Facultad de ciencias

Optimización de Librería de tensores para Aprendizaje automático mediante las extensiones de vectorización del procesador

(Tensor library optimization for automatic learning through the vectorial extensions of the processor)

Trabajo de Fin de Grado

Grado en ingeniería Informática

Autor: Sergio Martínez Arribas

Director: Pablo Abad Fidalgo

Junio-2024

RESUMEN

La informática está sufriendo una gran revolución y crecimiento en lo que se respecta a inteligencias artificiales, incorporándolas cada vez a más aspectos de nuestra vida. Uno de los tipos más comunes y usados ahora mismo son los *Large Language Models,* como puede ser el conocido *ChatGPT*, capaces de generar lenguaje como si de un humano se tratara.

Sin embargo, estos avances tienen una dificultad, los algoritmos usados en estas aplicaciones requieren grandes requerimientos computacionales y son muy costosos, y por tanto necesitan software optimizado y librerías matemáticas especializadas, como por ejemplo GGML.

La librería *GGML* es usada en algoritmos de *machine learning* y aprendizaje automático para tratar de ejecutar de forma eficiente operaciones con tensores en hardware de propósito general. Sin embargo, esta librería es reciente, sufriendo modificaciones constantes, tratando de adaptarse a los cambios en el hardware y los modelos. En este trabajo intentaremos ampliarla para que haga uso de las extensiones x86 para operaciones vectoriales, AVX512 y AMX, que no estaban siendo utilizadas en la librería en la fecha de realización de este TFG. Hacer uso de estas extensiones permitirá, a los procesadores que las incluyan, operar con un mayor número de datos de manera simultánea. Evaluaremos los resultados obtenidos con la herramienta perf, usando la librería a través de aplicaciones como llama.cpp o *benchmarks* propios.

ABSTRACT

The computing industry is nowadays suffering a big revolution and growth in the matter of Artificial Intelligences, making them more available in all the aspects of our daily lives. One of the more common and used are the Large Language Models such as the well-known *ChatGpt*, that can generate words and talks as a human would do.

All of this has a disadvantage, the algorithms used in these applications require a big computational cost, and they need optimized software and specialized mathematic libraries such as GGML.

GGML is a library used by machine learning algorithms and automatic learning that tries to execute efficiently operations with tensors on general purpose hardware. This library is quite new, being constantly modify, adapting to the changes in the models and the hardware. In this work we will try to improve it through the vectorial extensions available in x86 processors, such as AVX512 and AMX, that were not being used at the start of this work. We will evaluate the results with perf, using the library through applications such as llama.cpp or our own benchmarks.

Contenido

1. Introducción y motivación

1.1. Large Language Models

Los *Large Language Models (LLM)* utilizados en tareas de procesado de lenguaje, como *ChatGPT, Bard* o *Llama*, han representado un punto de inflexión en el potencial impacto de la AI en nuestra vida diaria. Estos modelos consisten en grandes grafos de cálculo de matrices de distintos tamaños, que transforman los datos de entrada(texto de entrada) para dar una salida(el lenguaje generado). Estas matrices se encuentran en tensores, que son simplemente contenedores para datos. Hasta ahora, los requerimientos computacionales de estos modelos forzaban a los procesos de entrenamiento e inferencia a la utilización de plataformas masivamente paralelas como las *GPUs*. Sin embargo, el empleo de mecanismos de optimización como la cuantificación de los pesos, que son los datos que componen estas redes neuronales, y funciones de activación de los modelos están consiguiendo reducciones drásticas en el coste computacional de los procesos de inferencia, haciendo al mismo tiempo a los procesadores de propósito general competitivos hasta cierto punto con las *GPUs*. En este contexto, el proyecto trabajará sobre una de las librerías de tensoresmás popular sobre la que se implementan modelos de LLM como *Llama2, Alpaca o GPT3*. Sobre dicha librería se propondrán mejoras en el uso de las extensiones de vectorización, a través de las funciones de tipo "*intrinsic*" para forzar al máximo el paralelismo de datos. Con la librería optimizada se analizarán las mejoras de rendimiento obtenidas por diferentes modelos de red neuronal.

1.2. Motivación del trabajo

Las mejoras de precisión obtenidas por las redes neuronales gracias a su creciente tamaño (número de pesos del modelo) ha permitido que *Large Laguage Models (LLM)* como *ChatGPT* [1], *Claude* [2] o *Google Bard* [3](recientemente renombrada a gemini) adquieran una potencia inimaginable hace unos años. En general, estos modelos no son de acceso público, excepto *Llama* [4] y Llama 2 [5], propiedad de la empresa Meta. Su carácter abierto ha generado un ecosistema de trabajo en torno a dichos *LLMs*, con la publicación de modelos optimizados en un tiempo asombrosamente corto. Así, en menos de un año desde el filtrado del primer modelo de *Llama*, encontramos modelos disponibles como *Alpaca 7B* [6] o *Vicuna-13B* [7], capaces de alcanzar niveles de precisión similares a sus competidores con modelos mucho más pequeños (*Alpaca 7B* es 26 veces más pequeño que *GPT-3*. Parte de las líneas de investigación surgidas en torno a estos modelos (poda, cuantificación) parecen haber acortado las diferencias de rendimiento entre los procesadores de propósito general y las GPU, haciendo de nuevo atractivo el uso de este tipo de hardware para labores de inferencia. Para su ejecución en *CPUs*, una de las librerías de tensores más utilizada para la implementación de estos modelos es *GGML* [8]. Con el objetivo de optimizar el rendimiento, esta librería ofrece soporte para operaciones vectorizadas. Debido a su reciente creación, el proceso de vectorización no es completo, y la librería no usa las extensiones de vectorización más recientes y parte de sus funciones siguen sin vectorizar. El objetivo de este trabajo es intentar completar el proceso de vectorización de la librería, actualizando las funciones ya vectorizadas e intentando vectorizar aquellas donde parezca razonable explotar el paralelismo a nivel de datos. El resto del documento esta organizada de la siguiente forma: en el capítulo 2, explicaremos la estructura y funcionamiento de la librería *GGML, e*n el capítulo 3, hablaremos sobre las extensiones vectoriales y su como se suelen usar, en el capítulo 4, hablaremos de como vamos a

vectorizar la librería *GGML* y algunas peculiaridades sobre las extensiones vectoriales, en el capítulo 5, vamos a analizar los resultados de las pruebas hechas sobre un *benchmark*, en el capítulo 6, vamos a analizar los resultados de las pruebas hechas sobre una red neuronal, y por último en el capítulo 7, hablaremos de las conclusiones sacadas y el trabajo futuro.

2. La librería ggml

GGML es una librería de operaciones con tensores (matrices multidimensionales) implementada inicialmente por Georgi Gerganov [9]. Su principal objetivo es llevar a cabo labores de inferencia en modelos de procesado de lenguaje populares, como LLAMA [10] o WHISPER [11]. Su código fuente, escrito en C y disponible públicamente [8], implementa la funcionalidad necesaria para su ejecución optimizada en procesadores de propósito general (de hecho, su implementación original tenía como objetivo su ejecución en el procesador M1 de Apple). Para ello, proporciona soporte a modelos con formatos de precisión reducida en sus pesos (flotantes de tamaño 16 bits, enteros de 8 bits e incluso modelos cuantizados a 4 o menos bits por peso) y sus operaciones básicas hacen uso activo de las extensiones vectoriales de los procesadores actuales.

En esta librería el trabajo con modelos basados en redes neuronales profundas se estructura en dos pasos. En una primera fase, la arquitectura del modelo se "traduce" a un grafo de cálculo, definiendo las operaciones que se realizarán a cada dato de entrada y posteriormente inicializando en memoria los tensores (reserva de espacio e inicialización de sus valores, también definidos como pesos). Una vez finalizada la implementación de la estructura de cálculo asociada al modelo, es necesario "solicitar" de manera explícita que se calcule el valor de salida para un valor de entrada dado. Las secciones 2.1 y 2.2 describen, a través de un ejemplo sencillo, las estructura y funciones que intervienen en los procesos de creación y cálculo, explicando con un poco más de detalle su funcionamiento. Para ilustrar dichos procesos usaremos un ejemplo concreto, correspondiente a la siguiente operación con tensores de dos dimensiones y 4 elementos:

$F(x)= A*x +C$

Para completar la descripción de la librería, en la sección 2.3 definimos las operaciones con tensores más relevantes implementadas en la librería y en la sección 2.4 analizamos un primer ejemplo para trabajar con una red neuronal sencilla.

Antes de detallar los procesos de creación y cálculo, es necesario definir las estructuras básicas sobre las que se construye la librería, así como la relación entre ellas. Se trata de tres elementos de tipo *struct* y de nombre contexto, objeto y tensor. En todos ellos uno de sus campos es una zona de memoria en la que se almacenan los datos de ese *struct* y el resto son metadatos relacionados con su función en la ejecución del grafo. En primer lugar, el contexto representa todo el contenido de un grafo de cálculo, incluyendo sus uniones, los datos con los que se va a operar y las operaciones a realizar. Todos los elementos que se definen dentro del grafo son de tipo objeto, y estos objetos pueden definir desde las relaciones entre elementos (qué tensores se operan entre sí y la ubicación del tensor resultado) hasta las operaciones o los propios tensores. Los objetos de tipo tensor serían los más relevantes, representando un contenedor de datos multidimensional.

2.1. Inicialización y creación del grafo

La estructura básica a cargo de la definición y uso del grafo de cálculo se denomina contexto. Los primeros pasos en la implementación de un modelo se llevan a cabo a través de la función *ggml_init*, con dos tareas principales, definición de parámetros para la ejecución (tamaño del modelo en memoria, puntero a la dirección del buffer donde se almacenarán los datos del modelo) e inicialización del contexto (función *ggml_init* Figura

2-1, lleva a cabo la reserva de una zona de memoria de tamaño adecuado para almacenar los metadatos, como por ejemplo el tipo de operación, y los datos de los tensores).

```
struct ggml_init_params params = {

    .mem_size   = 16*1024*1024,
    .mem_buffer = NULL,
};

struct ggml_context * ctx = ggml_init(params);
```

Figura 2-1 Código inicialización librería GGML

De esta fase de inicialización, cabe destacar la relevancia del campo *mem_size*. Indica la memoria necesaria para albergar los componentes del modelo: los tensores con sus pesos, el grafo de cálculo, las matrices intermedias de resultados y los metadatos asociados a estos. Una vez inicializado el contexto, el siguiente paso consistirá en definir e inicializar las estructuras vectoriales mencionadas, que contendrán los pesos de la red neuronal y los valores intermedios que se irán generando al operar nuestro grafo.

En el proceso de inicialización distinguiremos dos tipos de objeto tensor. En primer lugar, encontramos los tensores que solo almacenan datos, sin operación asociada. En nuestro ejemplo estos tensores serían los encargados de almacenar los valores de las matrices A, x y C. La matriz x en este caso contará como un dato de entrada de nuestro grafo. En segundo lugar, tenemos los tensores asociados a una operación, en cuyo caso almacenan qué tensores de datos constituyen sus operandos, así como un tensor resultado. De nuevo, en nuestro ejemplo este segundo tipo de tensor serían el que representa el resultado de multiplicar A y x (lo denominaremos *mul* en el resto del capítulo) y el resultado final del cálculo F.

En función de sus dimensiones, cada tensor de datos tiene una función de creación propia (por ejemplo, *ggml_new_tensor_3d* para matrices de 3 dimensiones), a la que se pasa como parámetros el tipo de dato, tamaño y contexto al que se asocia. De acuerdo con estos parámetros se calcula el tamaño que se necesita usar del buffer que hay en el contexto, y se crea un objeto (función *ggml_new_object*) de tipo tensor. La ubicación de cada nuevo objeto dentro del buffer (*mem_buffer* en Figura 2-1) se establece en este punto. Se añade a continuación del último objeto creado en el contexto, comprobando si el buffer tiene espacio suficiente para guardar el nuevo tensor. Cada objeto almacena en sus metadatos información correspondiente a su ubicación en el buffer (distancia hasta el siguiente objeto, offset desde la primera dirección del buffer) así como un puntero a la ubicación de los datos del tensor. Al finalizar la creación el nuevo objeto actuará como último en su contexto, informando de su posición al objeto anterior. Tras la creación de este objeto se crea una estructura del tensor para almacenar los datos, que empieza justo tras la estructura del objeto asociado al tensor. En este punto ya se han creado todas las estructuras necesarias para inicializar los datos del tensor. Continuando con nuestro ejemplo, la creación e inicialización de los tensores de datos A, x y C se describe en Figura 2-2.

```
struct ggml_tensor * a = ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 4);
struct ggml_tensor * x = ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 4);
struct ggml_tensor * c = ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 4);
for (int i = 0; i < 4; i++) {
      *(float*)((char *)a->data + a*b->nb[0])=2;
}
```

```
for (int i = 0; i < 4; i++) {f
      *(float*)((char *)x->data + i*x->nb[0])= input;
      //Input debería ser una entrada
}
for (int i = 0; i < 4; i++) {
      *(float*)((char *)c->data + i*c->nb[0])=4-i;
}
```

Figura 2-2 Código creación de tensores

Ilustraremos la creación de un tensor de operación con un ejemplo concreto, el producto escalar de matrices. La función *ggml_mul_mat*, que recibirá el contexto del grafo y los dos tensores operando, estará a cargo de la creación del objeto tensor (de una forma similar al caso anterior), en cuyos metadatos se incluirá el tipo de operación (GGML_OP_MUL_MAT) y los dos tensores operando. En este tipo de objeto la estructura de datos asociada al tensor se usará para almacenar el resultado de la operación (en este punto solo se reserva el espacio, no se lleva a cabo la operación). Cabe destacar que Los tensores operando pueden ser tanto tensores con operaciones como sin ella. En nuestro ejemplo habría que crear las operaciones de suma y multiplicación de la forma indicada en Figura 2-3.

```
struct ggml_tensor * mul = ggml_mul(ctx, a, x);
struct ggml_tensor * f  = ggml_add(ctx, mul, c);
```

Figura 2-3 Creación tensores de operaciones

Tras la inicialización de todos los objetos de tipo tensor en nuestro contexto, llega el momento de crear el grafo de cálculo, a través de la función *ggml_new_graph*. primero comprueba el tamaño que va a ocupar el grafo en función del tamaño que se le ha pasado. Tras esto crea un nuevo objeto correspondiente al grafo, de la misma manera en la que se crea un objeto correspondiente al tensor, la única diferencia entre estos dos objetos es simplemente el tipo. Tras crear el objeto se crea una estructura de tipo grafo, a la que damos valores. En esta estructura hay metadatos como el tamaño del grafo, y lo más importante una serie de punteros a *arrays* de tensores, donde se irán guardando en uno de ellos las hojas del grafo, y en otro los nodos. Todo lo antes mencionado es simplemente la función para crear la estructura de datos asociada al grafo, e inicializar los valores de esta misma. Para construir el grafo como tal, llamaremos a *ggml_build_forward_expand*, pasándole el grafo recién creado y el último nodo de nuestro grafo, es decir el resultado de todas las operaciones que se han llevado a cabo. Esta llama a *ggml_build_forward_impl* y esta a su vez llama a *ggml_visit_parents*, una función que se ira llamando a sí misma recursivamente para construir el grafo. Esta función recibe el grafo y un tensor. Esta es básicamente una función que empezando por el tensor pasado ira recorriendo los nodos llegando hasta las hojas del grafo, hasta haber recorrido todos y que estén apuntados en su respectivo array de la estructura asociada al grafo. La función va apuntando en una tabla hash los nodos/hojas ya visitados. Vamos a ver cómo sería la construcción del grafo de nuestro ejemplo empezaríamos con estas operaciones:

```
struct ggml_cgraph * grafo = ggml_new_graph(ctx);
ggml_build_forward_expand(grafo, f);
```

Figura 2-4 Código cración del grafo

La función *ggml_visit_parents* comienza por el nodo F. Comprobará si ya se ha visitado, y en caso de ser falso marcará dicho nodo como visitado (a través de una tabla hash). Tras esto, de forma recursiva comenzará a mirar los tensores base que serán operados, siendo

el primero de ellos el tensor mul, volviendo a ejecutar *ggml_visit_parent,* esta vez con mul. A continuación, se repite el mismo proceso con A. En este caso, al no tener A operandos asociados, se determina que es una hoja del grafo, se inserta en el array de hojas y no se invoca la función *ggml_visit_parent* para A. El primer grafo de la Figura 2-5 muestra la parte del grafo recorrido hasta este punto.

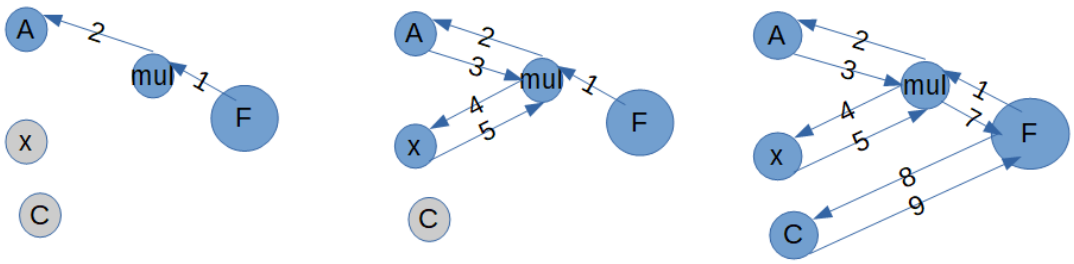

Figura 2-5 Ejemplo gráfico de cómo se va creando el grafo

Una vez terminado con el tensor A volvemos al tensor mul, y se repite para x la misma operación que con el tensor A. Una vez recorridos todos sus operandos, mul determina cual es la operación que realizar en este nodo del grafo y se apunta en el array de nodos. En este momento, como se muestra en el segundo grafo de la Figura 2-5, el grafo ha descubierto los nodos en este orden: F, mul, A, x, teniendo el array de hojas {A,x} y el de nodos {mul}.

Una vez colocado en su array, el tensor mul retorna el control al tensor F, que hace una nueva llamada a la función con se segundo operando, el tensor C. C sigue los mismos pasos que A y x, apuntándose en el array de hojas. Se regresa nuevamente al tensor F, que ya no tiene más operandos, por ello al igual que los otros nodos se apunta en el array de nodos, y retorna finalizando las llamadas recursivas. El descubrimiento del grafo finalizaría así, con el array de nodos {mul,F} y el de hojas {A,x,C}, habiendo descubierto el grafo como se ve en la Figura 2-5.

Como podemos ver con esta manera de generar las uniones entre nodos/hojas, los nodos quedan insertados en la lista en el orden que han de ser operados, primero mul y luego F. Cada tensor se inserta en su correspondiente lista una vez ya han retornado de las llamadas a la función todos sus operandos y justo antes de irse de ese nodo/hoja por última vez.

2.2. Cálculo con un grafo

Una vez construido nuestro grafo con todos sus nodos y hojas, es momento de dar valores de entrada para computar de acuerdo con el grafo. Esto se lleva a cabo a través de la función *ggml_graph_compute*, que dividirá las operaciones de cada nodo entre los *threads* definidos como parámetro. Siguiendo con nuestro ejemplo, vamos a ilustrar cómo se llevaría a cabo el reparto si usamos dos *threads*. La operación del nodo “mul”, usando vectores de 4 elementos, el primer *thread* será el encargado de multiplicar los dos primeros elementos de A por los dos primeros elementos de B, guardándolos en los dos primeros elementos de mul, mientras que el segundo hará lo mismo con los elementos restantes.

El tamaño de los datos a operar en cada nodo puede ser distinto, ya que el tamaño de las matrices en el grafo puede expandirse o contraerse. Por esta razón, el proceso de

paralelización en *threads* se lleva a cabo de manera independiente para cada nodo del grafo. En cada *thread* se va comprobando la operación que hay en cada nodo, y en función de esta se llama a la función que implementa la operación. Finalmente, el tensor del nodo almacena el resultado de la operación. Como los nodos son añadidos al grafo en el orden que se han de realizar las operaciones, los operandos siempre están listos cuando van a ser operados. En nuestro ejemplo, la finalización de la función *ggml_graph_compute* implica que se ha recorrido el grafo completo y el tensor F almacena el resultado de nuestros cálculos.

2.3. Operaciones básicas

En el apartado anterior hemos definido un grafo con una serie de operaciones muy sencillas, como multiplicación (elemento a elemento) o suma. Además de éstas, la librería implementa muchas más operaciones para permitir crear todo tipo de grafos y redes neuronales. En la Tabla 2-1 Operaciones más comunes GGML se enumeran únicamente las operaciones más habituales para esta tarea. Como describiremos en el capítulo 4, la optimización de estas funciones a través de las extensiones de vectorización del procesador ha sido el objetivo fundamental de este TFG.

Tabla 2-1 Operaciones más comunes GGML

Función	Operación
ggml_vec_add	Suma, elemento a elemento, de dos vectores
ggml_vec_mul	Multiplicación, elemento a elemento, de dos vectores
ggml_vec_div	División, elemento a elemento, de dos vectores
ggml_vec_sub	Resta, elemento a elemento, de dos vectores
ggml_vec_dot	Producto escalar de dos vectores, el resultado tendrá tantas filas, como columnas el primer operando y tantas columnas como columnas el segundo operando.
ggml_vec_sqrt	Raíz cuadrada de los elementos de un vector
ggml_vec_relu	Función de activación RELU de los elementos de un vector
ggml_vec_mad	Multiplica los elementos de un vector por un escalar y suma el resultado a otro vector.
ggml_vec_abs	Valor absoluto de los elementos de un vector

2.4. Ejemplo de una red neuronal, MNIST

Un ejemplo sencillo de red neuronal para ilustrar el uso de la librería *ggml* son las redes *Fully Connected* entrenadas para la identificación de dígitos en imágenes. En esta sección vamos a describir la implementación de una red con dos etapas *Fully Connected* entrenada con la base de datos MNIST [12] y con un formato de imagen de entrada de 28 x 28 *pixels* con una escala de grises representada por un valor flotante entre 0 y 1.

La arquitectura de la red, que se convertirá en ggml en un grafo de cálculo, se muestra en Figura 2-6. El dígito de entrada se etiqueta como CONST2, vector de 784 elementos. El resto de recuadros color verde son los pesos del modelo, que serán cargados desde un archivo que se pasa a nuestro programa. Como podemos ver primero empezaremos multiplicando la entrada por el peso CONST 1, luego se le sumara CONST3, al resultado de esto se le realizara la operación relu, y se repetirá los mismos pasos multiplicándole por CONST 0 y luego añadiéndole CONST 4, finalmente al resultado de esto se le aplicara la función de activación *soft_max*.

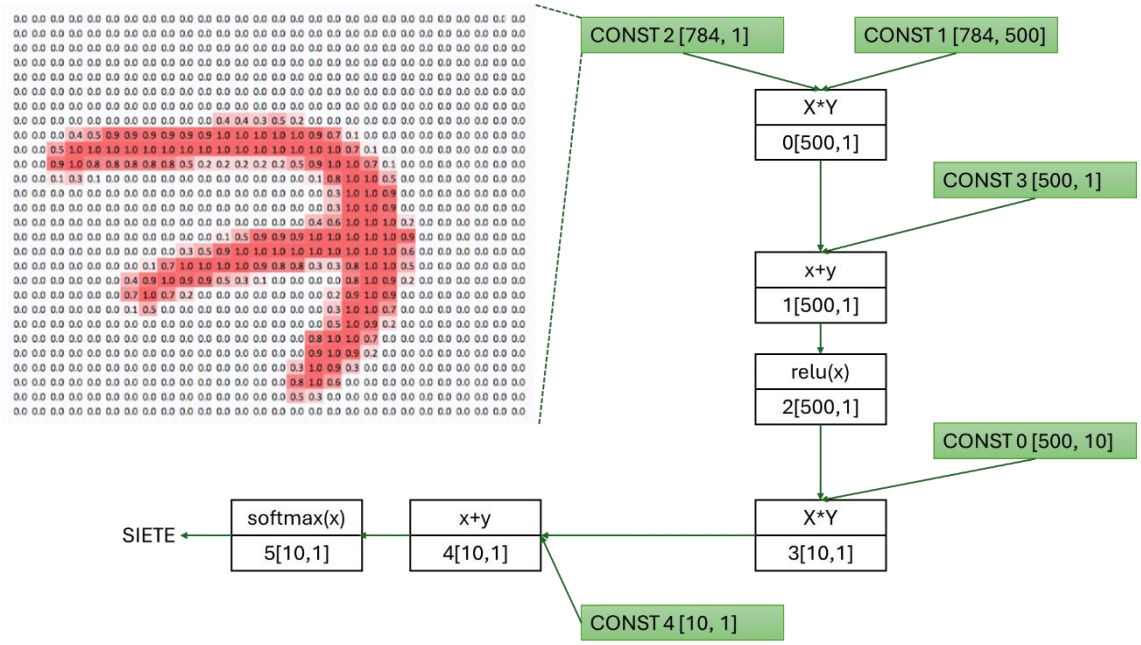

Figura 2-6 Red MNIST Fully Connected

Aquí podemos ver un ejemplo de código que define estas operaciones con sus respectivos pesos y dato de entrada:

```
ggml_tensor * fc1 = ggml_add(ctx0, ggml_mul_mat(ctx0, fc1_weight, input), fc1_bias);
ggml_tensor * fc2 = ggml_add(ctx0, ggml_mul_mat(ctx0, fc2_weight, ggml_relu(ctx0, fc1)),
fc2_bias);

ggml_tensor * probs = ggml_soft_max(ctx0, fc2);
```

Figura 2-7 Creación tensores operaciones grafo MNIST

Como podemos ver organiza las transformaciones de los datos de entrada en dos capas en las que en ambas se multiplica por un peso y se suma otro, finalizando con un *soft_max*. Las funciones *ggml_add, ggml_mul_mat* y *ggml_relu* crean tensores que operaran sus datos de entrada usando respectivamente las funciones antes explicadas de *ggml_vec_add, ggml_vec_dot* y *ggml_vec_relu*. La función *soft_max* realizara una transformación el vector convirtiendo el vector en una distribución de probabilidad. Es decir, esta función nos acabara retornando un vector en el que en cada una de sus posiciones tendremos la probabilidad de que sea ese número. Esto lo hace transformando valores reales en valores reales pero entre 0 y 1, quedando en la posición donde antes estaba el número más alto ahora también el número más alto pero entre 0 y 1. Otra aspecto a tener en cuenta es que la entrada es un vector de 784 elementos mientras que la salida es un vector de 10, podemos ver que estas transformaciones en el número de elementos las hace a través de los productos escalares, ya que da como resultado un tensor con tantas filas como columnas el primer operando y tantas columnas como columnas el segundo operando, hay que recordar que *mul_mat* multiplica fila por fila. Esta es una de las razones por las que se usa una entrada en tamaño vector para ir reduciendo el tamaño de la matriz que pasará al siguiente, y seguir conservando el peso de gran tamaño al inicio.

3. Vectorización en Procesadores de propósito general

3.1. SIMD, instrucciones vectoriales

Para aumentar las cotas de rendimiento, los procesadores han incorporado progresivamente técnicas de paralelismo de datos. Este modelo de cómputo, denominado SIMD (*Single Instruction Multiple Data*), consiste en realizar una misma instrucción sobre múltiples datos, una situación habitual cuando se trabaja con estructuras de datos vectoriales. El tipo de operando y operación, así como el número de datos simultáneos con el que podemos trabajar se conoce como "extensión vectorial", es dependiente de la arquitectura y ha evolucionado de manera significativa en los últimos 20 años. Las extensiones SSE, AVX y AMX descritas en este capítulo del documento corresponden a las disponibles es las arquitecturas x86_64 (AMD e Intel), que han sido las utilizadas en este TFG. Aunque no se incluyan en este documento, hay que indicar que hay más fabricantes que usan este tipo de extensiones. A modo de ejemplo, ARM tienen dos principales extensiones vectoriales, NEON (128 bits) y SVE (escalable de 128 a 2048).

La primera extensión vectorial en las arquitecturas x86 de Intel, denominada MMX [13], fue lanzada en el año 1996. En ella se reutilizaban los registros de 64 bits de la unidad de cálculo de punto flotante para almacenar enteros compactados (dos de 32 bits, cuatro de 16 bits u ocho de 8 bits) y operar con ellos simultáneamente. Esta extensión y todas las posteriores han sido soportadas por los dos grandes fabricantes de procesadores con arquitectura x86, Intel y AMD. Dos años después, Intel lanzó la primera versión de sus extensiones SSE [14][15]. En este caso, el tamaño del registro se incrementó hasta 128 bits y el procesador pasó a contar con un conjunto de registros propio (xmm0-xmm7) para las operaciones vectoriales. En su primera versión, el tipo de operando se limitó a flotantes de precisión simple, implementando instrucciones de movimiento de datos, conversión de datos, operaciones aritméticas y lógicas. SSE siguió evolucionando a través de tres versiones posteriores. En la primera actualización, SSE2 [15], se extendió el tipo de operandos a flotantes de doble precisión y enteros de 8, 16 y 32 bits mientras que las extensiones posteriores (SSE3 [15] y SSE4 [15]) introdujeron un conjunto importante de nuevas instrucciones.

En el año 2008, Intel remplazo las extensiones SSE con una nueva serie denominada AVX [16] (*Advanced Vector Extensions*), dando un salto importante y pasando a 16 registros (YMM0-YMM15) de 256 bits cada uno, que podía operar con punto flotante. Entre las novedades de AVX están la inclusión de operaciones no destructivas (el registro destino no tiene por qué ser uno de los registros origen) y permitir operandos no alineados en memoria. Adicionalmente, incluía nuevas operaciones como permutaciones o la fusión en una sola operación de la suma y la multiplicación (FMA). La segunda versión de AVX, denominada AVX2, completó la "traducción" de todo el repertorio de instrucciones de SSE a AVX, permitiendo operar en los registros de 256 con número enteros. Para continuar mejorando sus extensiones vectoriales, en la versión AVX512 se incrementó la cantidad (de 16 a 32) y la longitud (de 256 a 512) de los registros vectoriales. Otra de las mejoras en esta versión fue la inclusión de 8 nuevos registros para implementar operaciones enmascaradas, es decir, aquellas que no operan sobre todos los operandos que hay en el registro vectorial. En realidad, AVX512 es una etiqueta que designa un conjunto amplio de extensiones que ha crecido con cada nueva microarquitectura, cada una con su etiqueta propia (F, CD, ER, PF, VL, DQ, BW, IFMA, ...). Entre todas ellas, AVX512F es la extensión

estándar que implementa todas las funcionalidades básicas (ampliación de los registros vectoriales, operaciones enmascaradas o traducción de operaciones de versiones anteriores).

La última extensión SIMD disponible en los procesadores de Intel se denomina AMX (*Advanced Matrix Extensions*), un conjunto de instrucciones diseñadas para trabajar con matrices, orientado a aplicaciones del área de inteligencia artificial y machine learning. Esta extensión cuenta con ocho registros bidimensionales de 1 kilobyte cada uno. denominados *tiles*. De momento, en su primera versión la única operación disponible es la multiplicación escalar de matrices, orientado como hemos mencionado a acelerar la operación algebraica más común en la mayoría de aplicaciones basadas en redes neuronales. En el capítulo 6.5 desarrollaremos con más detalle esta extensión y cómo la hemos usado en la librería ggml.

El soporte hardware para la vectorización tiene dos componentes principales. Por un lado, necesitaremos un conjunto de registros del tamaño máximo del vector, con unas dimensiones mucho mayores que las de los tipos de datos convencionales. Adicionalmente, es necesaria la existencia de unidades funcionales que permitan operar con todos estos datos simultáneamente. En la Figura 3-1 podemos ver un esquema del *backend* de un procesador Intel con microarquitectura Cascade Lake con una extensión vectorial de 512 bits, concretamente de la parte de ejecución del procesador.

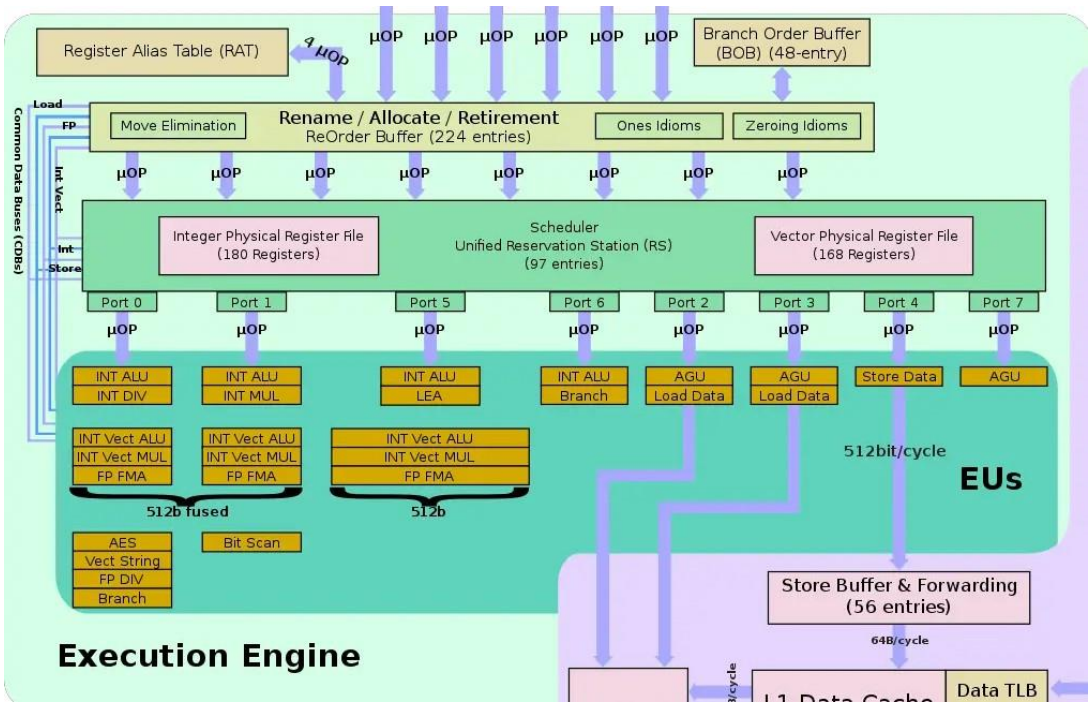

Figura 3-1 Execution Engine de la arquitectura Cascade Lake de Intel

Como podemos ver tenemos dos bancos de registros, uno correspondiente a los registros normales de 64 bits, y otro correspondiente a los registros vectoriales de 512 bits en este caso. Además, por un lado tenemos unidades funcionales capaces de manejar 64 bits y otras capaces de operar con 512 o 256 bits simultáneamente. Los procesadores Intel reutilizan las unidades funcionales para varios tipos de extensiones, permitiendo combinar varias de menor tamaño para crear una que pueda operar con operandos más grandes. Como podemos ver en la Figura 3-1 que hay dos unidades que pueden actuar como dos

unidades para instrucciones de 256 bits o juntarlas para que sea una unidad para instrucciones AVX512.

3.2. *Intrinsics* y vectorización de GGML

La utilización de los componentes vectoriales de un procesador (registros y unidades funcionales) requiere ejecutar las instrucciones apropiadas del repertorio de instrucciones del procesador (código ensamblador). En ocasiones, el proceso de compilación de códigos en lenguajes de alto nivel (C/C++) no es capaz de generar este tipo de instrucciones en ensamblador. En estos casos, una forma directa de forzar el uso de instrucciones vectoriales sería llevar a cabo un proceso de "*inlining*" en nuestro código, mezclando partes programadas en lenguajes de alto y bajo nivel. Este proceso es complejo y tedioso, y muchos programadores desconocen cómo se trabaja con instrucciones de tan bajo nivel.

```
__m512 _mm512_add_ps (__m512 a, __m512 b)
```

Figura 3-2 Intrinsic de suma de dos vectores AVX512

Afortunadamente, existen librerías disponibles para la mayoría de los compiladores actuales que a través de funciones con sintaxis de alto nivel proporcionan acceso directo a las instrucciones vectoriales del procesador. Dichas funciones reciben el nombre de *intrinsics*, y cada versión de AVX tiene su repertorio propio. Las funciones *intrinsics* [*17*], como la mostrada en Figura 3-2, proporcionan una API en C que da acceso a otras instrucciones en ensamblador, y por esta razón su sintaxis es relativamente compleja y autocontenida (el nombre de la función da pistas sobre aspectos clave de la vectorización). Los *intrinsics* de AVX512 tienen todos el siguiente formato:

Data _op_suffix(data_type param1, data_type param2, data_type param3)

Op representa la operación asociada al *intrinsic*, mientras que *suffix* nos indica qué tipo de datos se van a operar (flotantes, enteros...). Por ejemplo, para una operación con vectores de 256 se usará como tipo de dato *__m256* en el nombre de la función y *_mm256* acompañando al operando. Adicionalmente, se puede indicar el tipo de dato que contiene el vector mediante una letra (o la ausencia de ésta precediendo al tipo de dato). Por ejemplo, en el vector 512 bits, *_mm512* sería un vector que contiene flotantes de precisión simple, *_mm512d* flotantes de precisión doble y *_mm512i* enteros. Finalmente, algunos los *intrinsics* que lleven a cabo operaciones con máscara usarán tres operandos siendo este último un operando de un tipo no mencionado, *_mmask*.

4. Vectorización de la librería GGML

En este capítulo describiremos gran parte del trabajo de desarrollo de código realizado durante este TFG. Comenzaremos indicando el punto de partida de la librería (nivel de vectorización inicial) y explicaremos todas las aportaciones realizadas. El resultado final del trabajo está públicamente disponible mediante un repositorio [18]. Ggml no cuenta con documentación para desarrolladores y la organización del código es ligeramente caótica (un único fichero .c de más de 20000 líneas reúne gran parte de la funcionalidad de la librería). Aunque el volumen de cambios sobre el código original parezca pequeño, la comprensión sobre la estructura básica de la librería y la forma de adaptar las funciones a las nuevas extensiones vectoriales ha supuesto un gran esfuerzo.

La librería ggml, a pesar de su gran popularidad, solamente cuenta con unos meses de vida el primer reléase público(con funcionalidades muy simples) tenía menos de un año al comienzo de este TFG. Por esta razón, parte de sus funciones básicas de operación con tensores no soportan las extensiones de vectorización más recientes del procesador (AVX512) y muchas otras ni siquiera están vectorizadas. Debido al elevado coste computacional de los modelos de red neuronal que se implementan haciendo uso de este tipo de librerías, el trabajo de vectorización es muy relevante por las ventajas de rendimiento que se pueden obtener.

4.1. Punto de partida

Cuando se dio comienzo a este TFG (noviembre-2023), el último *commit* disponible en el repositorio de la librería solamente contaba con las operaciones vectorizadas que se indican en Tabla 4-1 Operaciones ya vectorizadas. El resto de las operaciones no estaban vectorizadas con ningún tipo de extensión. Las extensiones para las que existía vectorización eran NEON (procesadores ARM), power9-vector (procesadores POWER9 de IBM), y SSE3, AVX y AVX2 (procesadores x86-64 de Intel y AMD). Dentro de estas funciones, se hace uso de uno o varios *intrinsics*, que se esconden bajo los seudónimos de Tabla 4-2 Pseudonimos usados en las funciones ya vectorizadas. Todas estas operaciones se encontraban disponibles para dos tipos de datos distintos, flotantes de 32 bits y flotantes de 16 bits.

Tabla 4-1 Operaciones ya vectorizadas

ggml_vec_dot_f32	Producto escalar con flotantes de 32 bits
ggml_vec_dot_f16	Producto escalar con flotantes de 16 bits
ggml_vec_dot_f16_unroll	Producto escalar con flotantes de 16 bits, pero con un desenrollado de lazo más grande
ggml_vec_mad_f32	Multiplicación de vector por flotante y suma de otro vector
ggml_vec_mad_f32_unroll	Igual que el anterior pero con un desenrollado de lazo más grande
ggml_vec_scale_f32	Multiplica todos los elementos de un vector por un flotante

Tabla 4-2 Pseudonimos usados en las funciones ya vectorizadas

GGML_F32_VEC_ZERO	Llena un vector de ceros
GGML_F32_VEC_SET1	Llena un vector con un número

GGML_F32_VEC_LOAD	Carga datos de memoria en un vector
GGML_F32_VEC_STORE	Guarda datos de un vector en memoria
GGML_F32_VEC_FMA	Multiplica un vector por otro y le suma un tercer vector
GGML_F32_VEC_ADD	Suma dos vectores
GGML_F32_VEC_MUL	Multiplica dos vectores
GGML_F32_VEC_REDUCE*	Combina todos los elementos de 4 registros vectoriales retornando un solo flotante. Usada en *vec_dot*.

**REDUCE no esconde un único intrinsic debajo, si no varios de ellos que hacen esa operación.*

4.2. GGML y AVX-512

Nuestro trabajo se ha centrado en vectorizar las funciones que operan con flotantes de 32 bits. Este tipo de dato suele ser el más usado en los pesos de las redes neuronales, además de ser el dato usado en los modelos de llama.cpp [19], *framework* desarrollado sobre esta librería. A través del ejemplo en Figura 4-1, que muestra el código correspondiente a la operación de suma de vectores, vamos a describir la estructura de estas funciones de operaciones con vectores, que es siempre el mismo. Mediante macros creadas con la directiva *define*, el compilador comprueba si se ha habilitado algún tipo de vectorización, y si no es así el código que se genera para implementar la operación sobre el vector es de tipo secuencial (no hace uso de instrucciones vectoriales). Como se ve en Figura 4-1, en esta función el código de la parte vectorial está compuesto por macros, de forma que esta parte del código no necesita modificarse con cada nueva extensión vectorial que se añade. Debajo de estas instrucciones genéricas, a través de diferentes etiquetas ***define*** se hace uso de los *intrinsics* adecuados para cada tipo de procesador. La búsqueda de la extensión de vectorización adecuada (y por tanto de sus *intrinsics* específicos) se lleva a cabo (en compilación) de manera progresiva al principio del código de ggml.c. A través de directivas ***if defined*** se va comprobando si un tipo de extensión vectorial está disponible en el procesador. Si no se encuentra disponible se pasa al siguiente tipo, de una generación anterior al comprobado previamente. Para cada extensión se hace un mapeo específico de las funciones genéricas de operación a sus *intrinsics* correspondientes. El código de Figura 4-2 muestra cómo sería la cadena de defines para las extensiones vectoriales de un procesador Intel/AMD, así como el mapeo de las macros GGML_F32_VEC_LOAD, GGML_F32_VEC_STORE y GGML_F32_VEC_ADD.

```
inline static void ggml_vec_add_f32 (const int n, float * z, const
float * x, const float * y) {
#ifdef GGML_SIMD
    const int np = (n & ~(GGML_F32_STEP - 1));
    GGML_F32_VEC ax[GGML_F32_ARR];
    GGML_F32_VEC ay[GGML_F32_ARR];
    GGML_F32_VEC sum[GGML_F32_ARR];

    for (int i = 0; i < np; i += GGML_F32_STEP) {
        for (int j = 0; j < GGML_F32_ARR; j ++) {
            ax[j] = GGML_F32_VEC_LOAD(x + i + j * GGML_F32_EPR );
            ay[j] = GGML_F32_VEC_LOAD(y + i + j * GGML_F32_EPR);
            sum[j] = GGML_F32_VEC_ADD(ax[j], ay[j]);
            GGML_F32_VEC_STORE(z + i + j * GGML_F32_EPR, sum[j]);
        }
    }
```

```
    //leftovers
    for (int i = np; i < n; i++) {
        z[i]  = x[i] + y[i];
    }
#else
    for (int i = 0; i < n; ++i) z[i]  = x[i] + y[i];
#endif
}
```

Figura 4-1 Implementación de la función de suma

```
#if defined(__ARM_NEON) && defined(__ARM_FEATURE_FMA)
...
#elif defined(__AVX512F__)

#define GGML_SIMD
#define GGML_F32_STEP 64
#define GGML_F32_EPR  16

#define GGML_F32x16         __m512
#define GGML_F32x16_LOAD    _mm512_loadu_ps
#define GGML_F32x16_STORE   _mm512_storeu_ps
#define GGML_F32x16_ADD     _mm512_add_ps
...
#elif defined(__AVX__)

#define GGML_SIMD
#define GGML_F32_STEP 32
#define GGML_F32_EPR  8

#define GGML_F32x8          __m256
#define GGML_F32x8_LOAD     _mm256_loadu_ps
#define GGML_F32x8_STORE    _mm256_storeu_ps
#define GGML_F32x8_ADD      _mm256_add_ps
...
#elif defined(__POWER9_VECTOR__)
...
#elif defined(__wasm_simd128__)
...
#elif defined(__SSE3__)

#define GGML_SIMD
#define GGML_F32_STEP 32
#define GGML_F32_EPR  4

#define GGML_F32x4          __m128
#define GGML_F32x4_LOAD     _mm_loadu_ps
#define GGML_F32x4_STORE    _mm_storeu_ps
#define GGML_F32x4_ADD      _mm_add_ps
...
#elif defined(__loongarch_asx)
...
#elif defined(__loongarch_sx)
...
#endif
```

Figura 4-2 Definición de los intrinsics bajo seudónimos que serán usados

En general, las funciones vectorizadas presentan una estructura de doble bucle, que se ha mantenido para las nuevas implementaciones. El uso de estos dos bucles y el array es debido a que así se hace un desenrollado de lazo, para aprovechar todos los registros

vectoriales posibles, haciendo que el compilador no reutilice el mismo registro continuamente al usar los arrays. Para realizar la vectorización de las funciones lo hemos hecho en dos pasos. Primero ha habido que pasar las funciones ya vectorizadas en AVX a AVX512. Debido a que éstas ya usaban los seudónimos para realizar las operaciones, solo ha habido que añadir los *intrinsics* de AVX512 a sus correspondientes seudónimos, como se puede ver en la sección _AVX512F_ de la Figura 4-2.

El segundo paso tras actualizar todas las funciones AVX256 a AVX512, será vectorizar el resto de las funciones usando estos seudónimos. Un ejemplo de este trabajo se muestra en la Figura 4-1, que previamente a nuestro trabajo solo disponía de código en su versión serie (la parte tras el *endif*). En toda la vectorización se ha seguido el mismo formato sacado de las funciones ya vectorizadas, consistente en el uso de este doble bucle y arrays de registros vectoriales. En algunos casos, debido a que no existían previamente, ha habido que añadir nuevos seudónimos, como en el caso de la división y resta. Estos seudónimos están solo implementados en AVX y AVX512.

También hay que destacar que algunas operaciones como *sqrt*, o aquellas que usan mascara, es decir que en su implementación secuencial usan *if* a la hora de operar, han sido únicamente implementadas en AVX512, al ser la única que tiene instrucciones de máscara o de raíz cuadrada. Además en esta nueva extensión algunos seudónimos están definidos de forma más eficiente debido a que por ejemplo la operación reduce tiene una instrucción propia en AVX512, que no existía en sus predecesoras.

4.3. Peculiaridades trabajando con instrucciones vectoriales

Cuando se trabaja con instrucciones vectoriales, surgen algunos comportamientos imprevistos que hemos “sufrido” durante el desarrollo de este TFG. Intentamos describir aquí de manera breve nuestra experiencia pues la detección de las causas de ciertas anomalías en los resultados puede considerarse también parte del trabajo (al menos en tiempo de dedicación).

En primer lugar, debemos ser conscientes de la capacidad para auto vectorizar código del compilador con el que estemos trabajando. Con el compilador *GCC* versión 12.2.0-14, comprobamos que, con las opciones de optimización adecuadas, el compilador puede usar instrucciones vectoriales sencillas, correspondientes al repertorio de SSE3 (128 bits). Esto podría hacernos pensar que no es necesario vectorizar, ya que el compilador lo puede hacer automáticamente (aunque sea a 128 bits), pero en operaciones de cierta complejidad como el producto escalar (en el que en una línea hace multiplicación y acumulación) el compilador no sabe cómo vectorizarla y no la hace. El efecto del compilador sobre el rendimiento se describirá con detalle en el capítulo de pruebas y evaluación de los resultados.

Otro aspecto que considerar trabajando con instrucciones SIMD es la microarquitectura del procesador. Arquitecturas como Skylake [20] y Cascade Lake [21] presentan una implementación de la instrucción FMA (*Fused Multiply-Add*), la más habitual en el proceso de inferencia en redes neuronales. Existen dos Unidades Funcionales (Puertos 0 y 1) capaces de hacer operaciones FMA con vectores de 256 bits simultáneamente. Las operaciones sobre vectores de 512 bits se hacen “combinando” dichas unidades, lo que quiere decir que para esta operación el procesador tiene más unidades de ejecución de instrucciones AVX256 que AVX512. Por esta razón, mientras AVX512 generaría menos

instrucciones al operar más datos en cada una, AVX256 debería poder lanzar más instrucciones por ciclo que AVX512, por lo que las diferencias en rendimiento serán mínimas en este caso.

Al hacer pruebas con instrucciones vectoriales, otra peculiaridad es la limitación en frecuencia del procesador según el tipo de operación. debido a la propia arquitectura de la CPU, AVX256 y AVX512 no son capaces de alcanzar la misma frecuencia máxima. Esto es debido al escalado dinámico de frecuencia propio de los procesadores Intel, que explicaremos más en detalle al hablar de los resultados de nuestras pruebas.

4.4. Aceleración con la extensión AMX

Como se mencionó en el Capítulo 3, la última mejora en los procesadores Intel es la extensión AMX, en la que dejamos de hablar de vectorización para pasar a trabajar directamente con las matrices. Dicha extensión hace uso de 8 registros bidimensionales (denominados tiles), de un tamaño máximo de 16 filas con 64 bytes por fila. Para operar con estos registros disponemos únicamente de una unidad de multiplicación de tiles, enfocado hacia la inferencia en LLM, IA generativas o reconocimiento de imágenes. Actualmente AMX soporta dos tipos de datos, enteros de 8 bits y flotantes de 16 bits con formato BF16 [22]. Este último tipo de dato pensado para realizar la conversión desde flotantes de 32 bits más rápidamente.

El uso de esta extensión se hace a través de *intrinsics*, aunque hay muchos menos que en extensiones como AVX, ya que en cuanto a operaciones solo implementa el producto escalar, aparte las operaciones básicas de load/store. En Figura 4-3 se muestra el modo de uso de esta extensión. Como se puede observar, el uso de los tiles requiere un paso previo de configuración (número de filas y columnas del tile), tras el cual el proceso es relativamente sencillo, carga de datos de memoria *(_tile_loadd*), operación *(_tile_dpbssd)* y almacenamiento de resultados *(_tile_stored)*.

```
#define MAX 1024
#define MAX_ROWS 16
#define MAX_COLS 64
#define STRIDE 64
#define ARCH_GET_XCOMP_PERM     0x1022
#define ARCH_REQ_XCOMP_PERM     0x1023
#define XFEATURE_XTILECFG       17
#define XFEATURE_XTILEDATA      18

//Define tile config data structure
typedef struct __tile_config
{
  uint8_t palette_id;
  uint8_t start_row;
  uint8_t reserved_0[14];
  uint16_t colsb[16];
  uint8_t rows[16];
} __tilecfg;

int main(){

   __tilecfg tile_data = {0};

   // Request permission to linux kernel to run AMX
   syscall(SYS_arch_prctl, ARCH_REQ_XCOMP_PERM, XFEATURE_XTILEDATA)
```

```
   // Load tile configuration
   tileinfo->palette_id = 1;
   tileinfo->start_row = 0;
   //Matrix result
   for (int i = 0; i < 1; ++i)
   {
     tileinfo->colsb[i] = MAX_ROWS;
     tileinfo->rows[i] =  MAX_ROWS;
   }
   //Matrix operands
   for (int i = 1; i < 4; ++i)
   {
     tileinfo->colsb[i] = MAX_COLS;
     tileinfo->rows[i] =  MAX_ROWS;
   }
   //Loading tile config
   _tile_loadconfig (tileinfo);
   // Load tile rows from memory
   _tile_loadd (2, src1, STRIDE);//Src1 should be the first operand
   _tile_loadd (3, src2, STRIDE);//Src2 should be the second operand
   _tile_loadd (1, res, STRIDE);//Res should be full of ceros first
time
   // Compute dot-product of bytes in tiles
   _tile_dpbssd (1, 2, 3);
   // Store the tile data to memory
   _tile_stored (1, res, STRIDE);
   _tile_release ();
}
```

Figura 4-3 Código uso extensión AMX

Como todos los tiles de AMX tienen las mismas dimensiones (16x32 elementos de tipo BF16, 16x64 elementos de tipo int8), la multiplicación de dos tiles no se podría realizar porque el número de filas de un operando no coincide con el número de columnas del otro. Para solventar esto, en uno de los tiles se interpretan las posiciones en la matriz de una forma un tanto particular. Para ilustrar esta forma de operar utilizaremos el ejemplo de la Figura 4-3 y Figura 4-4, con matrices de tamaño 4x8 y tipo de dato BF16. En el segundo operando se considera que dos elementos adyacentes corresponden a la misma columna, por lo que la operación para el primer elemento del resultado (C[0][0]) implicaría a la primera fila del Tile 0 y a las DOS primeras columnas en el caso del Tile 1. En la Figura 4-4 se muestran las operaciones necesarias para calcular el primer elemento de la matriz resultado. En el caso de usar el tipo de dato int8 para la operación, la interpretación de Tile 1 cambia y los valores se agrupan de 4 en 4.

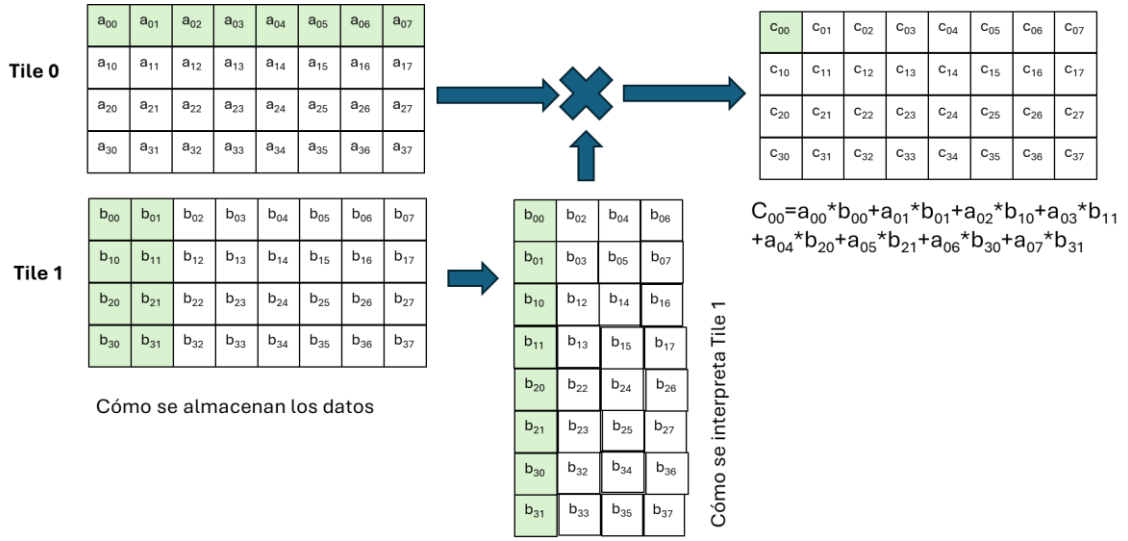

Figura 4-4 Colocación datos en memoria y como se interpretan en el Tile

Hay que considerar esta forma de operar al cargar los datos en los tiles. O bien definimos las estructuras matriciales teniendo en mente esta organización, o tendremos que reordenar si nuestras matrices en origen se almacenan en memoria de la forma convencional (ordenadas por fila, en posiciones contiguas de memoria).

Para su utilización en la librería ggml, hemos tenido que considerar la forma más adecuada de encajar la operación de multiplicación con AMX. Su uso de manera óptima requeriría cambios muy drásticos en el código, debido a aspectos como el reparto de los elementos de las matrices entre *threads* o el almacenamiento de matrices ya traspuestas para favorecer la localidad espacial en las operaciones de multiplicación. Por estas razones hemos optado por una implementación menos óptima pero más acotada en la cantidad de código a añadir y que permite compatibilidad total con el código ya existente.

Usaremos AMX para la operación FMA de dos vectores de gran tamaño, ya que ésta es capaz de realizar la multiplicación de un mayor número de elementos simultáneamente que usando un registro vectorial. Para ello hay que colocar correctamente los elementos de la segunda matriz para que se multipliquen cada uno por los elementos correspondientes de la primera matriz, es decir, teniendo en cuenta que los elementos van por pares como ya hemos explicado y que hay un máximo de 16 elementos por columna habría que colocar los 16 primeros pares del segundo vector en la primera columna, los 16 siguientes pares en la segunda columna y así sucesivamente.

Por tanto en la librería *ggml* hemos modificado la operación que realizaba el producto escalar de dos matrices *BF16*, repitiendo hasta completar todos los datos el hacer una transposición del segundo operando para tenerle colocado como nosotros queremos, lanzando la multiplicación, y finalmente sumando todos los elementos de la matriz resultado que se encuentran en la diagonal, ya que estos son los datos relevantes. El producto escalar es la única operación que hemos intentado mejorar con AMX, ya que es la única operación disponible en esta extensión. El resultado de la implementación hecha se encuentra en un repositorio que contiene la aplicación llama.cpp[29] con una versión de GGML que contiene aparte de esta implementación toda nuestra vectorización.

5. Evaluación, *Benchmark* Sintético

Para garantizar la correctitud de los cambios realizados sobre la librería, se ha implementado un banco de pruebas sencillo, cuyo objetivo es inicializar dos matrices, e ir llamando a diferentes funciones que generarán un grafo ggml para operar con una de las operaciones existentes en la librería. La operación que llevar a cabo y el tamaño de las matrices a operar son parametrizables por línea de comandos. Los cálculos se repiten 10 veces para recoger resultados promedio y evitar medidas que nos puedan falsear los resultados. En cada ejecución, las operaciones sobre las matrices se llevan a cabo de forma convencional (en serie) y a través de *intrinsics*. La comparación de resultados de ambas formas de operar garantiza en todo momento que los *intrinsics* seleccionados operan de manera adecuada. Para tomar las medidas de rendimiento se usa la librería PAPI [23], que proporciona un interfaz de funciones c para acceder directamente a los contadores de eventos hardware [24]. En Figura 5-1 proporcionamos un ejemplo de la implementación de una de las operaciones, en este caso la del producto escalar. El código del *benchmark* implementado se hace disponible públicamente a través de un repositorio de software [25].

```
void perform_vec_dot(struct ggml_context * ctx, struct ggml_tensor *
a, struct ggml_tensor * b, int size) {
    printf("Performing vec_dot operation\n");
    int event_set = PAPI_NULL;
    int times[iterations];
    int medium_time = 0;
    int start, end;

    struct ggml_tensor * vec_dot = ggml_mul_mat(ctx, a, b);
    struct ggml_cgraph * grafo = ggml_new_graph(ctx);

    ggml_build_forward_expand(grafo, vec_dot);

    printf("Starting compute\n");
    for (int i = 0; i < iterations; i++) {
        printf ("Iteration %d\n", i);
        start = PAPI_get_real_usec();
        ggml_graph_compute_with_ctx(ctx, grafo, 1);
        end = PAPI_get_real_usec();
        times[i] = end - start;
        printf("Time: %d microseconds\n", times[i]);
        medium_time += times[i];
    }


    printf("Compute finished\n");
    printf("Medium time: %d microseconds\n", medium_time/iterations);
}
```

Figura 5-1 Ejemplo de una función del benchmark

Para medir la mejora del nuevo código se compiló el código forzando un tipo de extensión vectorial (AVX256 frente a AVX512) o su ausencia, modificando los *CMakefile* usados en compilación de forma adecuada. Tanto en el caso de este *benchmark* como del *framework* del capítulo 6, las pruebas han sido realizadas sobre dos microarquitecturas de servidor

del fabricante Intel. En primer lugar, hemos utilizado un procesador Intel Xeon Silver4216 con 16 *cores* de microarquitectura Cascade Lake. En este procesador el número de operaciones FMA por ciclo es de una para AVX512 y 2 para AVX 256. El sistema opera a una frecuencia máxima de 3.2 GHz y cuenta con 128Gb de RAM. En segundo lugar, parte de las pruebas se han llevado a cabo en un procesador Intel Xeon Silver 4514Y. En este caso el número de operaciones FMA para AVX512 se incrementa a dos por ciclo, contando además este modelo con la extensión AMX. La frecuencia máxima y el tamaño de RAM son similares en este caso.

Primero tomaremos una medida con la ejecución serie, con AVX256 y AVX512, cada una de ellas con tres tamaños distintos de matriz, 1024, 2048 y 4096 todas ellas cuadradas. Además, para evitar que el *multithreading* afecte a los resultados de nuestras medidas, y nos den resultados que no se corresponden con la realidad, se ha ejecutado todo con un solo thread.

5.1. Resultados (Compilación O3)

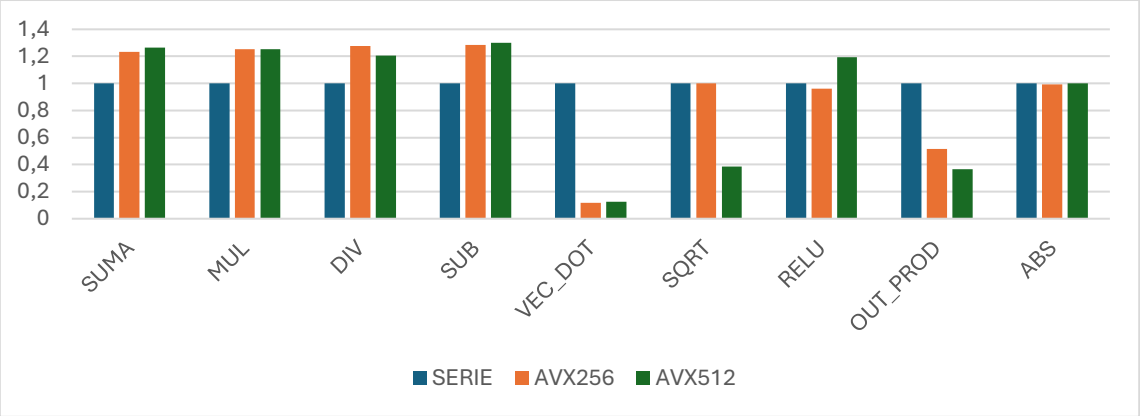

Figura 5-2 Tiempos distintas operaciones matriz 1024x1024

Figura 5-3 Tiempos distintas operaciones matriz 2048x2048

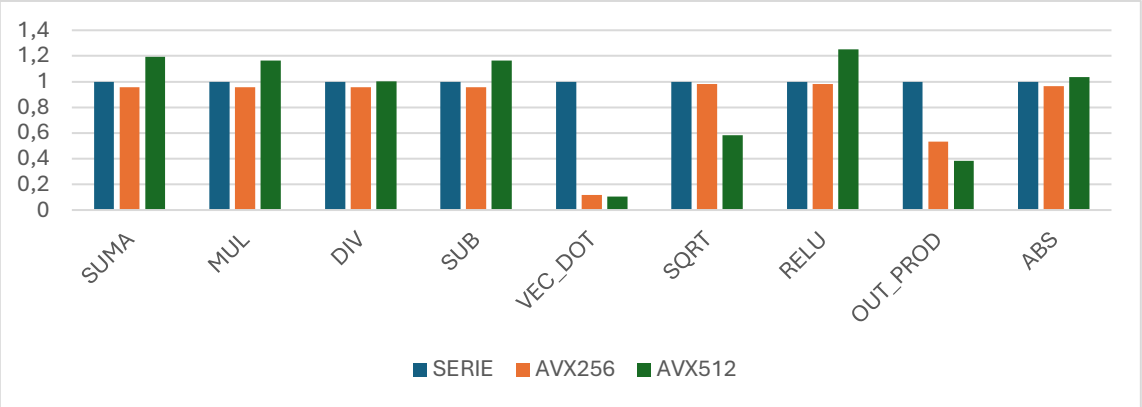

Figura 5-4 Tiempos distintas operaciones matriz 4096x4096

Empezaremos compilando nuestra librería usando las opciones por defecto. Al compilar esta librería, en este caso se tendrá desactivada el *flag* correspondiente a AVX512, lo que hemos tenido que activar, y al compilador se le indica usar el *flag* O3, el cual le indica que debe optimizar el código todo lo posible., incluso modificando el comportamiento del programa sobre el que podría indicar el código de alto nivel, ya sea cambiando operaciones por otras más eficientes, haciendo optimizaciones en lazos o introduciendo instrucciones que optimizan la ejecución. Esto último es una cosa que veremos al ver las instrucciones generadas. Los resultados obtenidos para los diferentes tamaños de matriz evaluados se muestran en las figuras Figura 5-2, Figura 5-3 y Figura 5-4. El eje vertical representa el tiempo de ejecución normalizado frente al valor SERIE, mientras que en el eje horizontal se muestran las diferentes operaciones vectorizadas.

Como podemos observar, en varios tipos de operación los resultados no responden en absoluto a las expectativas previas. No se observa ninguna mejora de las aplicaciones con el uso de instrucciones vectoriales sobre su versión serie, y en las operaciones básicas (suma, resta, multiplicación y división) la vectorización arroja un resultado peor. Solamente en las operaciones *VEC_DOT, SQRT y OUT_PROD* la vectorización mejora los tiempos de ejecución. La operación *SQRT* solo se implementa en AVX512, y por eso los resultados SERIE y AVX256 coinciden. En el caso de la operación *VEC_DOT*, AVX512 no es capaz de proporcionar ninguna mejora sobre AVX256. La explicación a este comportamiento se proporcionó en la sección 4.3, y tiene que ver con el número de unidades funcionales de cada tipo en la microarquitectura del procesador empleado. Con AVX256, el *scheduler* del procesador es capaz de planificar más instrucciones por ciclo al disponer de más unidades funcionales capaces de operar con ese tipo de vector.

Para tratar de encontrar una explicación a la falta de mejora de los tiempos de ejecución en algunos casos, vamos a analizar con más detalle el caso de la operación SUMA. Para ello, instrumentalizaremos el código con la librería PAPI [27] para medir el tipo y cantidad de operaciones sobre números flotantes que se observan en cada caso. Los resultados se muestran en la Figura 5-5, donde el eje vertical indica el número total de instrucciones y el horizontal las instrucciones en punto flotante de tipo vectorial y el total de instrucciones.

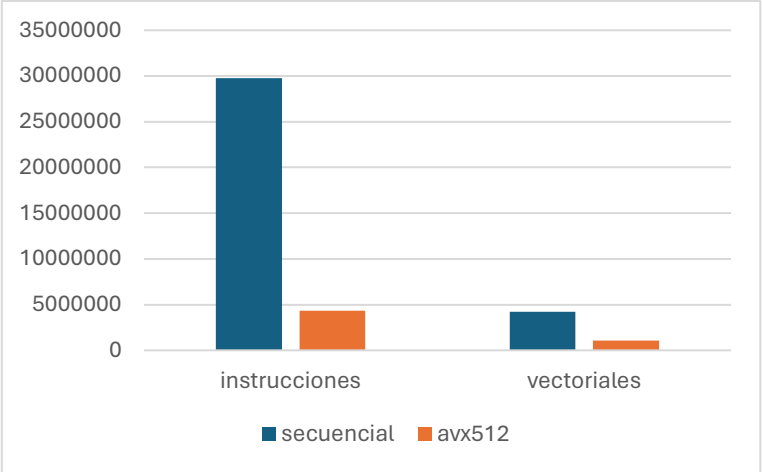

Figura 5-5 Comparación instrucciones totales y vectoriales entre secuencial y AVX512

Se observa que el código que usa las extensiones AVX512 genera menos instrucciones. Sin embargo, se aprecia un fenómeno extraño, que consiste en la aparición de instrucciones de tipo vectorial (concretamente 4 veces más que AVX512) en un código para el que no se hace uso de *intrinsics* (secuencial). Esto es debido a que el compilador es capaz de auto vectorizar parte del código, posiblemente limitado al uso de instrucciones SSE de tamaño 128 bits, de ahí que el número sea 4 veces mayor. Cabe destacar que, como indicábamos en la sección 4.3, el compilador es únicamente capaz de auto vectorizar operaciones muy básicas, como una suma o multiplicación, y esto no aplica a casos más complejos como el producto escalar. Si embargo, incluso con parte del código auto vectorizado, no acaba de parecer muy lógico que sigan tardando los mismos tiempos pues las extensiones vectoriales SSE deberían obtener peor rendimiento. Como se comentó en la sección 4.3, esto es debido al escalado en frecuencia que hace el procesador en función de las unidades vectoriales en uso. Desarrollaremos este aspecto relacionado con la frecuencia más adelante, en la sección 6.3.

5.2. Resultados (Compilación O0)

Para intentar evaluar las ventajas de la vectorización de manera más precisa, eliminaremos la capacidad del compilador para auto vectorizar, utilizando el nivel de optimización más bajo posible, en este caso con la opción O0. Con este cambio obtenemos los resultados que se muestran en Figura 5-6. Debido a la similitud en los resultados para los diferentes tamaños de matriz, en este apartado solo mostramos los valores obtenidos para una matriz de 4096.

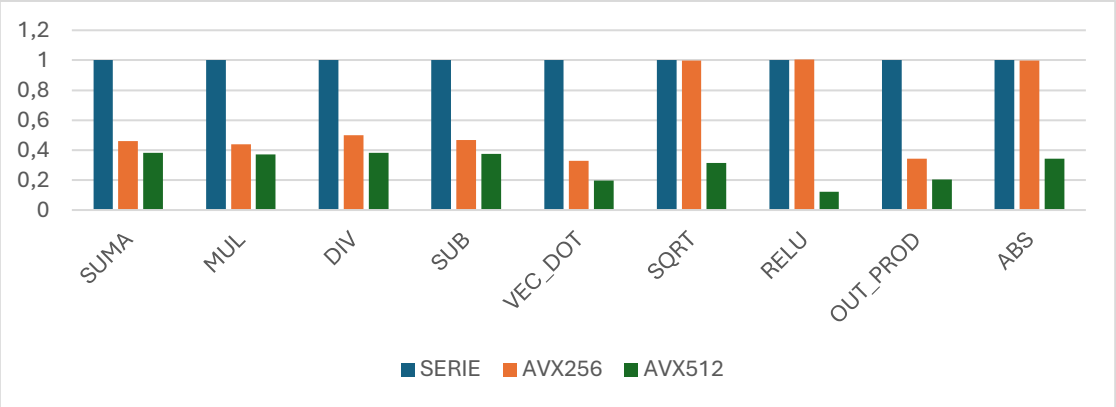

Figura 5-6 Tiempos distintas operaciones matriz 4096x4096

En este caso sí que podemos observar en comportamiento esperado a priori. Podemos ver hay una notable mejora en todos los tipos de operación. Aquí el principal cambio ha sido la no generación de instrucciones vectoriales por parte del procesador en la implementación serie, lo que ha hecho las operaciones vayan bastante más lentas. Del mismo modo que en el caso O3, la implementación serie pierde mucho respecto a AVX, y AVX256 está algo más parejo con AVX512, aunque en esta ejecución AVX512 obtiene mejor *speedup* sobre AVX256. Es posible que la falta total de optimización en el código no permita eliminar ciertas dependencias de datos, y por tanto 256 no pueda usar alguna de sus fortalezas frente a 512.

5.3. Aumentando las unidades funcionales

Como ya hemos hablado antes una de las posibles limitaciones que nos podíamos encontrar a la hora de usar *AVX512* sobre *AVX256* , era que normalmente los procesadores presentaban un menor número de unidades funcionales capaces de operar con 512 bits. Por tanto vamos a ver si conseguimos una mejora del IPC al usar un procesador con dos unidades funcionales de FMA de 512 bits, frente a otro con una sola. Para medir el IPC, vamos a lanzar diez operaciones de nuestro *benchmark* del tipo *vec_dot*, ya que esta es la que usa la operación FMA, y mediremos el total de instrucciones y ciclos en estas 10 operaciones. Primero mostraremos los resultados con una FMA, y luego con dos.

Instrucciones	Ciclos	IPC
145.378.686.669	140.006.671.838	1,04

Figura 5-7 IPC una FMA

Instrucciones	Ciclos	IPC
144,850,996,512	123,201,594,548	1,18

Figura 5-8 IPC dos FMAs

Podemos ver como el número de instrucciones en ambas ejecuciones permanece similar, los ciclos son bastantes más bajos en la *CPU* con dos *FMAs*, mejorando el IPC. Por lo que el número de unidades funcionales es importante debido a que puede contribuir a una mejora en los tiempos de ejecución a través de una disminución de los ciclos. Además en esta prueba también se ha podido ver una mejora en la frecuencia media a la que ejecuta el procesador con dos *FMAs*, debido a ser más nuevo y tener una implementación más eficiente energéticamente lo que desarrollaremos en el apartado 6.3, lo que contribuye también a la mejora de tiempos de ejecución.

6. Evaluación, Llama.cpp

En este apartado extenderemos las pruebas de rendimiento de la librería ggml cuando es utilizada como parte de un *framework* orientado a labores de inferencia para modelos de procesado de lenguaje. (*LLMs*). El elemento clave en la arquitectura de estos modelos se denomina *transformer* [27], cuya estructura de funcionamiento se muestra en la Figura 6-1. Los modelos LLM que vamos a ejecutar se construyen como una cadena de decodificadores (columna derecha en la figura) conectados entre sí de forma secuencial, cuya estructura se traduce a un grafo de cálculo ggml tal y como vimos en el capítulo 2.1. La operación principal en cada decodificador del modelo es el "*Multi-Head Attention*". Tanto esta operación como las partes de *Feed-Fordward* tienen una carga computacional elevada, con operaciones de producto escalar sobre matrices de gran tamaño.

Las pruebas de la vectorización de esta librería usando un LLM la haremos usando un *framework* del mismo creador que ggml.c, de nombre llama.cpp [19]. Se usará el modelo de 7B de pesos de LLaMa2 en su formato GGUF, un formato especial de archivo binario usado para cargar modelos en la librería GGML y guardar datos de estos mismos también. Debido al tamaño de los modelos, las pruebas se realizarán maximizando el uso del procesador, ejecutando un *thread* por *core* disponible (32 *threads* en nuestro caso). En el modo de ejecución utilizado se utiliza un *prompt* fijo (la frase de partida en nuestro caso ha sido: "The key to be succesfull is ") y se solicita al modelo que genere los próximos 200 tokens (aunque no es del todo preciso, un token se podría aproximar por una palabra). Igual que en el capítulo anterior, tomaremos medidas empleando diferentes tipos de vectorización en la librería ggml, y utilizaremos también distintos niveles de optimización en el compilador. Además, utilizaremos la herramienta de *profiling* de eventos hardware perf [28] para medir varios datos interesantes como frecuencia media, ciclos, instrucciones y operaciones de punto flotante secuenciales y vectoriales.

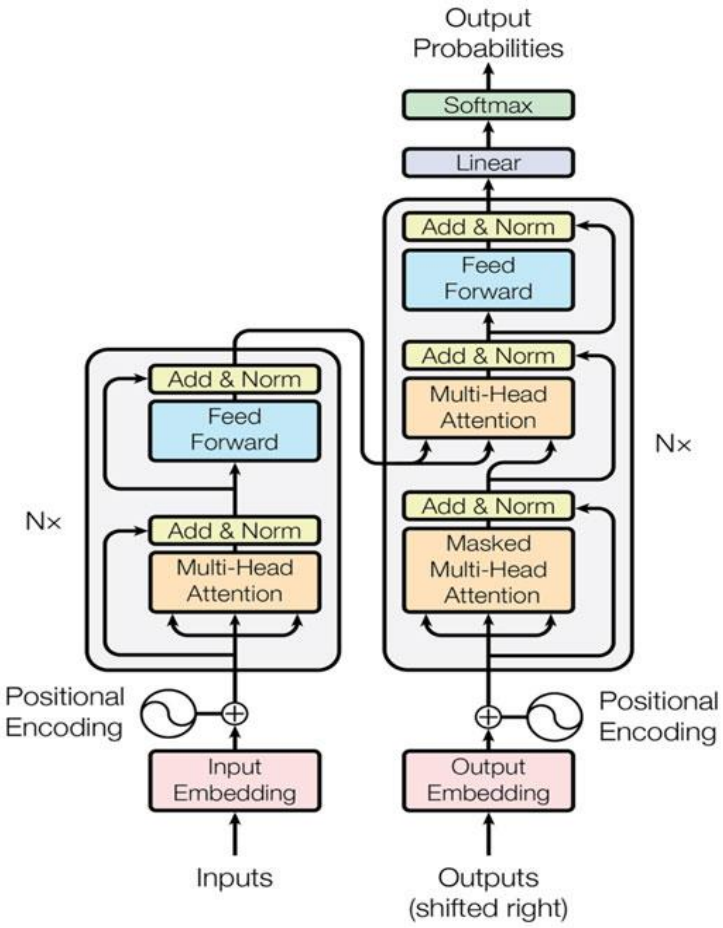

Figura 6-1 Arquitectura Transformer

6.1. Resultados (Compilación O0)

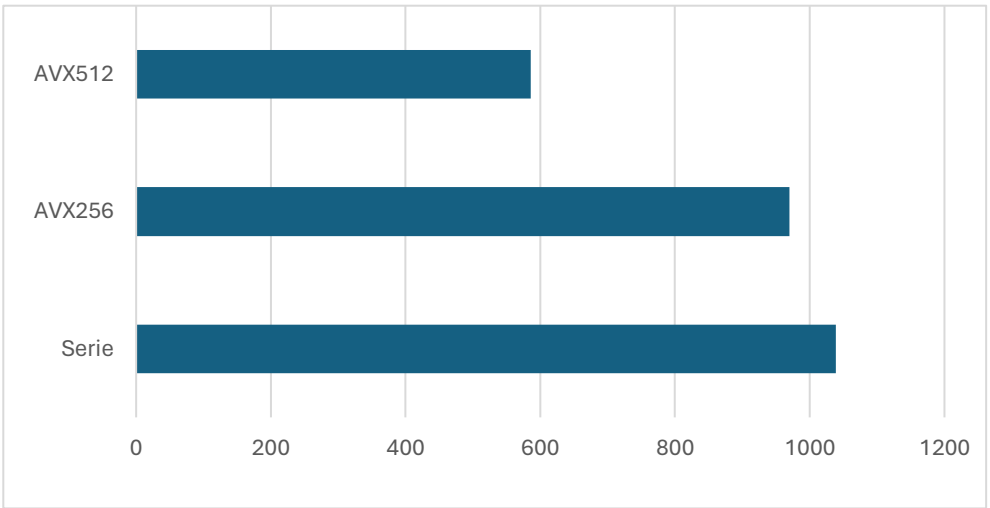

Figura 6-2 Tiempos por token(ms)

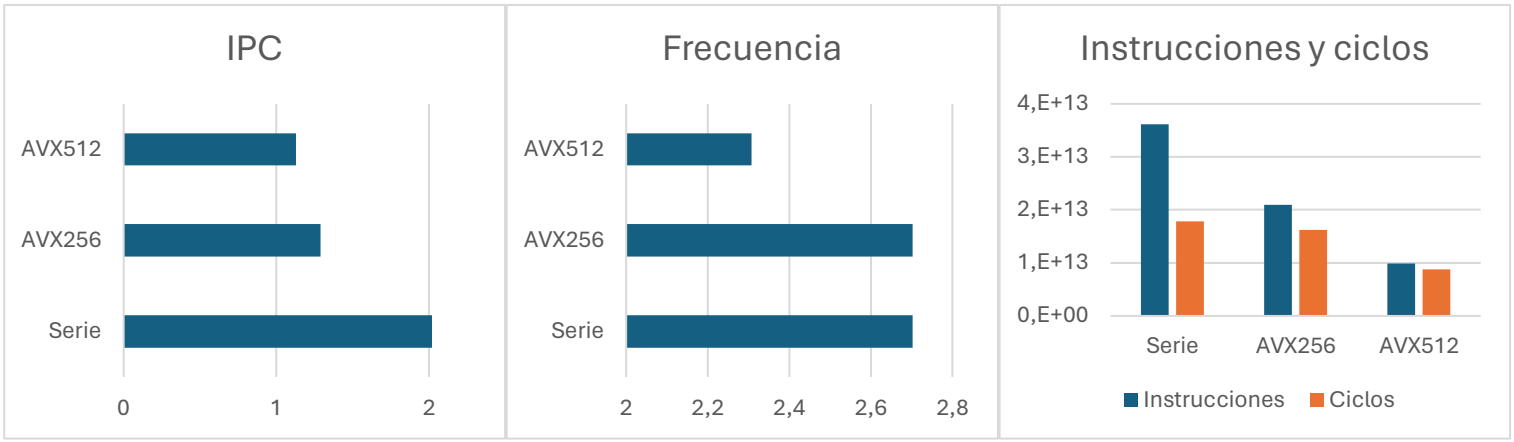

Figura 6-3 Comparativa ejecución secuencial, AVX256 y AVX512

La Figura 6-2 muestra los tiempos de predicción del siguiente token y la Figura 6-3 muestra los resultados devueltos por perf en la ejecución del código cuando el *framework* es compilado sin ningún tipo de optimización. Como podemos ver tenemos un claro ganador, la implementación en AVX512, que consigue *speedup* de casi un 65% sobre su implementación en AVX256, frente al 10% que obtiene AVX256 sobre la implementación sin vectorizar.

Puede resultar llamativo que el mejor valor de IPC se obtenga por la implementación serie como vemos en la Figura 6-3, pero hay que recordar que una cosa son instrucciones y otra bien distinta número de operaciones. Cada instrucción vectorial es capaz de ejecutar de manera simultánea un número elevado de operaciones (16 en el caso de AVX512). Las instrucciones no vectorizadas disponen de más unidades funcionales, por lo que su IPC será mayor, pero vectorizando ejecutamos muchas más operaciones por ciclo. Dado que AVX256 dispone de más unidades funcionales, llama la atención que no obtenga un valor de IPC superior a AVX512. Posiblemente la falta de optimización en compilación haga que AVX256 no utilice de forma óptima todas las unidades funcionales disponibles.

Evidentemente, como muestra Figura 6-3, AVX512 genera muchas menos instrucciones (9 billones frente a los 20 billones de AVX256 o los 36 billones de la secuencial). Gracias a esto y a pesar de operar a una frecuencia más baja que las otras dos (Figura 6-3), es capaz de ejecutar a mayor ritmo. Finalmente, destacaremos dos aspectos más que influyen en los resultados obtenidos. En primer lugar, cabe recordar que ciertas operaciones como la raíz cuadrada, u otra muy importante y usada bastante en redes neuronales como *relu*, solo están implementadas en la AVX512, otro motivo más para comprender las diferencias de rendimiento. En segundo lugar, la implementación hecha de la operación reduce (utilizada en la operación de producto escalar) para AVX512 es bastante más eficaz, y esta es una de las operaciones más usadas.

En resumen, AVX512 es bastante mejor en este caso debido a la buena implementación de su conjunto de instrucciones que incluye más instrucciones y mejor implementadas. No hay que olvidar que aunque veamos que el porcentaje de instrucciones vectoriales es muy pequeño respecto al total de instrucciones, el uso de instrucciones vectoriales de mayor tamaño también implica menos recorridos a un bucle, o menos accesos a memoria.

6.2. Resultados (Compilación O3)

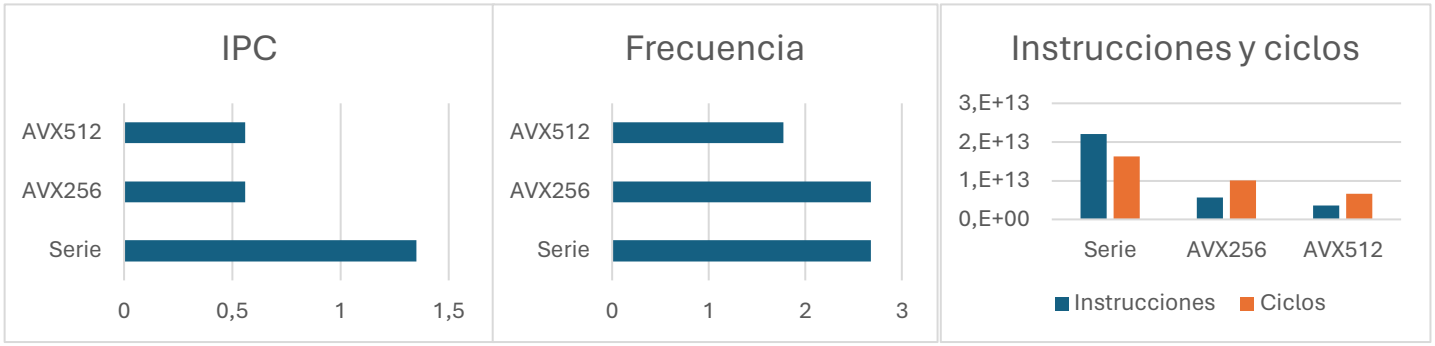

Figura 6-4 Comparativa secuencial, AVX256 y AVX512

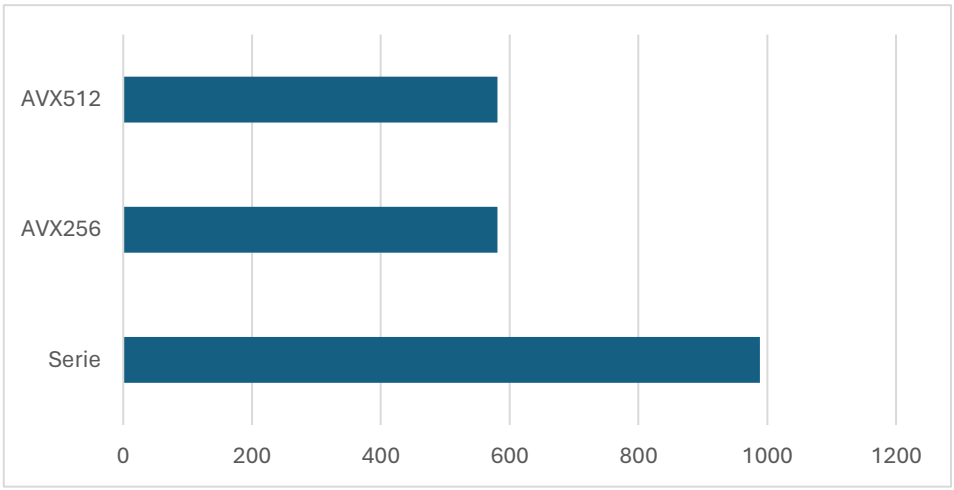

Figura 6-5 Tiempos por token(ms)

A continuación, probaremos la ejecución compilando y optimizando el código con O3, con los resultados mostrados en Figura 6-4 y Figura 6-5. Como vemos, en este caso AVX512 no es capaz de mejorar los tiempos de ejecución de AVX256 (Figura 6-5). De nuevo haremos uso de perf y los datos sobre eventos hardware para intentar buscar una explicación a este comportamiento. La primera conclusión a la que llegamos es que esto no se debe a un mejor uso de las unidades funcionales vectoriales por parte de AVX256, ya que en ambos casos se observa un IPC similar. Teniendo en cuenta que AVX256 tarda bastante más ciclos que AVX512, la igualdad de tiempo solo tiene una explicación, la frecuencia. Como podemos ver en Figura 6-4, mientras AVX256 mantiene su anterior frecuencia, 2,7GHz, pero mejorando el número de instrucciones, AVX512 baja su frecuencia considerablemente, a 1,8 GHz, mientras que también baja su número de instrucciones, 3 billones de instrucciones. Otro dato curioso es que genera muy pocas instrucciones vectoriales la ejecución secuencial sobre el total de operaciones en punto flotante, 79 millones de un billón, al contrario que las pruebas hechas en nuestro *microbenchmark*, debido a que la mayoría de las operaciones hechas en la red neuronal, son aquellas algo más complejas como *relu*, *gelu*, o el producto escalar que el compilador no sabe cómo vectorizar.

6.3. Optimización y escalado de frecuencia

El problema de la frecuencia máxima que alcanza AVX512 es inherente al propio procesador. Las extensiones vectoriales parecen ser muy ineficientes energéticamente

(AVX256 también, aunque en menor medida), por lo que Intel hace uso de técnicas de escalado de frecuencia dinámico para limitar la frecuencia máxima y reducir el consumo energético (y por tanto el calor generado). Este escalado es progresivo, y cuantas más de estas instrucciones estén en uso y en más *cores* menor frecuencia. Este comportamiento nos permite explicar muchos de los resultados de nuestro proceso de evaluación, como la mayor diferencia entre AVX256 y AVX512, al compilar con O0 y O3, como podemos ver tanto al usar Llama.cpp o nuestro propio *benchmark*.

Esto en nuestra librería se puede ver acentuado porque en el mismo código apliquemos técnicas como desenrollado de lazo o el uso de arrays de variables de tipo registro vectorial en lugar de una sola variable, que pretenden forzar a usar mayor cantidad de registros del procesador y unidades de ejecución por ciclo, tienen como contraparte que bajan la frecuencia al tener en uso más cantidad de procesador.

Es curioso observar cómo el escalado en frecuencia se vuelve más agresivo cuando el código está compilado de forma optimizada. Tras revisar el código y llevar a cabo alguna prueba adicional, hemos detectado que la optimización O3 maximiza el uso de los registros vectoriales, a través de técnicas como el desenrollado de lazo antes comentado, y por tanto incrementa el uso de las unidades vectoriales al permitir un mayor paralelismo, forzando a una bajada de frecuencia más pronunciada. La compilación O0 no se ve afectada por este fenómeno, debido a que por su falta de optimización, usa un menor número de registros vectoriales a la vez al no interpretar bien desenrollados de lazos. El código de la librería ggml nos permite aplicar ciertos parámetros que limitan el alcance de las optimizaciones. Es posible, y así lo hemos comprobado, limitar el número de registros vectoriales en uso para evitar optimizaciones habituales como el desenrollado de lazo. Es posible pensar que el uso de menos registros vectoriales es la solución, pues nos permitirá operar a mayor frecuencia, pero esto no se ve compensado por el incremento en el número de instrucciones.

En nuestro caso, el escalado en frecuencia explica de forma clara por qué el código optimizado y sin optimizar para AVX512 obtienen los mismos resultados, y por qué AVX512 no es capaz de batir de forma más clara a AVX256. Se trata en todos los casos de un problema que tiene que ver con la frecuencia de operación. Como esto es dependiente del procesador sobre el que trabajemos, en la sección 6.4 incluimos un análisis de rendimiento con frecuencias fijas para comprender cual es el margen de mejora de AVX512 si pudiese funcionar a la misma frecuencia.

6.4. Igualando la frecuencia

Para estas pruebas limitaremos la frecuencia máxima de todos los *cores* de nuestra máquina, limitándolos a algo menos de la frecuencia más baja que hemos visto en una ejecución de AVX512. Por ello pondremos la frecuencia a 1.6 GHz como máximo. Repetiremos todas las pruebas hechas anteriormente, es decir las tres implementaciones con ambas compilaciones. Los resultados obtenidos se muestran en las figuras Figura6-6 y Figura6-7. En estas no se muestran la frecuencia ya que es la misma en todas: 1.6 GHz.

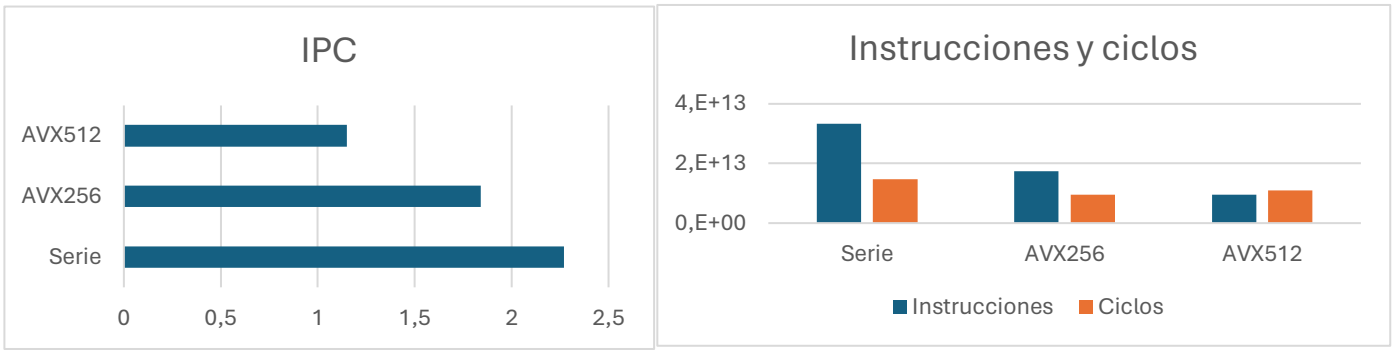

Figura 6-6 Comparativa secuencial, AVX256 y AVX512

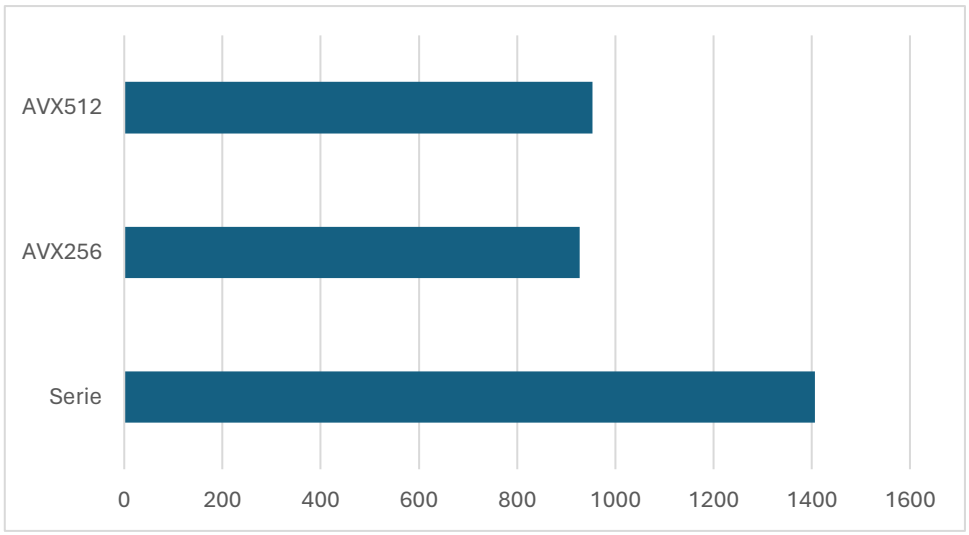

Figura 6-7 Tiempos por token (ms)

Es importante destacar que las medidas de perf en este caso no son del todo fiables y no las tendremos muy en cuenta , ya que por un lado puede estar afectado por el *multithreading*, y por el otro al cambiar la frecuencia puede afectar también estos datos, por ello aunque hay una disminución en el número de instrucciones respecto a la anterior ejecución, por ejemplo en AVX256 pasando de 20 billones a 17 billones, realmente no ha cambiado, básicamente no puede variar al ser el mismo binario. Esto además ha sido comprobado lanzando pruebas con un solo *thread* y se ve que no varía el número de instrucciones al cambiar la frecuencia. En estos resultados destaca que AVX256 obtiene un gran *speedup* sobre la ejecución secuencial, muy pareja a la que obtiene AVX512, contrario a lo que pasa a las ejecuciones con frecuencia dinámica, en la que AVX256 apenas obtiene *speedup* mientras AVX512 obtiene bastante. Es curioso que esto pase ya que todas ellas han tenido que bajar su frecuencia y deberían mantener sus diferencias. El motivo por el que AVX256 gana terreno a nuestra implementación en cuanto a *speedup* se debe a un incremento de su IPC, que es de 1,84 instrucciones por ciclo, y un mayor uso de sus unidades funcionales, acercando su *IPC* a la implementación secuencial, ya que esta y AVX512 mantiene su *IPC* igual**.** Una hipótesis podría ser el que el procesador estuviera permitiendo usar más de su capacidad al bajar su frecuencia, debido al escalado dinámico de frecuencia antes mencionado. Es decir que AVX256 pudiera usar más unidades funcionales de las disponibles en cada ciclo ya que al bajar la frecuencia a pesar de que en

cada ciclo se consume más, al estar más parte del procesador en uso, como cada ciclo más lento, el consumo por minuto permanece igual. De ahí se explicaría su mejor IPC respecto a cuando esta la frecuencia dinámica. AVX512 no se aprovecha de esto debido a su mayor ineficiencia energética y su menor número de unidades funcionales.

Ahora vamos a compilar con O3 nuevamente.

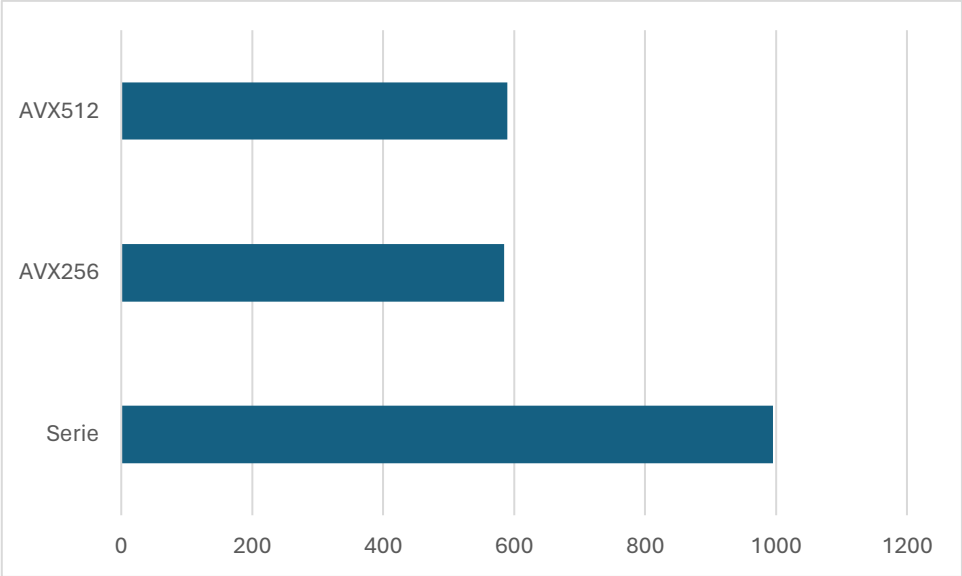

Figura 6-8 Tiempos por token(ms)

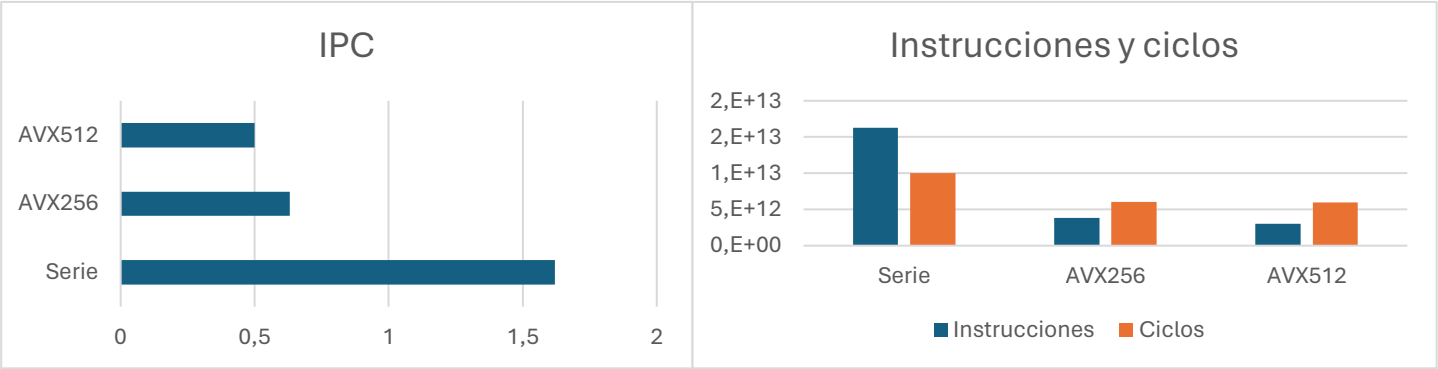

Figura 6-9 Comparativa secuencial, AVX256 y AVX512

Podemos ver que AVX512 mantiene su resultado anterior, ya que realmente él no ha bajado su frecuencia, no es capaz de ejecutar a más sin disparar el consumo. Aquí de nuevo es importante destacar que las medidas devueltas por perf no son muy fiables debido al cambio en frecuencias y *multithreading*. Como podemos ver tanto AVX256 como la ejecución secuencial no varían y mantiene su *speedup* respecto a AVX512 (a pesar de que una de las ventajas de AVX256 era correr a más frecuencia y al limitarla podría alejarse) a una mejora en su IPC, 1,62 en serie y 0,63 en AVX256, lo que hace que pase a ejecutar a menor frecuencia sea capaz de emitir más instrucciones por ciclo. Esto se podría deber a lo antes explicado de poder usar más unidades funcionales sin disparar el gasto energético, al reducir la frecuencia.

6.5. Implementación AMX

En este último apartado analizaremos las mejoras de rendimiento que somos capaces de obtener cuando la librería ggml hace uso de la extensión matricial AMX descrita en el capítulo 4.4. Las pruebas, de la misma forma que en la sección anterior, se harán ejecutando llama.cpp (200 *tokens*, 32 *threads*), esta vez sobre un procesador INTEL XEON SILVER 4514Y, cuya microarquitectura cuenta con una unidad de multiplicación de matrices (TMUL). Como esta unidad solamente puede operar con tipos muy concretos de datos (BF16 e int8), utilizaremos el tipo de dato BF16 para la operación de multiplicación de matrices, tanto en la implementación de ggml con AMX como en la implementación con AVX512. En estos modelos el resto de las operaciones se hacen en formato FP32, y es necesario un proceso de "traducción" a BF16 cada vez que se haga una multiplicación. Afortunadamente el *overhead* de traducción es muy pequeño, pues BF16 está diseñado en parte con ese objetivo. La comparativa de resultados de ejecución para ambas implementaciones se muestran en la Figura 6-10.

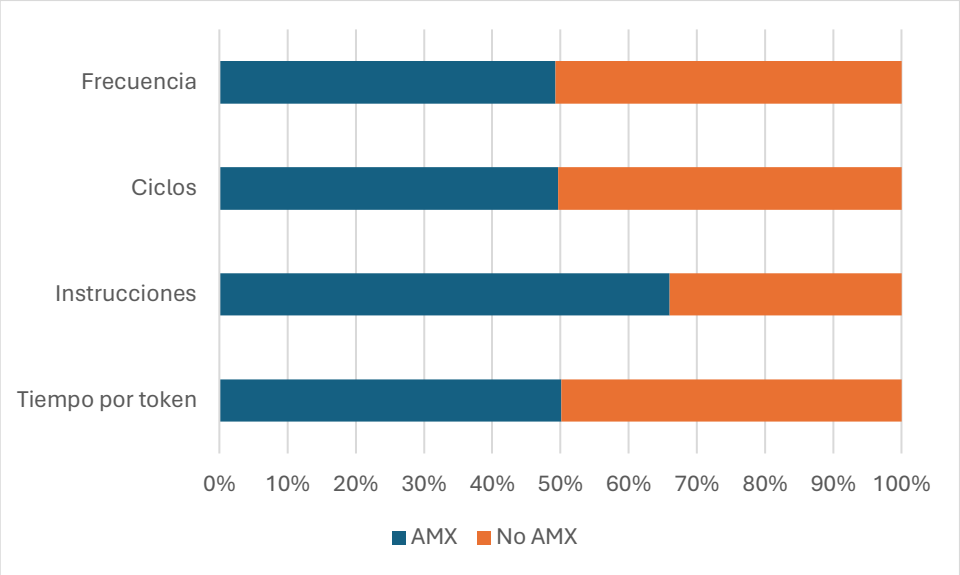

Figura 6-10 Comparativa resultados AMX y sin AMX

Como se puede ver el tiempo de generación de token es similar en ambos casos, y nuestra implementación AMX no mejora a la ya existente para AVX512 en ggml. Cabe destacar que el número de ciclos y frecuencia permanece similar en ambos casos, lo que significa que el escalado en frecuencia forzado por el procesador es similar para ambos tipos de vectorización. También es destacable que este modelo de procesador pueda operar con vectoriales a una frecuencia mayor que el usado en capítulos anteriores. Sin embargo, en el mismo número de ciclos la implementación con AMX ha ejecutado casi el doble de instrucciones que AVX512. Esto quiere decir dos cosas, la primera que el *overhead* introducido por el proceso de reordenación de matrices que se explicó en el Capítulo 4.4 es significativo, y la segunda es que gracias a que AMX es capaz de operar 16 veces más datos simultáneamente, conseguimos igualar los tiempos de ejecución.

Debido a la estructura de la librería ggml, la implementación de multiplicación que se ha podido hacer no ha sido la óptima. El reordenamiento de los vectores a operar, la suma de

la diagonal hecha al final o la ejecución secuencial del último resto de vector cuando no son múltiplos de 512 (en AVX512 en el peor de los casos serían 31 elementos, en AMX 511 a operar secuencialmente), impiden que AMX obtenga mejores resultados

A pesar de que no hemos conseguido más rendimiento, sacamos un aspecto muy favorable, y es que estos datos demuestran la gran efectividad de AMX, ya que aún a pesar de generar ejecutar casi el doble de instrucciones, el uso de AMX equilibra esta pérdida permitiéndonos que el tiempo de ejecución permanezca igual.

7. Conclusiones y Trabajo Futuro

En este trabajo, se ha conseguido implementar de manera satisfactoria las extensiones de AVX512 en la librería ggml que utiliza el *framework llama.cpp*. Igualmente, se ha hecho una implementación haciendo uso de las extensiones AMX presentes en computadores de última generación.

Sin embargo, el uso de las nuevas extensiones de Intel y su influencia en el rendimiento de Large Language Models como LLaMa, se ven influenciados por un número de factores propios de la arquitectura del procesador que estemos usando, como el número de unidades de cómputo para una determinada longitud de vector o el escalado de frecuencia dinámico propio de Intel.

La extensión AVX512 parece tener un buen funcionamiento llegando a dar una pequeña ganancia a pesar de todas las desventajas que puede tener, y en el peor de los casos no empeorando las otras extensiones y sus implementaciones. Hay que tener en cuenta, que su comportamiento se veía influenciado por el impacto del escalado de frecuencia del procesador, cuyo efecto se veía incrementado con las opciones de optimización.

La implementación de la extensión AMX ha habido que adaptarla al funcionamiento de la librería ggml. En cualquier caso, a pesar de no haberla podido implementar de forma óptima, parece que las extensiones AMX funcionan considerablemente mejor que sus predecesoras, consiguiendo, incluso con el aumento de instrucciones impuesto por la adaptación, rendimientos semejantes a la implementación vectorial existente.

Siguiendo esta última línea, una de las principales líneas de trabajo futuro en el desarrollo de esta librería debiera ser la implementación óptima de la extensión AMX. Es decir, pasando a dividir los tensores a multiplicar en matrices más pequeñas, e ir operándolas tras distribuirlas a los threads. Esto implica un gran cambio en el funcionamiento de la librería, pero podría dar una gran ganancia respecto a el uso de las otras extensiones vectoriales. Otra línea de trabajo a desarrollar podría ser la vectorización de las funciones *quantizadas*, que en esta librería solo está soportado parcialmente, como el producto escalar, pero nuevamente con un tamaño de vector de 256 bits. Así, se podrían seguir obteniendo ganancias usando la mayor potencia y variedad de operaciones de AVX512, e incluso usando AMX.

8. Bibliografía

[1] OpenAI. *ChatGPT (Version 3-5)*. Obtenido de https://chatgpt.com/

[2] Anthropic Claude (2023) https://claude.ai

[3] Google Gemini (2024) https://gemini.google.com/app

[4] Hugo Touvron, et. al. "LLaMA: Open and Efficient Foundation Language Models" https://arxiv.org/abs/2302.13971v1

[5] Hugo Touvron, et. al. "Llama 2: Open Foundation and Fine-Tuned Chat Models" https://arxiv.org/abs/2307.09288

[6] Rohan Taori, et. al. "Stanford Alpaca: An Instruction-following LLaMA model", https://github.com/tatsu-lab/stanford_alpaca

[7] Chiang, Wei-Lin, et. al. "Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90\%* ChatGPT Quality", https://lmsys.org/blog/2023-03-30-vicuna/

[8] https://github.com/ggerganov/ggml

[9]Gerganov official page https://ggerganov.com/

[10] AIMeta. Introducing LLaMA: A foundational, 65-billion-parameter large language model https://ai.meta.com/blog/large-language-model-llama-meta-ai/

[11] OpenAI. Introducing Whisper. https://openai.com/index/whisper/

[12] MNIST Database of Handwritten digits https://www.kaggle.com/datasets/hojjatk/mnist-dataset/data

[13] ¿Que es MMX? https://www.geeksforgeeks.org/what-is-mmx-multimedia-extension/

[14]Intel. Details about Intel Streaming SIMD extension intrinsics https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/details-about-sse-intrinsics.html

[15]Intel. Tecnología Intel Instruction Set Extensions https://www.intel.la/content/www/xl/es/support/articles/000005779/processors.html

[16]Intel. Intel® C++ Compiler Classic Developer Guide and Reference https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/details-of-avx-intrinsics.html

[17]Intel Intrincs Official Guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

[18]Repositorio versión final del trabajo(Última revision 12/06/2024) https://github.com/Sergiomartinez23/ggmlSergio

[19] https://github.com/ggerganov/llama.cpp

[20] https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

[21] https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake

[22] Google. El formato númerico bfloat16.
https://cloud.google.com/tpu/docs/bfloat16?hl=es-419

[23] https://github.com/icl-utk-edu/papi/wiki/

[24] Dr Robert N. M. Watson 2020-2021 Advanced operating system: Hardware performance counters

https://www.cl.cam.ac.uk/teaching/2021/L41/2020-2021-l41-hwpmc.pdf

[25] https://github.com/Sergiomartinez23/ggmlSergio/tree/master/examples/benchmarks

[26] https://github.com/icl-utk-edu/papi

[27]Machine Learning Mastery. The transformer model.
https://machinelearningmastery.com/the-transformer-model/

[28] “perf: linux profiling with performance counters.” [Online]. Available:

https://perf.wiki.kernel.org/.

[29] Versión final ggml incluida en llama(última revisión 12/06/2024)
https://github.com/sergio123456789123456789/llamaAMX.cpp