendo realizar el procesamiento en el dispositivo. Para conseguir esto se plantea el reducir el tamaño de las redes neuronales, y desarrollar aceleradores de tareas específicas de AP [3].

2.2.1 Basados en FPGA

Un Field-programmable gate array (FPGA) es un chip computacional que puede programarse para implementar cualquier circuito digital. Se componen de arreglos de diferentes tipos de bloques programables, como lo son circuitos lógicos, entradas y salidas, entre otros. Estos bloques pueden interconectarse mediante interruptores programables. Para programar la funcionalidad que se busca en un FPGA se utilizan lenguajes de descripción de hardware como Verilog o usando síntesis de alto nivel [10].

Las GPUs han sido el tipo de aceleradores de hardware más común en las tareas de AP, sin embargo, por su consumo de energía es un reto su implementación en dispositivos que funcionan con baterías, por eso las tendencias en estos dispositivos apuntan al uso de aceleradores basados en FPGAs y ASICs (Application-Specific Integrated Circuit) por su alta eficiencia en consumo energético. Gracias a que las FPGAs soportan operaciones con pipeline, pueden llegar a tener incluso un mejor rendimiento en tareas de inferencia que las GPUs [20].

2.2.2 DMA

El direct memory access (DMA) es un mecanismo de hardware que permite a los componentes transferir sus datos de entrada y salida de forma directa, desde o hacia la memoria principal, sin necesidad de utilizar el procesador del sistema [13].

Actualmente los DMAEs (Direct Memory Access Engines) forman la base de las comunicaciones en muchos sistemas computacionales por su capacidad de mover datos con alto rendimiento mientras esconden la latencia de memoria y minimizan la carga del procesador permitiéndole a este enfocarse en otras tareas [8].

2.2.3 AXI-Lite

El Advanced eXtensible Interface (AXI) es un protocolo para el intercambio de información entre un maestro y un esclavo. AXI-Lite es una variación más simple para comunicación menos pesada, por ejemplo, comunicación entre registros de estado y de control [25].

Tanto AXI como AXI-Lite están compuestos por 5 canales, el canal de lectura de dirección, el de escritura de dirección, el de lectura de datos, el de escritura de datos y el de escritura de respuesta [25]. La información puede ir en ambas direcciones entre el maestro y el esclavo y el tamaño de la transferencia puede variar, en el caso de AXI-Lite permite transferir solo 1 dato por transacción [25].

2.3 Compiladores

Casi todas las aplicaciones que existen dependen de algún compilador para poder ser procesadas, un compilador es un programa computacional capaz de traducir otros programas computacionales de forma que estén listos para ser ejecutados [14].

La estructura típica de un compilador está conformada por un *frontend* para manejar el lenguaje fuente, un *backend* con el que manejar el lenguaje objetivo, y una o varias representaciones intermedias (IR, siglas del inglés) que conectan estas dos etapas, también se incluyen etapas de optimización que reescriben las IR de una forma que se pueda considerar más eficiente [14].

2.3.1 *frontend*

Es la etapa encargada de hacer el análisis léxico y sintáctico del código fuente para luego generar una IR [9]. Además de estos análisis el *frontend* puede tener otras responsabilidades, como utilizar información relevante sobre el lenguaje/dispositivo objetivo para sus optimizaciones, por ejemplo, al proporcionar información sobre el tipo de datos validos en el lenguaje objetivo el *frontend* puede aplicar las conversiones necesarias [27].

El *Frontend* suele estar dividido en tres etapas, el *scanner* se encarga de transformar el flujo de caracteres del código fuente en un flujo de palabras, analizando si estas palabras son válidas en el lenguaje. La siguiente etapa es el *parser* que utiliza un modelo de reglas de sintaxis del lenguaje para acomodar estas palabras. Por último, el elaborador, este es llamado por el *parser* para realizar algunas tareas en el código de entrada como generar la IR, disponer el almacenamiento, entre otros [14].

El *scanner* es la única etapa que interactúa con todo el código fuente [14], este lee el código como entrada y produce un conjunto de *tokens*, estos están conformados por una palabra valida del lenguaje y una categoría semántica para esta. La forma en que estas palabras son reconocidas es mediante un conjunto de reglas que rigen la estructura del lenguaje, estas reglas se representan comúnmente como expresiones regulares.

Las expresiones regulares son una forma de indicar en que formas se pueden combinar los caracteres. Utilizando autómatas finitos se pueden generar los *scanners* a partir de un conjunto de expresiones regulares, usualmente cada expresión del conjunto define una categoría semántica del lenguaje, y los autómatas finitos se suelen construir de forma que sus nodos finales sirvan para indicar a que categoría pertenece la palabra encontrada[14].

Para el *parser* las expresiones regulares ya no son suficiente, por lo que se utilizan gramáticas libres de contexto (CFG siglas del inglés), para especificar lenguajes libres de contexto, este se encarga de relacionar la secuencia de *tokens* generados por el *scanner* con las reglas y estructura definidas por la gramática libre de contexto, de forma que si algo en esta secuencia de *tokens* no se apega a estas reglas entonces existe un error[14].

Una CFG denotada como G es un conjunto de reglas que describen la forma en que

se pueden formar sentencias, todas las sentencias que se pueden formar basadas en G se llama el lenguaje definido por G y se denota como L(G) y el conjunto de todos los lenguajes para las diferentes CFGs es llamado el lenguaje libre de contexto. En las CFGs se tienen símbolos no terminales, que son variables que pueden derivarse (reemplazarse) por otros símbolos según las reglas de las CFGs, y símbolos terminales que son palabras del lenguaje [14].

Usualmente una CFG (como G) tiene 4 parámetros, un conjunto de símbolos terminales, un conjunto de símbolos no terminales, un símbolo designado como el símbolo objetivo o de inicio, que representa el conjunto de sentencias en L(G) y un conjunto de reglas para guiar la derivación o reescritura de los símbolos no terminales. Para derivar una sentencia se parte del símbolo de inicio, se toma de este un símbolo no terminal, se deriva con una de las reglas de la gramática y se reemplaza, este proceso se repite hasta tener solo símbolos terminales [14].

El *parser* suele dar como resultado un *parse tree*, que es una representación gráfica de la derivación del programa de entrada, esta suele ser una forma larga de representar el programa. Un árbol de sintaxis abstracta (AST siglas del inglés) es otra representación similar, este mantiene la estructura del *parse tree*, pero elimina los nodos que representan los símbolos no terminales [14].

Un grafo acíclico dirigido (DAG, siglas del inglés) es una reducción del AST que evita duplicación de expresiones [14], en un DAG un nodo puede tener múltiples nodos padre y subárboles que son idénticos son reutilizados, con esto se puede tener una representación más compacta que con un AST. Al utilizar DAGs el compilador debe probar que el valor de una variable no vaya a cambiar entre usos para mantener la consistencia.

2.3.2 Representación Intermedia

Generar la IR es la última tarea manejada por el *frontend*. Usualmente estas consisten en una representación del código acompañada de algunas estructuras de datos con información adicional. La forma de las IRs no es única, y depende de las características de los lenguajes fuente y objetivo, entre algunas de las posibles formas están las IR en forma de grafo y las IR secuenciales que tienen una forma similar a un código ensamblador [14].

En LLVM [17] la representación intermedia es orientada al dispositivo objetivo, por lo que es similar a un código ensamblador. Está construida por bloques básicos e instrucciones, un bloque básico es una secuencia de operaciones que no contienen operaciones de control de flujo (branch, jump etc) y las instrucciones son representaciones abstractas de instrucciones computacionales (lógicas, aritméticas, de memoria etc). Además, la LLVM IR es independiente del *frontend* y el *backend* reduciendo el trabajo para los desarrolladores.

2.3.3 Optimizador

Cuando el *frontend* genera la IR este maneja las sentencias una a la vez y en orden, lo que genera una IR capaz de funcionar en cualquier contexto que el compilador genere, sin embargo el código va a ser ejecutado luego en un contexto más predecible, el optimizador entonces puede analizar la IR utilizar información del contexto y con este conocimiento reescribir el código de una forma que cumpla el mismo propósito pero sea más eficiente, para esto se suelen utilizar dos etapas [14].

La etapa de análisis determina donde el compilador puede aplicar transformaciones de forma segura y eficiente, algunos análisis que aplican los compiladores son, el análisis del flujo de datos donde se razona en tiempo de compilación sobre el flujo de valores en tiempo de ejecución y el análisis de dependencias que razona sobre las dependencias en las referencias de memoria [14].

La etapa de transformación utiliza los resultados del análisis para reescribir el código en una forma más eficiente. Para mejorar requerimientos de tiempo/espacio del código ejecutable se utilizan las transformaciones Myriad o para mejorar el tiempo de ejecución se usan tecnificas como el *loop-invariant code motion*. Esta etapa depende de que el análisis sea capaz de soportar las transformaciones que se quieran implementar [14].

2.3.4 *Backend*

Esta etapa es la encargada de transformar la IR generada por el *frontend*, que es independiente del lenguaje objetivo y el lenguaje fuente, en un programa de ensamblador que tenga un comportamiento equivalente [16].

Para conseguir generar el código el *backend* busca resolver tres problemas principales, primero la selección de instrucciones, proceso en que se convierten las operaciones de la IR en una o varias operaciones que sean equivalentes en el ISA objetivo. Luego la calendarización de instrucciones, este proceso busca seleccionar el orden para ejecutar las operaciones. Por último, la asignación de registros, proceso en que se decide que valores deben estar en que registros en cada etapa del código [14].

En compiladores basados en LLVM el *backend* utiliza una librería del procesador, en esta se almacena información sobre instrucciones, registros y demás características sobre el hardware que son relevantes al generar el código objetivo [6].

Para generar un programa desde la representación intermedia (como puede ser un DAG o un AST) el compilador debe transformar cada pieza de la IR en un equivalente para el ISA del *hardware* objetivo. Este proceso puede requerir elaborar detalles que están escondidos en la IR o combinar varias operaciones de la IR en una única instrucción en lenguaje maquina [14].

Una forma de hacer la selección de instrucciones es mediante herramientas de *tree-pattern matching*, para esto tanto la IR como el ISA del *hardware* objetivo deben representarse

como arboles (AST de bajo nivel, DAG). Teniendo un árbol (un AST, por ejemplo) para el código y un conjunto de árboles para operaciones del ISA, se hace una reconstrucción del AST con estos árboles de operaciones, donde cada pieza es un par de datos, primero un nodo del AST (x) y luego la raíz de un árbol de operación (y).

Una reconstrucción es un conjunto de estas piezas que cumple las siguientes condiciones [14].

- Primero hay una pieza para cada nodo del AST.
- La raíz de cada árbol de operación sobrepone a una hoja en algún otro árbol de operación, a no ser que se sobreponga a la raíz del AST.
- Donde dos árboles de operaciones se sobreponen, son compatibles, en clase de almacenamiento y tipo de valor.
- La sobreposición de cualquier par de árboles de operaciones se da en un único nodo.

La existencia de uno de estos pares que cumplan las condiciones indica que el subárbol del AST con raíz en x puede ser implementado por la operación con raíz en y. El compilador debe verificar que el AST y cada subárbol de este pueda implementarse por un conjunto de árboles de operaciones. Dada una reconstrucción de un AST con estas piezas el compilador puede generar código ensamblador haciendo un recorrido de arriba a abajo sobre el AST.

El tiempo que un conjunto de operaciones tarda en ejecutarse depende mucho del orden en que se ejecutan [14], la etapa de calendarización de instrucciones reordena las operaciones reduciendo el tiempo de ejecución, existe un conjunto limitado de formas legales de ordenar las operaciones de forma que se preserve el flujo de valores respecto al código original.

Para analizar como el flujo de los valores restringe el orden de operación, el compilador genera un grafo de dependencias, el calendarizador reordena las operaciones dentro de las restricciones dadas por el grafo de forma que se aprovechen al máximo los recursos del procesador y se escondan las latencias. Para eso se deben considerar características de la arquitectura como ejecución con *pipeline*, operaciones con latencias variables (por ejemplo, accesos a memoria) y el tener múltiples unidades funcionales [14].

Los registros suelen ser una de las características más importantes de un procesador [14], al ser de más rápido acceso que la memoria tienen un papel importante en el tiempo de ejecución, la etapa de asignación de registros decide que valores van a estar en registros y que valores van a estar en memoria en cada parte del código. Para esto el asignador usualmente agrega operaciones de carga y guardado para mover valores entre registros y memoria.

El código que la etapa de asignación recibe utiliza un numero arbitrario de registros, así que esta etapa de transformar el código a uno equivalente que se adapte a la cantidad finita de registros que tiene la arquitectura objetivo.

Para la asignación de registros la solución más común es el representar los conflictos entre registros mediante un grafo de interferencia y luego colorear el grafo para encontrar las asignaciones. La estructura de este método se divide en las siguientes etapas [14].

- El asignador encuentra todos los rangos de vida (LR, una definición de un valor y sus usos posteriores), y reescribe el código con un nombre único para cada rango.
- Se crea el grafo de interferencia, cada rango de vida es un nodo y se agregan conexiones entre un rango de vida (LR_i) y cada otro rango (LR_j) que está activo en una operación que define (LR_i), a no ser que esta sea una copia.
- Se revisa cada operación que es una copia (LR_i=¿LR_j) si cada LR parte de la operación no interfiere entonces estos se combinan, se remueve la copia y se actualiza el grafo. Este proceso se repite sobre los nuevos grafos hasta que ya no se pueda combinar nada.
- Se calcula para cada LR el costo estimado de mover a memoria los valores en el LR.
- Se colorea el grafo, primero se simplifica el grafo para construir un orden para el coloreado y luego se reconstruye el grafo asignando colores mientras inserta cada nodo. Si ningún nodo queda sin color entonces se reescribe el código y termina el método, si algún nodo no está coloreado se pasa al último paso.
- Para cada nodo sin colorear, se inserta una operación para pasarlo a memoria después de cada definición y una para restaurarlo de memoria antes de cada uso.

2.3.5 Transpiladores

Son un tipo especifico de compiladores. Estos traducen código de un lenguaje de programación a otro lenguaje de programación, con el reto de mantener la semántica entre los lenguajes de origen y objetivo y trabajando con la sintaxis de ambos [15].

Estos han sido utilizados para transformar código desde 1978 cuando Intel propone un transpilador para convertir programas de 8 bits a programas equivalentes de 16 bits, en 1981 se propone un transpilador con el objetivo de convertir código ensamblador 8080 a código 8086, y actualmente existen proyectos para transpilar de Python a Julia, de C a Rust y otros muchos [7]. Con estos ejemplos se puede recalcar una de las características que definen a los transpiladores, el proceso se hace entre dos lenguajes con niveles de abstracción similares.

Capítulo 3

Compilador para el acelerador de aprendizaje profundo ArPaReUV-AP

En esta sección se va a describir la metodología seguida para el desarrollo del proyecto, comentando sobre las herramientas utilizadas y el porqué se escogieron, además de las actividades realizadas y con que entregables y objetivos se relacionan.

Además, se va a explicar paso a paso la implementación de la solución con los detalles suficientes para que esta sea fácil de entender y de replicar si se requiere extender el ISA del acelerador con nuevas instrucciones.

3.1 Metodología

Para el desarrollo del proyecto se decide seguir una metodología incremental, es decir, que se planea dividir el proyecto en etapas o incrementos, de forma que se hagan avances funcionales, para luego seguir trabajando sobre estos hasta obtener el resultado final.

En la Figura 1 se pueden observar las etapas de la metodología. Estas se definen de la siguiente manera.

1. Investigación: En esta etapa se busca información sobre diferentes métodos y herramientas de compilación modernas, especialmente enfocados a arquitecturas SIMD y aceleradores de aprendizaje profundo.

2. Análisis: Se consideran las diferentes herramientas encontradas durante la investigación, se analizan sus enfoques y sus ventajas para seleccionar la que mejor se adapta al proyecto.

3. Diseño: Luego de seleccionar la herramienta, LLVM en este caso, se revisa su estructura para poder entender su funcionamiento y que modificaciones deben realizarse

con el fin de agregar nuevas instrucciones al compilador.

4. Implementación: Se agregan los cambios en los diferentes módulos de LLVM implicados en la compilación de código C a ensamblador, se implementan las clases necesarias y se implementan las diferentes formas de las instrucciones en las distintas etapas de compilación. Además, se implementa código C para verificar el compilador.

5. Verificación: Se compila el código C de prueba para verificar que las instrucciones agregadas se representan de la forma esperada en cada etapa.

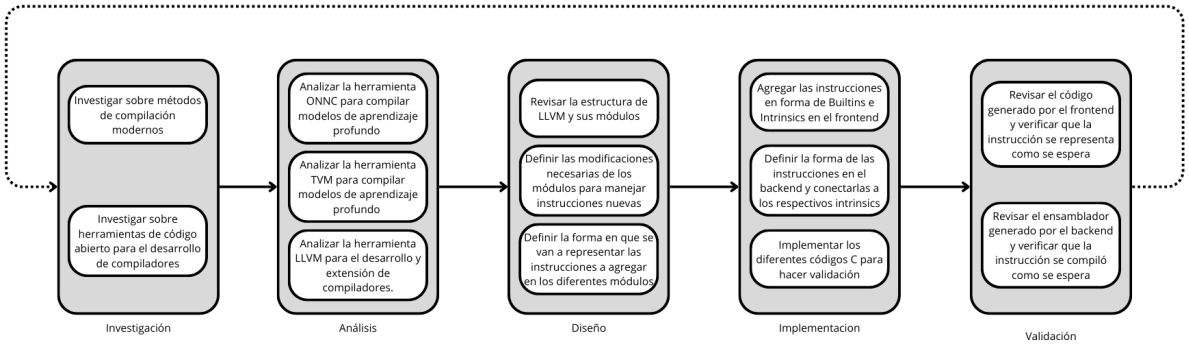

Fig. 1: Metodología seguida para el desarrollo del proyecto

3.1.1 Selección de herramientas

El desarrollar un compilador desde cero es un proceso sumamente complejo y tardado, por eso para la solución se decidió utilizar como base el proyecto de LLVM y Clang. LLVM es un compilador de código abierto, que tiene una estructura modular con el fin de permitir la modificación y extensión del mismo por otros desarrolladores, por otro lado, Clang es un proyecto relacionado a LLVM que funciona como frontend del compilador para los lenguajes de alto nivel C/C++ y que también sigue una estructura modular con el fin de poder ser extendido.

Estas herramientas se seleccionaron por múltiples razones, primero al ser alternativas de código abierto cumplen con las limitaciones de presupuesto que se tienen, además, también se puede tener acceso completo al código fuente para poder usarlo de referencia a la hora de extender los módulos. Otra razón es que son herramientas que están teniendo mucho crecimiento y son utilizadas en el desarrollo de muchos compiladores modernos por empresas tecnológicas muy importantes.

Se consideraron otras opciones como ONNC y Apache TVM, pero estas están más enfocadas a la compilación específicamente de modelos de aprendizaje profundo así que no se adaptaban bien a lo que se quería con este proyecto.

Se utiliza un sistema operativo basado en UNIX ya que una gran parte de LLVM y Clang es código C++ y este suele tener más soporte y mejor compatibilidad con sistemas UNIX, de igual forma se puede utilizar en otros sistemas operativos como Windows pero puede ser más propenso a problemas. Durante el desarrollo del proyecto se utilizó Ubuntu 22.04.5 LTS.

En la Figura 2 se puede observar un diagrama de alto nivel que muestra las herramientas utilizadas para el desarrollo y su relación.

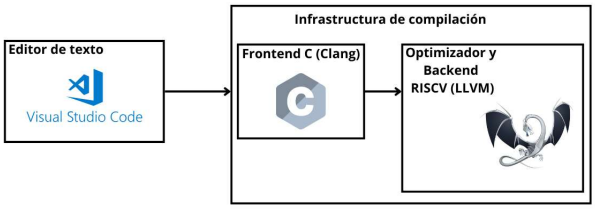

Fig. 2: Diagrama de alto nivel con las herramientas utilizadas

3.2 Desarrollo de la solución

En esta sección se va a abarcar la estructura de Clang y LLVM con el fin de detallar las modificaciones necesarias a estos para agregar instrucciones nuevas, además, se van a especificar las representaciones de estas instrucciones en cada etapa, desde el tipo de datos y la forma en que se definen hasta otras características que el compilador utiliza para poder compilar de forma eficiente y correcta cada instrucción, tomando las consideraciones necesarias según estas características.

En la Figura 3, se observa la arquitectura de LLVM utilizando Clang como frontend de C. Primero el código fuente escrito en C pasa por Clang, en Clang se realiza el análisis léxico, sintáctico y semántico, para dividir el código en C en tokens, revisar su estructura y validez y generar un AST con estos, para luego hacer la generación de código LLVM IR a partir del AST. Para estos se tiene información definida del tipo de datos validos en C y el tipo de datos válidos para LLVM IR.

Este código LLVM IR generado es un archivo de extensión .ll (LLVM IR es un lenguaje de programación independiente del lenguaje fuente o el lenguaje objetivo), la etapa del middlend toma como entrada un archivo .ll y le aplica distintas formas de optimización, esta se puede controlar a la hora de compilar para que se intente hacer más o menos optimizaciones. La salida es otro archivo .ll que está que fue reescrito por el optimizador.

El código optimizado es la entrada al *backend* de LLVM, el cual es dependiente de la arquitectura objetivo (RiscV Vector en este caso), en esta etapa se tiene información relevante de la arquitectura como tipos de registros y definición de estos, el ISA con los formatos y definición de las instrucciones, los tipos de datos validos e información sobre cómo se maneja la memoria (direccionamiento, alineamiento etc). Esta etapa genera un

DAG independiente a la arquitectura a partir del LLVM IR, luego un DAG dependiente a la arquitectura y por último el código ensamblador.

Por último, el archivo ensamblado .o debe pasar por la etapa de enlazado para así generar un archivo ejecutable que pueda ser utilizado por el acelerador.

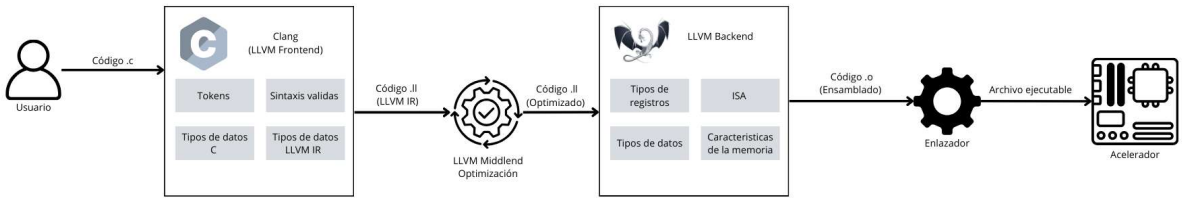

Fig. 3: Diagrama de arquitectura de LLVM

Al utilizar LLVM y Clang se tienen diferentes funcionalidades que pueden ser de utilidad, especialmente al trabajar en alguna expansión o modificación para poder seguir los resultados en diferentes etapas. En la Figura 4, se pueden observar las funcionalidades, el poder visualizar el código LLVM IR permite verificar la generación de código hecha por Clang para revisar los tipos de datos y atributos, también se pueden verificar solo las etapas de análisis del frontend. Si se instalan las herramientas con la opción de Debug se pueden verificar también los DAGs generados por el *backend*. Y además, generar el archivo ensamblado .o para poder enlazar.

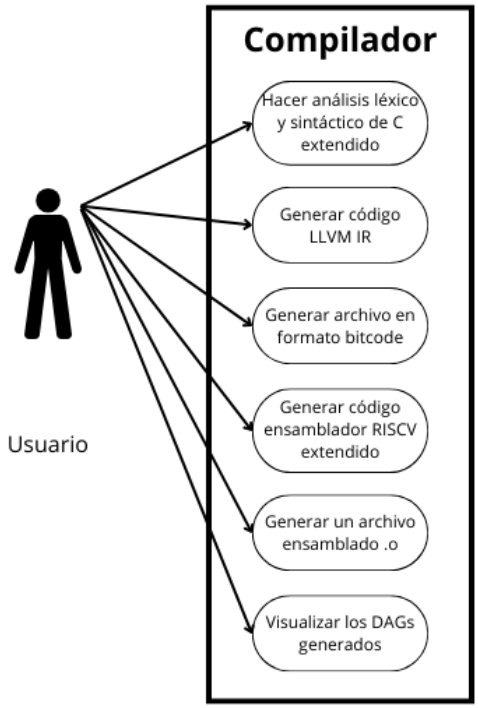

Fig. 4: Diagrama de funcionalidades por parte del usuario

3.3 FrontEnd

El frontend está definido totalmente en Clang, en esta etapa las instrucciones que se agregaron se representan primero como un *Builtin*, un tipo de función que el compilador conoce de forma intrínseca, es decir que no es parte de una biblioteca externa, y que el compilador sabe manejar directamente.

Esta es una forma muy común de proveer al programador en un lenguaje de alto nivel una forma más directa de interactuar con el hardware ya que usualmente estos *Builtins* van a ser representados luego por una o unas pocas instrucciones en el código ensamblador. Para este proyecto se van a relacionar uno a uno los *Builtins* y las instrucciones en ensamblador ya que se busca que estos representan una operación especifica que pueda hacer el acelerador.

A la hora de definir los *Builtins* de Clang estos pueden ser dependientes o independientes de la arquitectura objetivo, esto es importante para seleccionar la clase del *Builtin* y los tipos de datos ya que distintas arquitecturas populares ya tienen clases y tipos de datos definidos, además, tienen también métodos diferentes de disminuir el *Builtin* y generar código LLVM IR.

Por esto para agregar el nuevo *Builtin* se añadió una clase en el archivo riscv_vector_common.td, que es donde se definen las clases existentes para RVV (RiscV Vector extension), la clase definida se puede observar en la Figura 5, esta clase extiende RVVOutOp1Op2Builtin con el fin de aprovechar la generación de código automática que tienen ya las clases definidas para RVV y aprovechar también los tipos de datos vectoriales ya definidos. Luego los *Builtins* se definen en riscv_vector.td extendiendo la clase creada [1].

```
class AccOutOp0Op1Builtin<string intrinsic_name, string suffix, string prototype, string type_range>
    : RVVOutOp0Op1Builtin<suffix, prototype, type_range> {
  let HasMasked = false;
  let HasMaskedOffOperand = false;
  let HasVL = false;
  let MaskedPolicyScheme = NonePolicy;
  let HasTailPolicy = false;
  let HasMaskPolicy = false;
  let Log2LMUL = [0];
  let IRName = intrinsic_name;
}
```

Fig. 5: Definición de la clase para Builtins del acelerador

En la Tabla 2 se explican los parámetros de entrada que se utilizan a la hora de definir un nuevo *Builtin* de la clase añadida, con el fin de poder explicar y ejemplificar bien su uso.

Además, en la Tabla 3 se explican los atributos definidos para la clase, en general estos parámetros representan información que utilizan las instrucciones de RVV por la forma en que se define el ISA, pero ya que las instrucciones añadidas van a basarse en otro ISA se define una forma más simplificada para la clase de *Builtin* añadida.

[1] Ambos archivos se encuentran en llvm-project/clang/include/clang/Basic

Parámetro	Descripción	Ejemplo y explicación
Intrinsic name	Este es un parámetro de tipo *string* que define el nombre del *Intrinsic* de LLVM IR al que se debe transformar este *Builtin*, esto se hace para que el generador de código de Clang pueda hacer esta conexión de forma automática.	`"mul_vec"`: Con esta definición Clang va a buscar un *Intrinsic* que se haya definido con el nombre `"mul_vec"` además de los prefijos que se utilizan al definir *Intrinsics*
Type range	Es un *string* conformado por varias letras que definen los tipos de datos para los que se va a generar un *Builtin*, es decir si se define más de un tipo de dato se van a generar varios *Builtins* en una misma definición, para cada tipo de dato	"cif": Aqui la letra "c" indica el tipo de dato entero representado con 8bits (int8_t), la "i" el tipo de dato entero de 32bits (int32_t) y la "f" flotante de 32bits (float32_t).
Prototype	Es un *string* con diferentes transformadores de tipos, se utiliza junto al *Type range* para definir los parámetros del *Builtin* en C, siendo primero el tipo de retorno y luego el tipo de las entradas	"vvUe": Considerando un *type range* "i" este ejemplo representa que la salida (v) debe ser un vector de enteros de 32bits, el primer parámetro de entrada (v) debe ser otro vector de enteros de 32bits y el segundo parámetro (e) debe ser un entero de 32bits sin signo (U).
Sufix	Es un *string* similar al de *prototype*, pero puede ser opcional, este utiliza igualmente modificadores de tipo para definirse y se utiliza para acompañar el nombre del *Builtin* con el tipo de datos, luego de un guion bajo.	"vv": Con esta definición y un *type range* "i", el nombre del *Builtin* va a estar acompañado por un `_i32m1_i32m1`.

Tabla 2: Parámetros de la clase para Builtins del acelerador

Además de definir las instrucciones como un *Builtin*, se definieron también en forma de *Intrinsics*, estas son instrucciones que se van a utilizar durante la generación de código para representar en el código LLVM IR los *Builtins* definidos. Estas le indican al compilador que estas instrucciones no forman parte del estándar de C y con sus atributos le indican la forma en que se deben manejar para optimización y que luego en el *backend* se va a definir la forma en que van a ser reducidas estas *Intrinsics*.

Para definir los nuevos *Intrinsics* se modificó el archivo IntrinsicsRISCV.td[2], se definieron algunas clases para las instrucciones que compartían tipos de datos tanto de las entradas como de la salida y para las instrucciones que no tenían tipos de datos iguales a otras se

[2]El archivo se encuentra en llvm-project/llvm/include/llvm/IR

Atributo	Descripción
HasMasked	Indica si existe una forma con mascara para el *Builtin*. La máscara es una propiedad de RVV que ayuda a operar solo ciertos valores de los vectores.
HasMasked OffOperand	Indica si el *Builtin* espera un operador especial relacionado a las operaciones con mascara de RVV.
HasVL	Indica si el *Builtin* recibe una definición de VL, variable de RVV que indica el tamaño de los vectores, en este caso se indica que no lo toma en cuenta, pero puede ser una buena opción a considerar para manejar el tamaño de vectores variables en el ISA del acelerador.
. MaskedPolicyScheme, HasTailPolicy y HasMaskPolicy	Estos atributos indican las formas en que se van a manejar algunas operaciones que utilizan características de RVV, que no son importantes para estos *Builtins*.
Log2LMUL	Este parámetro indica el valor de LMUL, este indica la multiplicidad para las operaciones una característica de RVV que permite operar sobre varios vectores a la vez, se define como 0 que representa una multiplicidad estándar de 1.

Tabla 3: Especificación de los atributos de la clase para el Builtin

tuvo que hacer una definición de una en una. En la Figura 6 se muestra una de las clases definidas para uno de los tipos de *Intrinsics*.

```
class AccVecVecVecIntrinsics : DefaultAttrsIntrinsic<[llvm_anyvector_ty]
                                                     [LLVMMatchType<0>,
                                                      LLVMMatchType<0>],
                                                     [IntrNoMem,
                                                      IntrWillReturn,
                                                      IntrSpeculatable]>,
                               RISCVVIntrinsic;
```

Fig. 6: Clase definida para un tipo de Intrinsic

En este caso la clase se llama AccVecVecVecIntrinsics, para un tipo de *Intrinsic* que recibe dos vectores y retorna un vector. El primer atributo es una lista con los tipos de datos de las salidas, en este caso es solo un valor, llvm_anyvector_ty, que significa que la salida será un vector de cualquier tipo, el segundo parámetro es otra lista con los tipos de las entradas, en este caso ambas se definen con $LLVMMatchType < 0 >$, que significan que ambas deben tener el mismo tipo que la salida. El ultimo parámetro es otra lista donde se definen algunos atributos para del *Intrinsic*, en este caso se indica que la instrucción no lee ni escribe en memoria, que va a retornar algo y que el optimizador puede especular con ella respectivamente.

La clase además, se define para extender primero a DefaultAttrsIntrinsic, una clase que se utiliza para especificar ciertos valores estandar para los atributos de los *Intrinsics* y

que a su vez extiende la clase base de *Intrinsics*. Luego se define para que extienda a RISCVVIntrinsic con el fin que Clang pueda hacer de forma automática la vinculación con el *Builtin* y se aproveche la generación de código definida para RVV.

Para conseguir todo el proceso Clang pasa por diferentes componentes encargados de ir descomponiendo el código C y representándolo en diferentes formas hasta generar el LLVM IR. En la Figura 7 se puede ver el diagrama de los componentes de Clang.

Primero al invocar a clang para compilar un código C este pasa por el lexer, el cual descompone las cadenas de caracteres en tokens que sean válidos para Clang, esta secuencia es utilizada por el preprocesor para aplicar ciertas transformaciones a los tokens de ser necesario, los tokens procesados son tomados por el parser el cual se encarga de generar un árbol con nodos sintácticos, es decir que analiza solamente la estructura pero no el sentido como tipos de datos o cantidad de parámetros, el parser utiliza la definición de los *Builtins* para verificar su estructura.

El encargado de generar el AST es el analizador semántico, el que verifica del árbol generado por el parser que se estén utilizando los tipos de datos correctos y cantidad de parámetros correctos, para esto también utiliza la definición del *Builtin*, además utiliza las definiciones de los *Builtins* e *Intrinsics* para verificar si el *Builtin* corresponde a un *Intrinsic* definido. Luego de la validación se utiliza el AST para generar el LLVM IR, en esta etapa también se utiliza la definición de los *Builtin* e *Intrinsics* para lograr hacer la transformación de los nodos del AST correspondientes al *Builtin* en un *Intrinsic* que LLVM pueda reconocer.

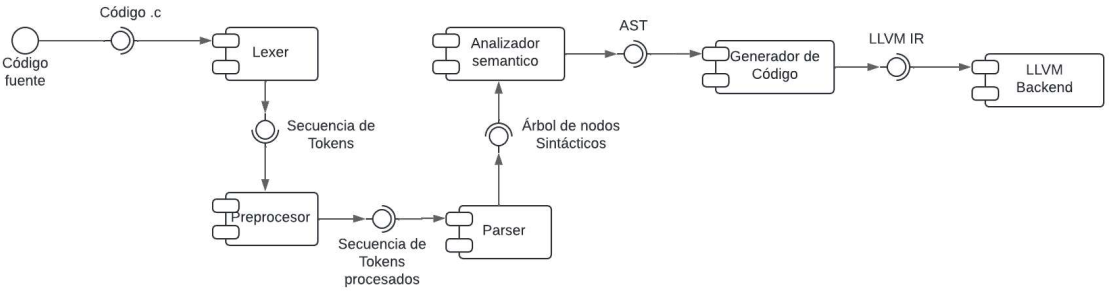

Fig. 7: Diagrama de componentes del Frontend

3.4 *Backend*

El *backend* de LLVM es la etapa donde se transforma la LLVM IR a código de ensamblador de la arquitectura objetivo. Para eso se debe definir la forma en que el *backend* va a vincular los *Intrinsics* con instrucciones reales.

Para efectos de este proyecto la definición se hizo uno a uno en todo el proceso, es decir un *Builtin* se transforma como un único *Intrinsic* para LLVM IR que a su vez se transforma

en una única instrucción en ensamblador. Al igual que para el frontend se aprovechan algunas definiciones de RVV para hacer la reducción y definición de las instrucciones.

Debido a que la relación es directa y que ninguna de las instrucciones a agregar interactúa con memoria o tiene efectos secundarios se puede hacer una reducción estándar, es decir no se tuvieron que agregar casos específicos para las instrucciones. Sin embargo, para posibles expansiones futuras del ISA se van a explicar más adelante algunas consideraciones para instrucciones que si requieran un proceso personalizado.

Para poder transformar el *Intrinsic* lo primero que se debe hacer es definir la forma de la instrucción a la que se va a reducir, para esto se creó el archivo RISCVInstrInfoAcc.td y se incluyó en RISCVInstrInfoV.td, que es donde se definen las instrucciones para RVV. La definición de una de las instrucciones se puede ver en la Figura 8. En este caso se utiliza la clase más básica *Instruction* ya que el formato para el ISA del acelerador aún no se ha establecido, para el futuro del proyecto se podría definir una clase para el ISA que extienda Instruction o alguna otra clase en el archivo RISCVInstFormats.td [3].

```
def MULVECT : Instruction {
    bits<32> Inst;
    bits<32> SoftFail = 0;
    bits<5> rs2;
    bits<5> rs1;
    bits<5> rd;
    let Namespace = "RISCV";
    let Size = 4;
    let hasSideEffects = 0;
    let mayLoad = 0;
    let mayStore = 0;
    let Inst{31-25} = 0b0000100;
    let Inst{24-20} = rs2;
    let Inst{19-15} = rs1;
    let Inst{14-12} = 0b111;
    let Inst{11-7} = rd;
    let Inst{6-0} = 0b0101011;
    dag OutOperandList = (outs VR:$rd);
    dag InOperandList = (ins VR:$rs1, VR:$rs2);
    let AsmString = "mul_vect\t$rd, $rs1, $rs2";
}
def : Pat<(nxv2i32 (int_riscv_mul_vec (nxv2i32 VR:$rs1), (nxv2i32 VR:$rs2))),
          (MULVECT VR:$rs1, VR:$rs2)>;
```

Fig. 8: Definición de la instrucción vec mul para el backend

Para explicar mejor la Figura 8 se van a definir sus atributos.

1. Primero se definen características de la instrucción, en este caso bits¡32¿Inst define el tamaño de la instrucción, SoftFail = 0 es un atributo para el desensamblador que no es relevante para todas las arquitecturas pero que se debe definir para que funcione correctamente la instrucción, luego se define el tamaño en bits de los registros destino y fuente, por ultimo Size define el tamaño en bytes de la instrucción y Namespace vincula la instrucción con una arquitectura, en este caso Riscv5.

[3]Estos archivos se encuentran en llvm-project/llvm/lib/Target/RISCV

2. HasSideEffects indica si la instrucción va a tener efectos secundarios que no van a ser capturados por operandos de la instrucción o banderas. En este caso 0 indica que no tiene efectos secundarios.

3. mayLoad/mayStore se utilizan para instrucciones que vayan a leer o escribir en memoria respectivamente, un 0 significa que la instrucción no interactúa con la memoria.

4. InstA-B es una forma para definir el valor de los bits de la instrucción desde el bit A hasta el bit B. De esta forma se deben definir los bits que corresponden a los registros de destino y fuente definidos al inicio de la instrucción y que la definición sea coherente con el tamaño indicado al inicio.

5. OutOperandList/InOperandList estos son de tipo DAG y se usan para indicar las entradas y salidas de la instrucción referenciando los registros fuente y registros de salida. En este caso outs VR:$rd indica que hay un único registro destino y que este es de la clase VR (Vector Register), en el caso de las entradas se define que hay dos registros de donde viene la información y ambos son de tipo VR.

6. AsmString define la forma en que se va a escribir la instrucción en los archivos de código ensamblador (.s). en este caso primero se indica el nombre a mostrar para la instrucción y luego los registros que participan (en el archivo .s no se muestra como $rd pero como el registro físico de la arquitectura que se utiliza, por ejemplo, v6).

Además de registros de tipo VR, el *backend* de RISCV tiene otros tipos como GPR (Registros de propósito general), FPR (Registros para punto flotante), ademas se puede agregar una clase nueva de registros de ser necesario, definiendola en el archivo RISCVRegisterInfo.td que extienda de alguna clase de registro RiscV o de la clase Register base. Ademas, se deben definir los registros existentes para esta nueva clase.

Luego de haber definido la instrucción se debe hacer la conexión entre el *Intrinsic* y la instrucción, para eso se definió un patrón (Pattern) junto a la definición de la instrucción, estos patrones se utilizan durante la selección de instrucciones en la etapa de generación de código para poder vincular un nodo del DAG generado a partir del LLVM IR con un instrucción máquina de la arquitectura objetivo.

En la Figura 8 se puede observar al final, como se definió el patrón para la instrucción vec mul, este recibe dos parámetros primero el *Intrinsic* y luego la instrucción. En ambos casos se definen las entradas en forma de dag con el tipo de registros. En el caso del *Intrinsic* también se debió definir los tipos de datos tanto del resultado de la operación como de las entradas, en este caso los tres son del tipo nxv2i32, que es uno de los tipos de datos vectoriales definidos para RVV que representa vectores escalables de enteros de 32bits para una multiplicidad de 1.

En la Figura 9 se puede observar un diagrama de componentes para la primera etapa del *backend*, esta es la etapa principal para la generación de código, es la encargada de recibir la LLVM IR y convertirla a una Machine IR (MIR).

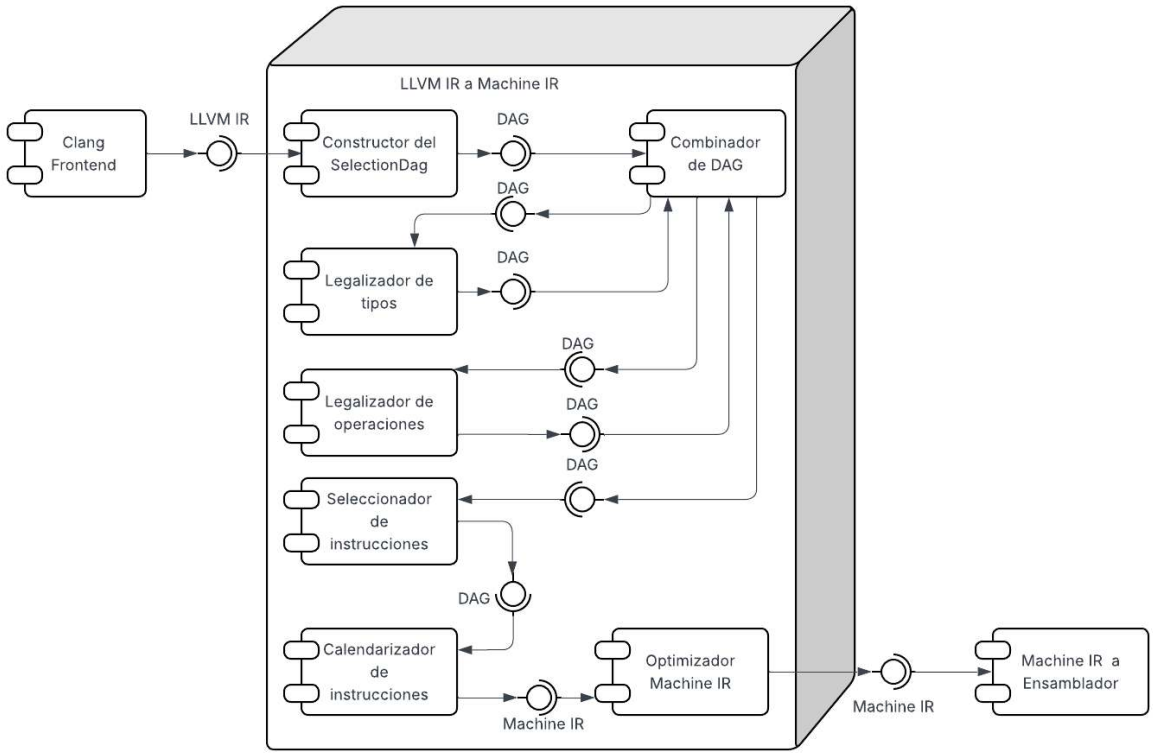

Fig. 9: Diagrama de componentes del *backend* para la generación de código MIR

Para iniciar el proceso el Constructor del SelectionDAG genera un DAG para cada bloque básico del código LLVM IR, estos DAGs pasan al combinador, que es un módulo que realiza optimizaciones sobre los DAGs, luego estos pasan al legalizador de tipos, ya que LLVM IR acepta muchos más tipos de los que maneja la arquitectura objetivo este módulo se encarga de convertir los tipos no soportados por la arquitectura en tipos soportados, el resultado se optimiza de nuevo, y se continua a el legalizador de operaciones, similar que con los tipos LLVM IR soporta muchas más operaciones de las que soporta la arquitectura objetivo, por lo que este módulo debe convertir estas operaciones no soportadas en operaciones validas y se optimiza una última vez, el seleccionador de instrucciones es ahora el encargado de convertir nodos genéricos del DAG por nodos específicos para la arquitectura, el calendarizador de instrucciones se encarga de tomar estos nodos acomodados en DAGs y generar una secuencia de nodos lineal o código MIR. Por último, esta MIR pasa por una fase de optimización.

El diagrama de componentes de la segunda etapa se puede observar en la figura 10 esta etapa comienza recibiendo la MIR donde el acomodador de registros se encarga de reemplazar los registros virtuales de la MIR por registros físicos pertenecientes a la arquitectura, luego se hace una optimización sobre la MIR con los registros definidos, seguido a esto se agrega el prólogo y el epilogo, segmentos de código encargados de asegurar un manejo correcto de la pila, luego el optimizador de mirilla analiza pequeñas secciones de código en busca de hacer reemplazos que mejoren la eficiencia, para por ultimo pasar al

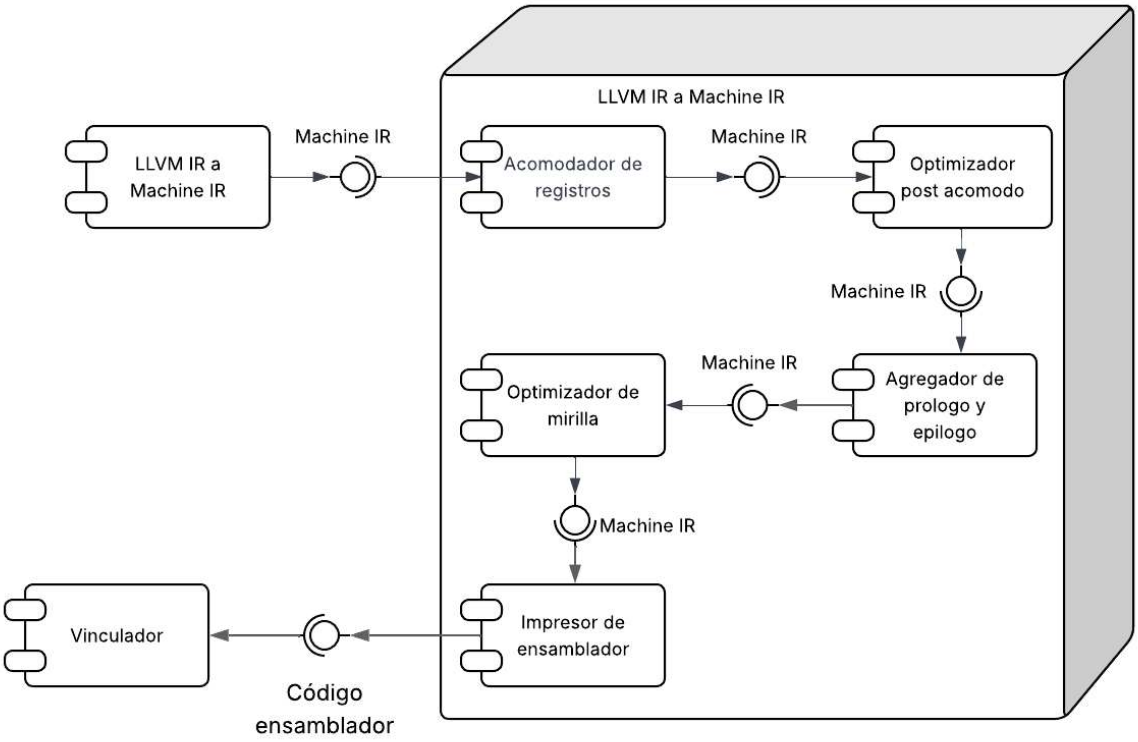

Fig. 10: Diagrama de componentes del backend para la generación de código ensamblador

impresor de ensamblador encargado de generar un código ensamblador de la arquitectura objetivo.

Para las instrucciones implementadas no era necesario definir una forma de reducirlas personalizada ya que todas eran operaciones aritméticas relacionadas uno a uno entre C y RISCV. Pero ahora se van a especificar algunas consideraciones para instrucciones que no cumplan estos requisitos.

LLVM permite el uso de Pseudoinstrucciones, estas son similares a las instrucciones definidas en el sentido de que reciben un dag para especificar las salidas y para especificar las entradas, además de otra información, pero estas no se definen en forma de bitcode ya que lo que distingue a este tipo de instrucciones es que no representan una instrucción de ensamblador, se utilizan para etapas intermedias y LLVM entiende que luego se va a especificar de forma explícita lo que debe hacerse con estas instrucciones, por ejemplo según algún parámetro convertirse a cierta instrucción específica para cierto tipo de dato, o bien ser reducida a varias instrucciones de ensamblador que se deban ejecutar en secuencia.

Estas son de gran utilidad para poder definir *Builtins* más complejos que no representan directamente una instrucción permitida en la arquitectura pero que luego LLVM va a saber cómo representarlos utilizando instrucciones de máquina, con esto se puede facilitar el trabajo a la hora de desarrollar en C, por ejemplo, al hacer análisis del código si se encuentran patrones de instrucciones que se repitan se pude generar un *Builtin* que agrupe

estas instrucciones y facilite la programación.

En el caso de ensamblador otra forma en que se pueden optimizar instrucciones más complejas es definir un patrón compuesto, en el que el intrínseco se relacione con varias instrucciones maquina anidadas para que se pueda generar código más eficiente por ejemplo casos donde el resultado de una operación va a ser la entrada de otra.

Si se quiere hacer una reducción personalizada se debe modificar el archivo RISCVISelLowering.cpp [4], en este primero debemos definir la operación mediante la función setOperationAction(), este recibe primero el tipo de nodo, luego el tipo de dato (*Machine value type*) y luego la acción (*Custom* para una reducción personalizada), esta función le indica al compilador que si encuentra un nodo del tipo indicado con los tipos de datos indicados debe buscar la definición para una reducción personalizada.

Luego en este mismo archivo se debe buscar la función RISCVTargetLowering::LowerOperation, esta es la encargada de legalizar las operaciones, es decir tomar operaciones que son válidas para LLVM IR y convertirlas en operaciones que sean válidas para la arquitectura objetivo. Para esto se debe conocer el identificador de la operación y generar un caso dentro de la función para este identificador. En el caso se debe entonces definir la forma en que el compilador debe construir el nuevo nodo del DAG para que sea válido.

Los pasos anteriores definen una forma personalizada para la etapa de legalización de operaciones de la reducción, pero también puede ser necesario personalizar la forma en que se hace la reducción en la etapa de selección de instrucciones. Para esto se debe modificar la función RISCVDAGToDAGISel::Select en el archivo RISCVDAGToDAGISel.cpp [5], igualmente se debe definir un caso para el tipo de nodo de la función que queremos reducir y ahí definir la forma en que queremos que se remplace el nodo genérico del DAG por un nodo maquina relacionado con la arquitectura objetivo.

3.5 Integración

En esta sección se va a presentar el compilador de forma integrada, comenzado por el diagrama que se puede observar en la Figura 11, este es un diagrama de clases general y sintetizado del compilador, en este se muestran solo algunas de las clases claves para la compilación y algunos de sus métodos y atributos, algunas clases se muestran para poder completar la estructura del diagrama pero no van a ser detalladas, a continuación se va a dar una breve descripción de las clases que si se detallan.

- clang::Sema, esta clase es la encargada de realizar el análisis semántico y la construcción del AST para lenguaje C.
- clang::CodeGenFunction, esta clase organiza el estado de cada función durante la generación de código LLVM IR.

[4] Este se encuentra en llvm-project/llvm/lib/Target/RISCV
[5] Este se encuentra en llvm-project/llvm/lib/Target/RISCV

- clang::Builtin::Context, almacena información sobre todos los *Builtins* en clang, para facilitar la consulta para otras clases.
- llvm::IRBuilder, provee la creación de instrucciones y la permite insertarlas en un punto arbitrario en los bloques básicos (Para código LLVM IR).
- llvm::Function, esta clase representa un procedimiento único en LLVM, esta contiene una lista de argumentos y de bloques básicos para la función.
- llvm::IntrinsicInst, esta clase se utiliza para inspeccionar llamadas a funciones *Intrinsics*.
- llvm::Intrinsic, contiene todas las funciones *Intrinsics* conocidas por LLVM.
- llvm::Instruction, es la clase base para todas las instrucciones de LLVM, contiene el *operation code* y el bloque básico al que pertenece una instrucción.
- llvm::Module, representa la estructura de más alto nivel en LLVM, representa una o varias unidades de traducción combinadas, Contiene una lista de funciones, variables globales y símbolos.
- SelectionDAG, se utiliza para representar porciones de funciones LLVM como un DAG, que incluye dependencias y no depende de la arquitectura objetivo. Este DAG es el primer paso para la selección de instrucciones.
- llvm::TargetLowering, esta clase define información que se utiliza a la hora de generar el selectionDAG a partir de código LLVM.
- llvm::MachineFunction, contiene una lista de bloques básicos de máquina, que forman una función compilada, ademas también contiene información de la arquitectura objetivo para el código generado.
- llvm::MachineInstr, contiene una representación para cada instrucción de máquina.
- llvm::RISCVTargetMAchine, esta es una interfase para acceder a toda la información relacionada a la arquitectura de RISCV.
- llvm::AsmPrinter, esta es la clase básica para todos los escritores de código ensamblador.

En la Figura 12 se muestra un diagrama de secuencia con el flujo del compilador al compilar un programa escrito en C hasta generar código ensamblador de RISCV. El flujo comienza al invocar el compilador mediante clang con un archivo de extensión .c, indicando además que la arquitectura objetivo va a ser RISCV de 32bits.

Luego el archivo pasa por el preprocesador y el *lexer* para convertir el código en una secuencia de *tokens*, el *parser* toma estos *tokens* uno a uno y genera un árbol con ellos con los que el analizador semántico va a generar luego un AST después de verificar forma y tipos.

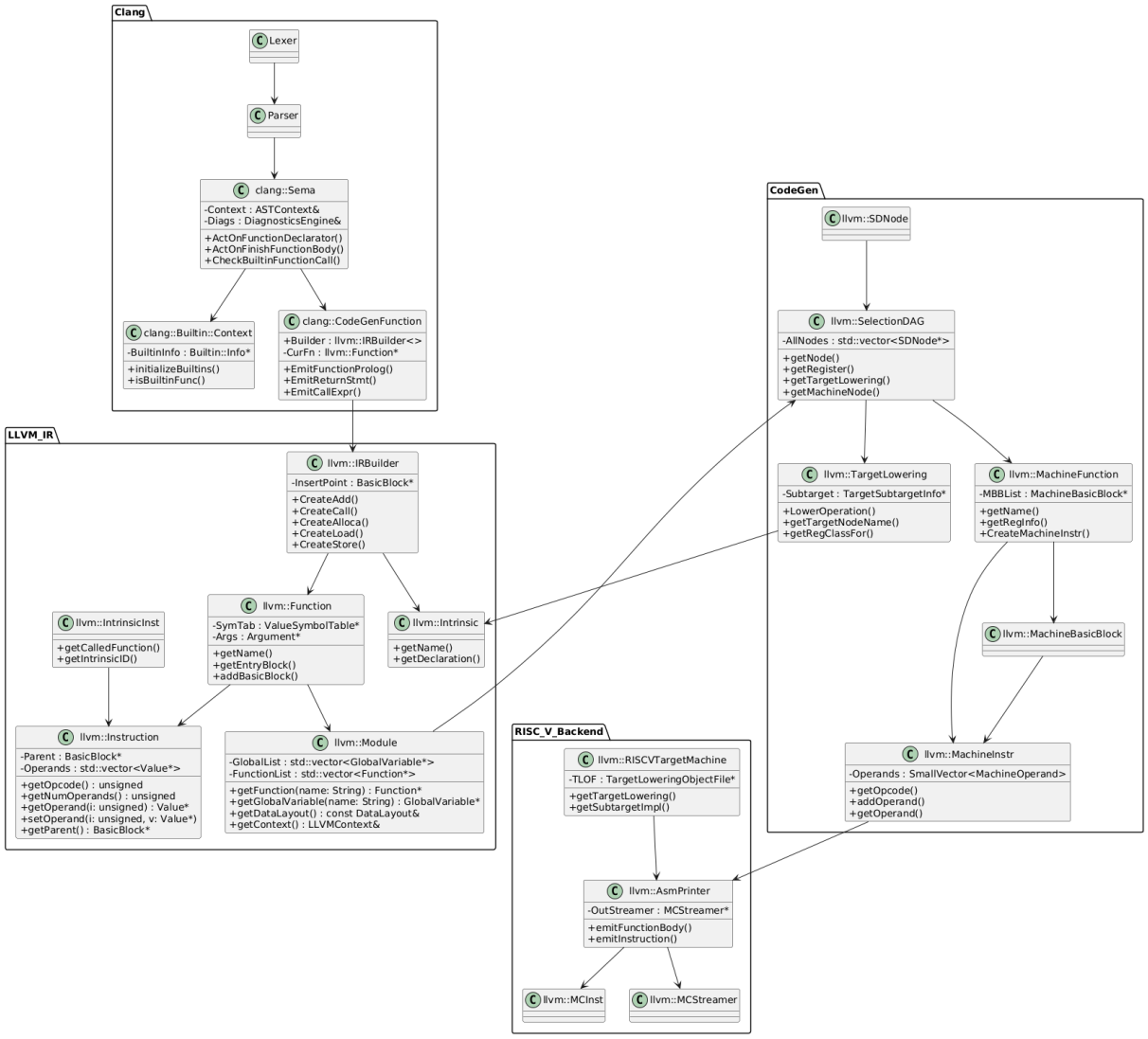

Fig. 11: Diagrama de clases general del compilador

Luego inicia la generación de código LLVM IR donde se crean las instrucciones, como *add, calls, mul, instrinsics* etc, luego estas instrucciones van construyendo estructuras de código más complejas. Cuando el código está listo esta se optimiza por los distintos *passes* del compilador.

Con el código optimizado se da paso a la selección de instrucciones que comienza al empezar a formar el DAG, transformando poco a poco instrucciones en nodos de este. Para finalizar los nodos del DAG que son independientes de la arquitectura objetivo se convierten en nodos del DAG de máquina, los cuales ya dependen de la arquitectura, estos nodos luego se convierten a instrucciones de ensamblador y se genera el código para la arquitectura objetivo.

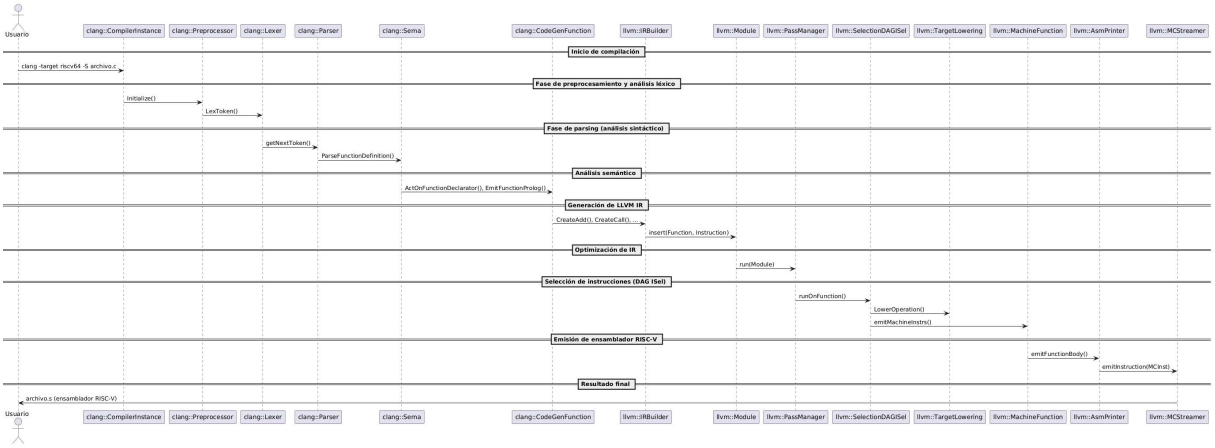

Fig. 12: Diagrama de secuencia del proceso de compilación

Capítulo 4

Resultados y análisis

En esta sección se presentan los resultados obtenidos al utilizar el compilador con las instrucciones agregadas, para luego continuar con el análisis de estos resultados y tener una mejor perspectiva de lo logrado con la extensión del compilador. Para obtener los resultados se van a implementar en el lenguaje C algunos algoritmos comunes que pueden aprovechar el uso de instrucciones SIMD (Single Instruction Multiple Data). Los algoritmos se van a implementar de dos formas, una implementación SISD (Single Instruction Single Data) utilizando solamente la base de C sin ninguna otra librería, y otra implementación SIMD que utiliza además de C estándar *intrinsics* de RVV y los *intrinsics* agregados para el proyecto.

El objetivo de hacer una implementación SIMD y otra SISD es poder comparar ciertas características, tanto de los códigos en C como de los códigos ensamblador generados por el compilador. Del código en C, se van a comparar la cantidad de líneas de código necesarias para implementar los algoritmos, mientras que de ensamblador se va a comparar el uso de registros (cantidad de registros diferentes utilizados), la cantidad de operaciones que involucran memoria, la cantidad de operaciones totales y el peso de los archivos generados. Además, se va a realizar al final un análisis de las implicaciones de usar *intrinsics* a la hora de escribir el código.

Además, se hablara de los resultados obtenidos de la investigación de las herramientas consideradas para resolver el problema, y analizar el porque se escogió LLVM y en que casos se podría querer migrar a otra herramienta. También se repasará la diferente documentación generada para el proyecto, principalmente con el fin que se pueda seguir expandiendo el compilador a medida que se definan nuevas instrucciones del ISA.

4.1 Herramientas consideradas

En la Tabla 4 se muestra una comparación de las tres herramientas que se consideraron para el desarrollo del proyecto, donde se brinda información de ciertas características importantes de las herramientas, que se utilizaron para escoger cual era la más indicada

para el proyecto.

Características	LLVM	TVM	ONNC
Enfoque	Tiene un enfoque muy general, permitiendo desde el desarrollo de *fronteds/backends* hasta la extensión de los existentes para diferentes lenguajes de alto nivel y arquitecturas.	Enfocado en compilar y optimizar modelos de aprendizaje de maquina y aprendizaje profundo.	Enfocado en compilar y optimizar modelos en forma de redes neuronales.
Entrada	Código en lenguajes como C/C++, Rust, Swift, Fortran y muchos otros lenguajes populares o en lenguajes para los que se desarrolle un *frontend*.	Modelos en formatos como TensorFlow, Pytorch, ONNX, Keras, Darknet y más.	Modelos en formato ONNX.
Hardware Objetivo	Una gran variedad de arquitecturas de CPUs como RISCV, ARM, MIPS entre muchas más, también GPUs con arquitecturas como CUDA o bien una arquitectura para la que se haya extendido el *backend*.	CPUs, GPUs, DSPs (Procesadores de Señales Digitales, siglas del ingles), microcontroladores, FPGAs. Muchas de las arquitecturas que soporta LLVM.	Desarrollado principalmente para aceleradores de aprendizaje profundo independientes, pero también funciona para GPUs, CPUs y DSPs que soporten LLVM.
Optimizaciones	Optimizaciones entre procesos, optimizaciones de ciclos, optimizaciones escalares estándar, eliminación de código muerto	Optimizaciones de grafos, optimizacion de programas de tensores, despacho de librerías, calendarización automática basada en aprendizaje	Selección de tensores, análisis de vitalidad de tensores, asignación de memoria mediante escaneo linear local entre otros.
Licencia	Código abierto (Apache 2.0).	Código abierto (Apache 2.0).	Código abierto (Apache 2.0).

Características	LLVM	TVM	ONNC
Facilidad	Es complejo, por la gran versatilidad que ofrece, es muy extenso y requiere conocimientos específicos del formato de los archivos *table description* propio de LLVM y de C++	Es más amigable, al permitir usar modelos en formatos como pytorch o trensorflow, ademas de tener parte de su implementación hecha con Python y su base con C++.	Es algo más amigable que LLVM, al ser más enfocado en ciertas tareas, también esta escrito principalmente en C++.
Extensibilidad	Tiene un diseño modular enfocado en ser extendible, permitiendo agregar nuevos *frontends*, nuevas fases de optimizacion y *backends* para distintas arquitecturas, o bien modificar los existentes.	Se pueden extender para nuevas arquitecturas objetivo, ademas de extender el proceso de optimizacion con nuevas fases.	Es modular, permite agregar soporte a nuevas arquitecturas objetivo, principalmente de DLAs, esta integrado con ciertas funciones de LLVM y también permite agregar nuevas fases de optimizacion.

Tabla 4: Tabla comparativa de las herramientas consideradas para el desarrollo

Por su capacidad de ser extendidas las tres opciones eran candidatas a tener en cuenta para el proyecto, ademas de las tres tener la posibilidad de trabajar con arquitecturas independientes y ser de código abierto, LLVM resultó ser la mejor opción para el desarrollo de este proyecto ya que lo que se buscaba era poder compilar de C a RISCV, por lo que se necesitaba una herramienta más flexible incluso si esta es la opción menos amigable para comenzar.

Otra ventaja de haber seleccionado LLVM es que ya que el acelerador se espera utilizar con modelos de aprendizaje profundo, en un futuro cuando se hayan desarrollado estos modelos y se haya finalizado por completo el diseño del ISA y del acelerador, el compilador se podría migrar sin mucha dificultad a alguna de estas otras dos opciones, si fuera conveniente, esto ya que Apache TVM utiliza a LLVM como base y ONNC a pesar que no lo usa como base directamente si tiene mucha influencia de LLVM en su diseño.

4.2 Instrucciones agregadas

En la Tabla 5 se muestran todas las instrucciones específicas del acelerador que se agregaron al compilador, en la tabla se indica la forma en que se llama a estas instrucciones en C y la forma en que se representan en ensamblador. En los algoritmos implementados

en las siguientes secciones no fue necesario el uso de todas estas instrucciones, pero todas fueron verificadas para comprobar que se definen en la forma deseada en C y se compilan de la forma esperada a ensamblador.

Instrucción en C	Instrucción en ensamblador
vint32m1_t v = __riscv_mul_vec_scalar(vint32m1_t v, int32_t s)	mul_vec_scalar vd, vs1, rs2
vint32m1_t v = __riscv_sum_vec_scalar(vint32m1_t v, int32_t s)	sum_vec_scalar vd, vs1, rs2
vint32m1_t v = __riscv_move_left_vec(vint32m1_t v, int32_t s)	sum_vec_scalar vd, vs1, rs2
vint32m1_t v = __riscv_move_right_vec(vint32m1_t v, int32_t s)	mov_right_vec vd, vs1, rs2
vint32m1_t v = __riscv_saturate_vec(vint32m1_t v, int32_t s)	saturate_vec vd, vs1, rs2
vint32m1_t v = __riscv_mul_vec(vint32m1_t v, vint32m1_t v)	mul_vec vd, vs1, vs2
vint32m1_t v = __riscv_sum_vec(vint32m1_t v, vint32m1_t v)	sum_vec vd, vs1, vs2
vint32m1_t v = __riscv_copy_vec(vint32m1_t v, vint32m1_t v)	copy_vec vd, vs1, vs2
vint32m1_t v = __riscv_extend_vec(vint32m1_t v, vint32m1_t v)	extend_vec vd, vs1, vs2

Tabla 5: Representación en C y ensamblador de las instrucciones agregadas

En la representación en C, el tipo de dato vint32m1_t es un tipo agregado a C al incluir *riscv_vector.h* este significa que el dato es un vector de enteros de 32 bits (para un multiplicador de grupo 1), el int32_t también se incluye y representa enteros de 32 bits. En la forma de ensamblador vd representa un registro vectorial destino, vs1 y vs2 representan registros vectoriales fuente y rs2 un registro escalar fuente.

En la Figura 13 se muestra un fragmento de código C donde se utilizan tres de los *intrinsics*, primero para multiplicar un vector ya cargado por un escalar para luego moverlo cierto número de veces a la izquierda y finalizar sumándolo con el valor de otro vector, en la Figura 14 se muestra el resultado de este fragmento en ensamblador, se observa como cada instrucción de C se convierte a una en ensamblador respectiva (la instrucción addi es de RISCV y en el fragmento controla la cantidad que se mueve el vector a la izquierda según el ciclo del bucle en que se está).

```
vint32m1_t mul2 = __riscv_mul_vec_scalar(v_fila,kernel[k_row][k_col]);
mul2 = __riscv_move_left_vec(mul2,k_col);
mul1 = __riscv_sum_vec(mul1, mul2);
```

Fig. 13: Fragmento de código C con ejemplos de intrinsics

```
mul_vec_scalar  v10, v8, a5
mov_left_vec    v10, v10, a1
addi    a1, a1, 1
sum_vec v9, v9, v10
```

Fig. 14: Fragmento de código ensamblador con ejemplos de intrinsics

De forma similar en la Figura 15 se muestra otro fragmento de ejemplo en C donde se utilizan dos *intrinsics* primero para mover un vector cargado una cantidad de veces a la

derecha y luego para multiplicar este vector con el vector inicial. Y en la Figura 16 se muestra el resultado en ensamblador.

```
vint32m1_t mov_r_vec = __riscv_move_right_vec(v_fila, i);
vint32m1_t mul = __riscv_mul_vec(v_fila, mov_r_vec);
```

Fig. 15: Sergundo fragmento de código C con ejemplos de intrinsics

```
mov_right_vec   v8, v8, a1
mul_vec v8, v8, v9
```

Fig. 16: Segundo fragmento de código ensamblador con ejemplos de intrinsics

4.3 Multiplicación de matrices

El primer algoritmo que se implementó es una multiplicación de matrices elemento por elemento es decir para un par de matrices A y B de tamaños NxM el resultado es una matriz C tal que $C_{ij} = A_{ij} * B_{ij}$, se implementó esta forma ya que se beneficia de uno de los *intrinsics*implementados para el acelerador, además es una operación común para aplicar filtros en procesamiento de señales.

Tanto la implementación SIMD como la SISD se compilaron para matrices de distintos tamaños para así poder comparar también el impacto que tienen las dimensiones de las matrices en las características que se van a evaluar. Se utilizaron tamaños pequeños para las matrices para poder definirlas de forma manual y que así el código generado sea más sencillo de analizar y se enfoque solo en la operación de multiplicar las matrices.

En la Figura 17 se muestra una gráfica de cómo cambia la cantidad total de operaciones en el código ensamblador para diferentes tamaños de matrices, en la implementación SIMD, el código C para el algoritmo consiste solamente de un ciclo que recorre todas las filas de la matriz, en el ciclo se utilizan *intrinsics* de RVV para cargar las filas como vectores y guardar luego los resultados, además del *intrinsic* vec_mul definido para el acelerador para multiplicar las filas de las dos matrices.

Y en la Figura 18 se muestra una gráfica similar para la implementación SISD, para este caso la implementación cuenta con dos bucles, uno para recorrer cada columna de una fila, y otro para recorrer todas las filas, en cada iteración se realiza una multiplicación de escalares y se guarda el valor en la matriz de salida.

Para cada aumento en el tamaño de las matrices se cuadriplican la cantidad de datos que las conforman, en la gráfica SIMD se puede ver que la cantidad total de instrucciones crece

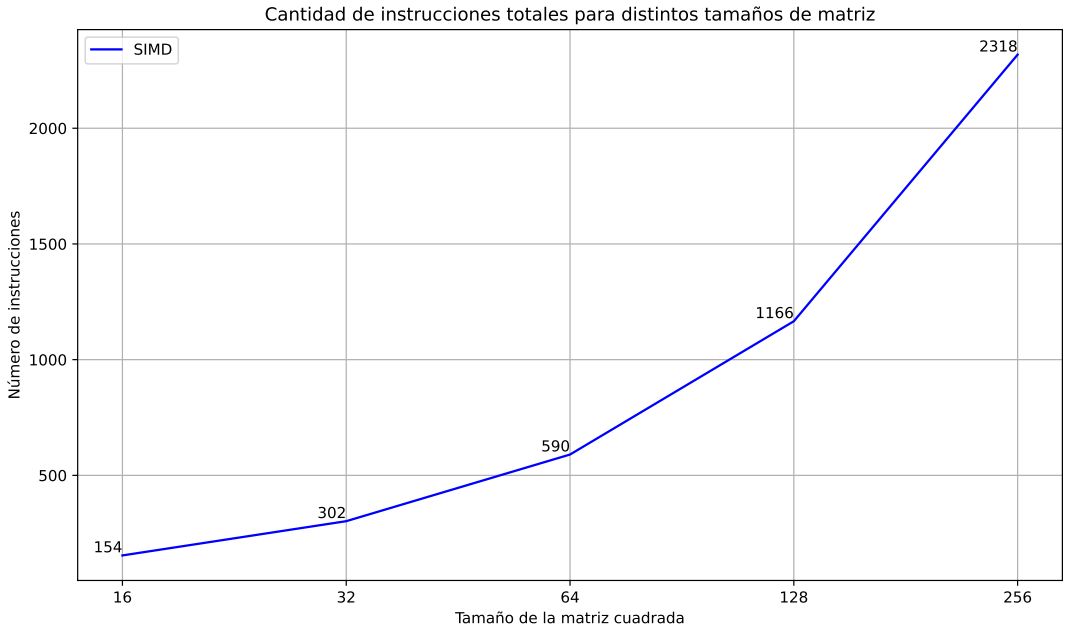

Fig. 17: Cantidad de instrucciones para distintos tamaños de matriz con implementación SIMD

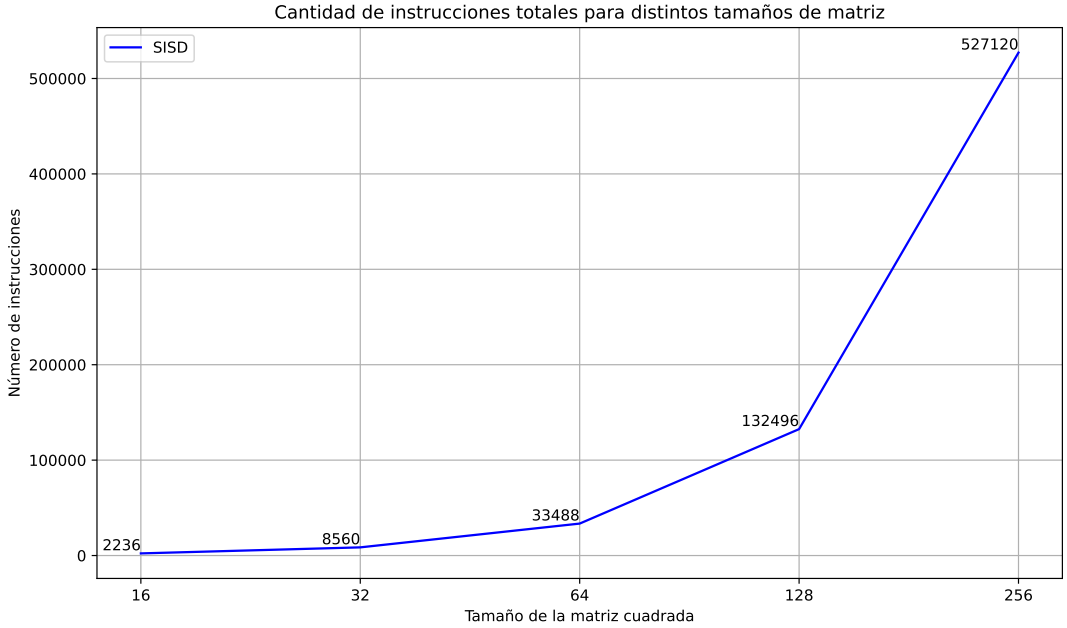

Fig. 18: Cantidad de instrucciones para distintos tamaños de matriz con implementación SISD

casi el doble entre distintos tamaños, mientras que en la gráfica SISD se puede observar que el crecimiento es de casi cuatro veces más instrucciones. Esto tiene un gran impacto ya que en el caso SIMD entre matrices de 16x16 y 256x256 la cantidad de instrucciones crece 15 veces, mientras que para el caso SISD la cantidad aumenta en 235 veces.

El hecho de que la cantidad de instrucciones para el caso SIMD tienda a duplicarse y en el caso SISD tienda a cuadriplicarse se da por la cantidad de bucles, con solo un ciclo menos al utilizar instrucciones SIMD se reduce considerablemente la cantidad de instrucciones totales. Por ejemplo, para matrices de 16x16 se necesitan 15 veces más operaciones al utilizar SISD en lugar de SIMD, mientras que para matrices de 256x256 se necesitan 227 veces más operaciones.

En la Figura 19 y la Figura 20 se muestran gráficas para la cantidad de instrucciones que acceden a memoria según el tamaño de las matrices para la implementación SIMD y la SISD respectivamente. El comportamiento es similar al de las instrucciones totales, en el caso SIMD la cantidad de accesos a memoria se duplican al crecer las matrices mientras que para el caso SISD se cuadriplican.

Al comparar la cantidad de accesos a memoria en matrices 16x16 y matrices 256x256 se observa que en el caso SISD se requieren 256 veces más accesos a memoria al aumentar el tamaño de las matrices, mientras que en el caso SIMD solo 16 veces más accesos a memoria. Este es un comportamiento similar al visto para instrucciones totales, pero se observa que el crecimiento es aún mayor en accesos a memoria que en instrucciones totales, en el caso SISD la cantidad de instrucciones que acceden memoria pasa de ser de un 34.34% de las instrucciones totales a un 37.29%, mientras que para SIMD pasa de 31.16% a un 33.13%.

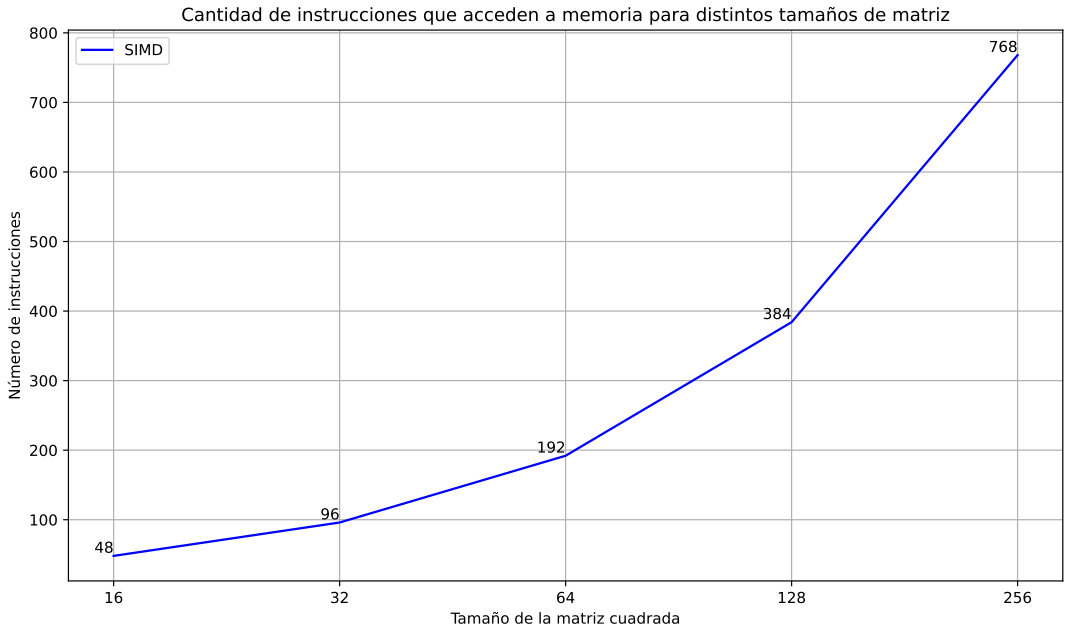

Fig. 19: Cantidad de instrucciones que acceden a memoria para distintos tamaños de matriz con implementación SIMD

De este análisis se puede observar el beneficio que tiene el uso de *intrinsics* en la cantidad de instrucciones que deben ser ejecutadas para realizar la multiplicación de matrices, en este caso con el uso únicamente de la operación especializada vec_mul.

Fig. 20: Cantidad de instrucciones que acceden a memoria para distintos tamaños de matriz con implementación SISD

Respecto al uso de registros, en la Tabla 6 se muestran los registros utilizados por cada implementación, en este caso no es necesario indicarlos según el tamaño de las matrices ya que para todos los tamaños de las matrices se utilizaron los mismos registros (cambiaron solo entre la implementación SIMD y SISD).

Implementación	Registros	Total
SIMD	sp, a0, a1, a2, a3, zero, v8, v9	8 registros
SISD	sp, t2, a6, t0, t1, a7, t3, a1, a3, a2, a5, a0, a4	13 registros

Tabla 6: Uso de registros para cada implementación

La diferencia puede parecer poca ya que solo se utilizan 5 registros más en la implementación SISD que en la SIMD, pero al ser el total de registros utilizados 13 y 8 respectivamente, estos 5 registros de más representan un aumento del 61% en el uso de registros entre la implementación SIMD a la SISD. El uso extra de registros se debe a que la implementación SISD al hacer uso de un ciclo más, requiere de más registros para controlar el flujo del programa, por ejemplo, para llevar un contador más o una condición de salida extra. Por lo que con solo el uso de la instrucción SIMD vec_mul también se puede obtener una mejora considerable en el uso de registros.

Analizando ahora el tamaño de archivos, en la Tabla 7 se muestra la comparación del tamaño de los archivos con el código ensamblador (extensión .s) generados para los distintos tamaños de matrices, tanto para la implementación SIMD como para la SISD.

De esta tabla podemos observar que el tamaño de archivo no difiere demasiado entre SIMD y SISD, siendo las diferencias para los tamaños 16x16, 32x32, 128x128 y 256x256

Tamaño matriz	Tamaño archivo SIMD [bytes]	Tamaño archivo SISD [bytes]
16x16	24 563	24 976
32x32	93 733	94 151
128x128	1 476 137	1 476 558
256x256	5 908 010	5 908 437

Tabla 7: Tamaño de los archivos con el código ensamblador para SIMD y SISD

de 413, 418, 421 y 427 bytes respectivamente. Esto ya que se requiere más código al manejar el flujo de los dos bucles para la implementación SISD que para el único ciclo de la implementación SIMD. Por lo también existe una mejora en el tamaño del archivo relacionada al uso de instrucciones SIMD, que si bien parece baja, el impacto puede depender del contexto ya que en dispositivos con recursos limitados cada byte ahorrado puede tener importancia.

La diferencia principal está en el tamaño de los archivos para los diferentes tamaños de matrices, por lo que se debe tener esto en consideración si se quiere manejar matrices de mayor tamaño, ya que el almacenar estos datos va a representar la mayor parte del espacio utilizado por el archivo de extensión .s.

Por último, respecto a la implementación del algoritmo en C. Este es un código bastante sencillo, por lo que se utilizaron solo 7 líneas de código para la implementación SIMD y 4 lineas para la implementación SISD, sin considerar las líneas para definir los datos. Por lo que la implementación SIMD requirió un 57% más líneas de código que la SISD, esto ya que se debe hacer la carga y guardado de los datos de forma manual. Además, se deben utilizar *intrinsics* relacionados a la arquitectura objetivo lo que a su vez implica una mayor dificultad al escribir el código ya que requiere que se conozca como llamarlos y su funcionamiento.

El análisis en esta sección se hizo sobre un algoritmo muy sencillo como lo es la multiplicación de matrices elemento por elemento, con el fin de poder observar el impacto que puede tener el uso de instrucciones SIMD incluso en algoritmos sencillos. En la siguiente sección del capítulo se va a implementar un algoritmo complejo como lo es la convolución, que implica el uso de más instrucciones SIMD para poder ver las implicaciones a mayor escala.

4.4 Convolución

El algoritmo implementado es una parte de la convolución, es específicamente lo que se va a ejecutar en los elementos de procesamiento no compartidos del acelerador. Para ejecutar la convolución en estos elementos de procesamiento se planea utilizar una interpretación diferente de la operación, que permite una mejor ejecución en paralelo, pero el resultado es el mismo que con la interpretación estándar. Para las pruebas del compilador se

implementó un código que permite ejecutar la parte de acumulación del kernel de esta forma de la convolución. Al igual que para la multiplicación de matrices, se realizó una implementación SIMD y una SISD.

Comenzando por el análisis de los códigos C para implementar el algoritmo, en el caso SIMD se implementó utilizando 14 líneas de código, sin contar la definición de variables, el código necesita solo de dos bucles anidados y utiliza las *intrinsics* vec_mul_scalar, move_left_vec y sum_vec del acelerador, además de las de RVV para cargar y guardar a memoria. En el caso SISD se necesitaron 36 líneas de código, además de los dos bucles anidados, se debieron implementar funciones equivalentes a los *intrinsics* utilizados, para cargar los datos y para mover un vector a la izquierda se necesitó de funciones que cuentan con 2 bucles cada una, para la suma de vectores y multiplicar un vector por un escalar se utilizó un ciclo para cada tarea.

La implementación SIMD requiere de un 61.11% menos líneas de código en C respecto a la implementación SISD, si bien requiere de conocimiento extra para utilizar los *intrinsics* los resultados muestran que es algo que vale la penas utilizar al implementar. Además, el código SIMD utiliza 66% menos bucles, lo cual es de suma importancia ya que los bucles anidados representan el mayor crecimiento en la cantidad de operaciones que deberá ejecutar el *hardware*, ya que si por ejemplo en un ciclo que se repite 3 veces se utiliza una función que necesita de un ciclo que se repite 5 veces, este ciclo interno se va a repetir al final 15 veces, y el crecimiento es mayor a medida que se necesiten más bucles anidados.

En la Figura 21 se muestran la cantidad de instrucciones totales y cantidad de instrucciones que requieren acceder a memoria que debe procesar el *hardware* para ejecutar esta etapa de la convolución haciendo una implementación SIMD. Este resultado se obtiene para procesar un único vector de entrada, el vector contiene 8 enteros de 32 bits, pero se trabaja con tamaño de vectores de 10 elementos para poder hacer el desplazamiento a la izquierda sin perder información. Se utiliza un kernel de tamaño 3x3, con datos enteros de 32 bits, que es con el cual se procesa el vector.

Para la implementación SIMD los accesos de memoria representan un 15% de las instrucciones totales que se deben ejecutar, los accesos a memoria consideran tanto carga y guardado de datos vectoriales como de datos escalares.

En la figura 22 se muestra una gráfica similar para el caso de la implementación SISD, las condiciones fueron las mismas, un único vector de entrada para procesar, de tamaño 10 pero con solo 8 enteros de 32 bits, y un kernel 3x3 con enteros. Para esta implementación se observa que los accesos a memoria representan un 37% de las instrucciones totales.

Al analizar ambos resultados es claro que la implementación SISD requiere más accesos a memoria en proporción a la cantidad de instrucciones totales, con una diferencia de 22% más que la proporción para la implementación SIMD. Esto es importante ya que los accesos a memoria son instrucciones que dependen de los tiempos y métodos que tiene el *hardware* para acceder a memoria y suelen ser instrucciones tardadas, además que para el compilador son más difíciles de optimizar ya que estas generan efectos secundarios.

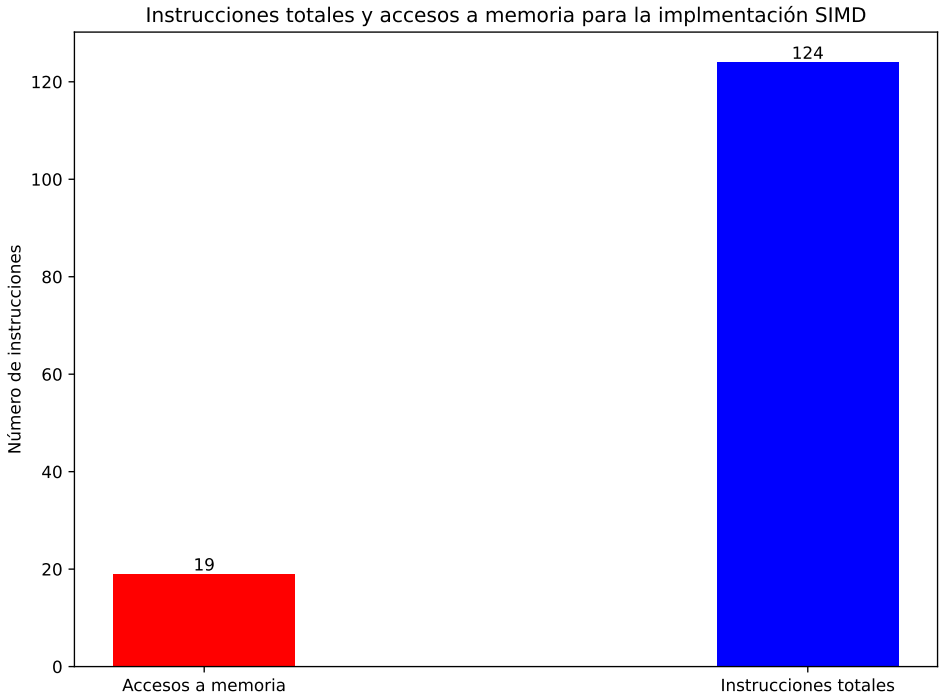

Fig. 21: Instrucciones y accesos a memoria totales para la implementación SIMD de una etapa de la convolución

Comparando ahora las Figuras 21 y 22 directamente se puede ver que la implementación SISD requiere de 26 veces más instrucciones totales que la versión SIMD para ejecutar el algoritmo, además se requiere 62 veces más accesos a memoria. Esta comparación permite observar aún mejor la ventaja que tiene SIMD respecto a los accesos de memoria, ya que en el caso de SISD a pesar de que las instrucciones totales son solo 26 veces más, los accesos a memoria son 62 veces más.

En el caso del uso de registros en la Tabla 8 se muestran los registros utilizados para la implementación SIMD y la SISD y el total de registros. De la tabla se obtiene que la implementación utilizando SIMD utiliza 31% menos registros que la implementación SISD, esto es algo esperable ya que el código SISD es mucho más extenso y requiere de más registros que controlen el flujo del programa, por lo que de nuevo hay una ventaja en el uso de SIMD relacionada con el reducir la cantidad de bucles necesarios para ejecutar el algoritmo.

Implementacion	Registros	Total
SIMD	zero, a0, a1, a2, a3, a4, a5, a6, a7, t0, v8, v9, v10	13 registros
SISD	zero, sp, ra, a0, a1, a2, a3, a4, s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11	19 registros

Tabla 8: Registros utilizados en la convolución en la implementación SIMD y SISD

Para analizar el espacio ocupado por los archivos con el código ensamblador (extensión

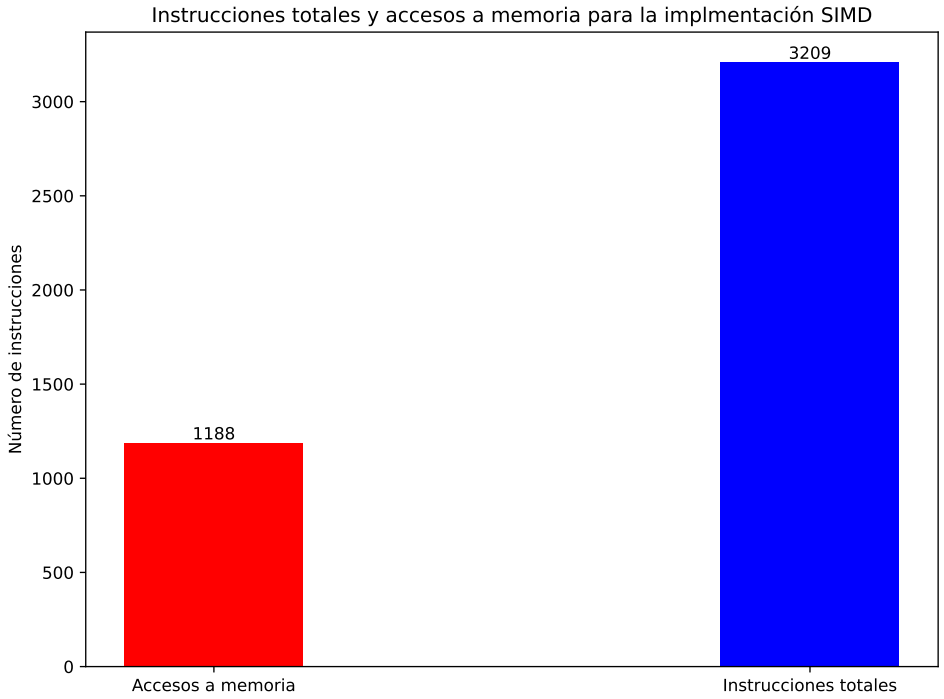

Fig. 22: Instrucciones y accesos a memoria totales para la implementación SISD de una etapa de la convolución

.s) y el tiempo que toma compilar el código C para generar el ensamblador, se presenta esta información en la Tabla 9. Para obtener el tiempo de compilación se realizó 10 veces el proceso de compilar y se utilizó el promedio de las 10 ejecuciones con el fin de tener un resultado más confiable.

Implementación	Tamaño del archivo ensamblador [bytes]	Tiempo de compilación [s]
SIMD	4 359	0,0885
SISD	14 406	0,0661

Tabla 9: Espacio ocupado y tiempo necesario para generar el archivo ensamblador

Al observar la tabla se puede apreciar que en este caso la diferencia en el tamaño de los archivos .s si es considerable, la versión SISD ocupa 10047 bytes más que la SIMD, lo que representan un aumento del 330% en el espacio necesario para almacenar el código SISD, esto es un aspecto clave en casos de *hardware* con recursos limitados, que en este caso se verían muy beneficiados por el ahorro de espacio necesario para almacenar el programa.

Respecto al tiempo necesario para compilar, a pesar de ser un archivo mucho más ligero el generado, el proceso si es más tardado para la implementación SIMD requiriendo esta 0,0224s más en promedio para generar el archivo, lo que representa un aumento del 132% respecto al SISD. Esto es algo que se esperaba ya que a pesar de generar menos código requiere compilar los *intrinsics* tanto del acelerador como de RVV lo que requiere utilizar y verificar más archivos e información, además del uso de características específicas de la

extensión RVV, como los registros de tipo vectorial. Esto puede o no ser relevante según el contexto donde se vaya a utilizar el compilador, ya que si se compila antes de ejecución esta diferencia no tiene mucho impacto, pero si se busca una compilación durante tiempo de ejecución esta diferencia si podría ser importante.

4.5 Documentación

El siguiente enlace lleva al repositorio de github con la documentación generada para el proyecto. Se va a proceder a explicar brevemente el contenido de los diferentes documentos.

- Informe.pdf, es una copia de este documento, en este los interesados van a encontrar la información detallada de lo investigado, tanto en teoría como en herramientas a tomar en cuenta, detalles de la solución del problema, no solo del proceso para extender LLVM pero también información sobre la estructura del compilador, recomendaciones para el futuro y las conclusiones obtenidas del proyecto.
- README.md, contiene el enlace con las instrucciones para instalar Clang y LLVM, instrucciones para agregar las modificaciones, recomendaciones y ejemplos para compilar junto a explicación de la instrucción para llamar al compilador, información de que es cada archivo en el repositorio.
- Guía-extensión.pdf, en este se muestra unicamente la información del proceso de extensión del compilador, el paso a paso y la explicación de los detalles importantes y las decisiones, ademas de algunas recomendaciones. Esto con el fin de facilitar el proceso de seguir extendiendo con nuevas instrucciones al hacer que sea más sencillo revisar la documentación relevante.
- TestIntrinsics.c, contiene un ejemplo donde se muestra la forma de llamar a cada *intrinsic* en el código C, y de como utilizar los *intrinsics* de RVV para cargar, guardar y definir el tamaño de vector.

Capítulo 5

Conclusiones

La mejor forma de desarrollar un compilador para una arquitectura de un acelerador desarrollada por un particular es el uso de *frameworks* de código abierto ya existentes enfocados en la posibilidad de ser extendidos o modificados para las necesidades del desarrollador, ya que el desarrollar el compilador desde cero es una tarea muy costosa. Si se desea hacer un compilador de algún lenguaje de alto nivel a ensamblador la mejor opción es LLVM, ya que al ser modular permite mucha versatilidad a la hora de extenderlo. Por otro lado si se quiere compilar específicamente modelos de aprendizaje profundo, opciones como Apache TVM o ONNC están enfocadas en específicamente la compilación de modelos representando en formatos comunes como ONNX, pythorch, entre otros.

LLVM permite facilitar el proceso de desarrollo de compiladores para aceleradores desarrollados por particulares, mediante su enfoque modular se puede hacer extensión de este sin la necesidad de modificar todas sus etapas. Según las necesidades del compilador y las características del acelerador se pueden aprovechar muchos de los módulos existentes como el *frontend* de algún lenguaje de alto nivel o distintas optimizaciones independientes de la arquitectura objetivo. El agregar nuevos *intrinsics* a un lenguaje como C permite aprovechar la base existente para manejo de bucles, definiciones de variables, verificación de tipos, entre otras, a la vez que se tiene la posibilidad de hacer uso directo de ciertas instrucciones del *hardware* objetivo, permitiendo exponer funcionalidades de bajo nivel en un lenguaje de alto nivel.

El uso de instrucciones especializadas en este caso de tipo SIMD, mediante *intrinsics*, aumentan la dificultad de producir el código C pues es necesario conocimiento extra de estas instrucciones y del *hardware* objetivo, sin embargo presentan ventajas en el código ensamblador generado, especialmente en ciertas tareas que requieren el procesar vectores o estructuras de datos similares, disminuyendo la cantidad de instrucciones que el *hardware* debe procesar, disminuyendo también la cantidad de accesos a memoria, tanto totales como en proporción a la cantidad de instrucciones, disminuyendo el uso de registros y con esto también generando archivos más ligeros y fáciles de analizar.

El uso de *intrinsics* y el extender el compilador aumenta los tiempos de compilación, por

lo que es importante ser cuidadoso con su uso en casos donde la compilación deba cumplir ciertas restricciones de tiempo, hacer código eficiente que utilice los *intrinsics* necesarios y extender el compilador de igual forma solo con las instrucciones necesarias.

Con la documentación generada durante el desarrollo del proyecto se facilitará a los miembros interesados de ECASLab el continuar con el desarrollo del compilador a medida que el ISA del acelerador siga evolucionando, utilizando la información presente en este documento podrán ver como funcionan las instrucciones ya implementadas pero más importante ver el paso a paso para agregarlas y tener el conocimiento de que es cada detalle en cada etapa, además de información extra y recomendaciones que van a ser de utilidad a medida que se necesiten instrucciones más complejas o para otras tareas como manejo de memoria. Además con los códigos usados para los resultados se presentan ejemplos reales de como se pueden utilizar los *instrinsics* tanto del acelerador como de RVV.

5.1 Recomendaciones

A la hora de definir los *Builtins* que se van a agregar a C se recomienda tener en consideración como van a representarse estos luego mediante instrucciones reales del ISA, ya que estos pueden querer abarcar demasiada complejidad que en C puede ser posible pero luego en ensamblador no poder ser representados por una instrucción.

Se recomienda considerar añadir *intrinsics* que sirvan para configurar características del *hardware* o para preparar el programa para instrucciones que puedan necesitarlo, ya que esta alternativa puede ser más sencilla y controlada que generar relaciones de un *Builtin* a varias instrucciones de ensamblador.

Ya que el ISA que se propone para el acelerador va a tener ciertas características similares a la extensión vectorial de RISCV se recomienda analizar la posibilidad de aprovechar ciertas características que ya están implementadas en el compilador para RVV, para el manejo de tamaños de vectores variables, el manejo de que elementos de los vectores se quieren procesar, entre otras, ya que esto disminuye el trabajo al extender el compilador. O bien utilizar estas como guía o base a la hora de implementarlas de forma separada para el acelerador.

Bibliografía

[1] About. URL https://portale.units.it/en/university/about. Accessed on April 01, 2025.

[2] Qué es el tec. URL https://www.tec.ac.cr/que-es-tec. Accessed on April 01, 2025.

[3] Ghattas Akkad, Ali Mansour, and Elie Inaty. Embedded deep learning accelerators: A survey on recent advances. *IEEE Transactions on Artificial Intelligence*, 5(5):1954–1972, 2024.

[4] Md Zahangir Alom, Tarek M. Taha, Chris Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Mahmudul Hasan, Brian C. Van Essen, Abdul A. S. Awwal, and Vijayan K. Asari. A state-of-the-art survey on deep learning theory and architectures. *Electronics*, 8(3), 2019. URL https://www.mdpi.com/2079-9292/8/3/292.

[5] Ayoub Benali Amjoud and Mustapha Amrouch. Object detection using deep learning, cnns and vision transformers: A review. *IEEE Access*, 11:35479–35516, 2023.

[6] Karthikeyan Kalyanasundaram Balasubramanian, Mirco Di Salvo, Walter Rocchia, Sergio Decherchi, and Marco Crepaldi. Designing risc-v instruction set extensions for artificial neural networks: An llvm compiler-driven perspective. *IEEE Access*, 12:55925–55944, 2024.

[7] Andrés Bastidas Fuertes, María Pérez, and Jaime Meza Hormaza. Transpilers: A systematic mapping review of their usage in research and industry. *Applied Sciences*, 13(6), 2023. URL https://www.mdpi.com/2076-3417/13/6/3667.

[8] Thomas Benz, Michael Rogenmoser, Paul Scheffler, Samuel Riedel, Alessandro Ottaviano, Andreas Kurth, Torsten Hoefler, and Luca Benini. A high-performance, energy-efficient modular dma engine architecture. *IEEE Transactions on Computers*, 73(1):263–277, 2024.

[9] Dejan Bokan, Miodrag ukić, Miroslav Popović, and Nenad Četić. Adjustment of gcc compiler frontend for embedded processors. In *2014 22nd Telecommunications Forum Telfor (TELFOR)*, pages 983–986, 2014.

[10] Andrew Boutros and Vaughn Betz. Fpga architecture: Principles and progression. *IEEE Circuits and Systems Magazine*, 21(2):4–29, 2021.

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018. URL http://arxiv.org/abs/1802.04799.

[12] ONNX Contributors. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2024.

[13] Alessandro Rubini Jonathan Corbet and Greg Kroah-Hartman. *LINUX DEVICE DRIVERS*. O'Reilly Media, 2005.

[14] Linda Torczon Keith D Cooper. *Engineering a compiler*. Morgan Kaufmann Publishers, 2023.

[15] Celine Lee, Abdulrahman Mahmoud, Michal Kurek, Simone Campanoni, David Brooks, Stephen Chong, Gu-Yeon Wei, and Alexander M. Rush. Guess sketch: Language model guided transpilation, 2024. URL https://arxiv.org/abs/2309.14396.

[16] R. Leupers. Compiler design issues for embedded processors. *IEEE Design Test of Computers*, 19(4):51–58, 2002.

[17] Chenyang Li and Jiye Jiao. Llvm framework: Research and applications. In *2023 19th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, pages 1–6, 2023.

[18] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: Analysis, applications, and prospects. *IEEE Transactions on Neural Networks and Learning Systems*, 33(12):6999–7019, 2022.

[19] Wei-Fen Lin, Der-Yu Tsai, Luba Tang, Cheng-Tao Hsieh, Cheng-Yi Chou, Ping-Hao Chang, and Luis Hsu. Onnc: A compilation framework connecting onnx to proprietary deep learning accelerators. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 214–218, 2019.

[20] Zi Ming Xiong. A survey of fpga based on graph convolutional neural network accelerator. In *2020 International Conference on Computer Engineering and Intelligent Control (ICCEIC)*, pages 92–96, 2020.

[21] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015. URL http://arxiv.org/abs/1511.08458.

[22] Juan Pablo Soto Quiros. Convolución matricial aplicado al procesamiento de imágenes. URL https://www.tec.ac.cr/sites/default/files/media/doc/presentacion_pablosoto.pdf.

[23] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Pearson Education, 2010. URL https://books.google.es/books?id=49OmnOmTEtQC.

[24] Pramila P. Shinde and Seema Shah. A review of machine learning and deep learning applications. In *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, pages 1–6, 2018.

[25] Xilinx. Axi reference guide, 2012. URL https://www.xilinx.com/support/documents/ip_documentation/axi_ref_guide/latest/ug761_axi_reference_guide.pdf.

[26] Yu Xing, Jian Weng, Yushun Wang, Lingzhi Sui, Yi Shan, and Yu Wang. An in-depth comparison of compilers for deep neural networks on hardware. *IEEE*, pages 1–8, 2019.

[27] Guoqing Zhang and Tapani Ahonen. A llvm based compiler for coffee. In *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*, pages 1–5, 2013.

Apéndice A

Implicaciones sociales, éticas y de equidad del proyecto

Las aplicaciones de aprendizaje profundo representan una de las mayores revoluciones tecnológicas de la actualidad, permitiendo hacer tareas que antes parecían imposibles, pero estas se encuentran limitadas actualmente por la necesidad de muchos recursos computacionales para un funcionamiento eficiente. Tareas que pueden tener un impacto social importante como la conducción automática requieren de poder ejecutar los modelos en los dispositivos sin necesidad de obtener resultados de servidores en línea, esto para poder trabajar en tiempo real y reducir la latencia o evitar accidentes relacionados a problemas de conexión.

Como una herramienta importante para el acelerador desarrollado, el compilador permite que el uso de este sea posible para más desarrolladores, sin la necesidad de estar del todo familiarizados con el lenguaje ensamblador y la arquitectura del acelerador. Por lo que ayuda a la capacidad de que más personas se puedan involucrar en el desarrollo de soluciones en el borde que aprovechen las ventajas del acelerador y que de esta forma nuevas aplicaciones puedan tener un impacto positivo en las vidas de las personas que las utilizan.

Una de las tareas más populares que se puede beneficiar del uso del acelerador es la del desarrollo de vehículos con conducción automática, por un lado personas que por ciertas condiciones físicas o mentales no pueden conducir, se van a ver beneficiadas por esta tecnología permitiéndoles transportarse de forma más libre en sus propios vehículos, mientras que por otro lado personas que trabajan conduciendo vehículos, pueden perder sus trabajos y por ende sus fuentes de ingresos a medida que estas tecnologías sigan creciendo y ocupando estos espacios.

Por otro lado siguiendo con el ejemplo de la conducción automática, esta también está acompañada de cuestiones éticas relacionadas con los accidentes que se puedan dar cuando un vehículo es manejado por un modelo, generando preguntas como a quien le recae la culpa, como poder determinar si el accidente está causado por un problema del sistema

o simplemente se dio un contexto en el que era inevitable, cual debería de ser el margen de error que se puede permitir a un modelo cuya tarea puede llegar a poner en riesgo la integridad de las personas.

Tareas relacionadas con la medicina también pueden beneficiarse de la ejecución de modelos al borde en ciertos dispositivos médicos, ya que estas requieren de que los resultados estén a tiempo y no pueden verse afectadas por problemas de conectividad, avances en esta área pueden significar menos riesgos en ciertos procedimientos un mejor monitorio de los pacientes, reduciendo la carga de trabajo para los profesionales de la salud y así reduciendo problemas que pueden estar relacionados a distracciones o cansancio.

En cuestiones ambientales el desarrollo de nuevas tecnologías especializadas puede traer menores consumos energéticos o de recursos que otras tecnologías como las GPUs, al desarrollarse estas con un enfoque especifico y por ende poder estar más optimizadas para tareas específicas. Pero a su vez lo más probable es que estas más que un reemplazo de las tecnologías existentes, se utilicen como complementos a estas, por lo que serían una nueva forma de consumir recursos y quizás en lugar de disminuir el impacto ambiental, aumentarlo.

La inteligencia artificial es una tecnología que ha venido para quedarse y que dándole un buen uso puede beneficiar en muchas formas a la sociedad, pero no cada tarea realmente la necesita y se debe tener cuidado con su uso, el entrenamiento de nuevos modelos consume grandes cantidades de energía, espacio para los centros de datos y otros recursos ecológicos relacionados a la producción del hardware y mantenimiento.

Por esto es de verdad importante valorar que usos valen la pena desarrollar y no permitir que esta se convierta en un juguete que se use por cualquier persona solo para preguntar o conversar sin un propósito real, para generar imágenes que luego se van a olvidar o hacer videos que puedan generar desinformación o afectar a personas involucradas. Se debe tener cuidado ya que los usos que pueden ser más lucrativos no son siempre los que pueden generar un beneficio real para la sociedad, y que si solo se desarrolla con fines económicos convertir una herramienta con tanto potencial en un juguete o un problema.