DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

Generación de código C++ optimizado para modelos de redes neuronales ONNX

Organización del Computador II
2023

Integrante	LU	Correo electrónico
Martín Emiliano Lombardo	49/20	mlombardo9@gmail.com
Ignacio Ezequiel Vigilante	61/20	nachovigilante@gmail.com
Tomás Spognardi	362/20	tomas.spognardi@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

Usualmente, a la hora de realizar una inferencia sobre una red neuronal, lo más conveniente es hacer uso de las ventajas que presenta la GPU, ya que los cálculos involucrados en esta tarea suelen verse muy beneficiados por el poder de paralelización de las mismas. Sin embargo, existen situaciones particulares (como al trabajar en sistemas embebidos) donde la utilización de la GPU se dificulta o los requerimientos no lo permiten. En ese entonces surge la necesidad de realizar dicha inferencia en la CPU, lo que resulta en una pérdida considerable de eficiencia debido a que no es posible alcanzar el mismo nivel de paralelismo.

En este contexto, surge la iniciativa de buscar maneras de implementar distintas operaciones para la inferencia de redes neuronales en la CPU de la forma más eficiente posible[1]. En el presente informe describiremos el funcionamiento de nuestra librería, `onnx2code`. Ésta es capaz de, dada una red neuronal, generar el código necesario para realizar la inferencia de la misma en lenguaje C++, el cual se puede compilar para utilizar en la arquitectura deseada. Haremos un enfoque especial en GEMM (General Matrix Multiplication), porque es principalmente donde realizamos el uso de técnicas y conocimientos relacionados a la materia.

Finalmente, compararemos la eficiencia de `onnx2code` con `tensorflow` y `onnxruntime`, dos runtimes de *machine learning* reconocidos, a la hora de evaluar distintos operadores y modelos.

1.1. ONNX

La unidad de trabajo de `onnx2code` es el **modelo**. Se trata de un grafo dirigido acíclico (DAG) donde los nodos representan distintos tipos de **operadores**, y las aristas indican que el resultado de un operador se debe utilizar como operando en otro. Por ejemplo, la siguiente imagen es una ilustración de una sección de INCEPTIONV1[2], la cual recibe una entrada a la que aplica MAXPOOL, y realiza un conjunto de convoluciones y activaciones, para luego concatenar el resultado y seguir utilizándolo en el modelo.

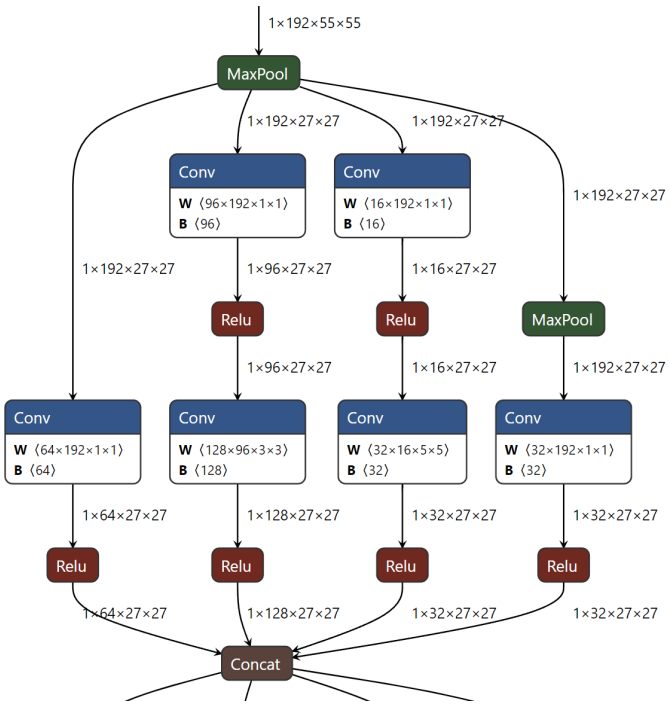

Figura 1: Extracto del modelo de INCEPTIONV1 renderizado con Netron

[1] En particular, nos vamos a restringir al uso de un sólo núcleo.

[2] «Inception» es la arquitectura de la red neuronal GOOGLENET, presentada en un paper en 2014. INCEPTIONV1 es una reproducción de la primera versión de esta red.

Estos modelos se pueden representar a través de distintos formatos. Para nuestra librería, elegimos ONNX (Open Neural Network Exchange), debido a que es sumamente interoperable: los formatos de múltiples librerías pueden ser convertidos a ONNX con facilidad, y viceversa. Esto no implica que la librería pueda utilizar cualquier modelo representable en ONNX ya que, para realizar la inferencia de uno, es necesario implementar cada uno de sus operadores.

2. Trabajo preliminar

Como no es factible implementar los ∼200 operadores que ofrece el formato ONNX, debemos elegir un subconjunto que nos permita soportar la mayor cantidad de modelos posible. Para ello, tomamos todos los modelos que se encuentran en el *ONNX Model Zoo* [6] y medimos la prevalencia de cada operador. Utilizamos esta información para seleccionar un conjunto abarcativo de operadores.

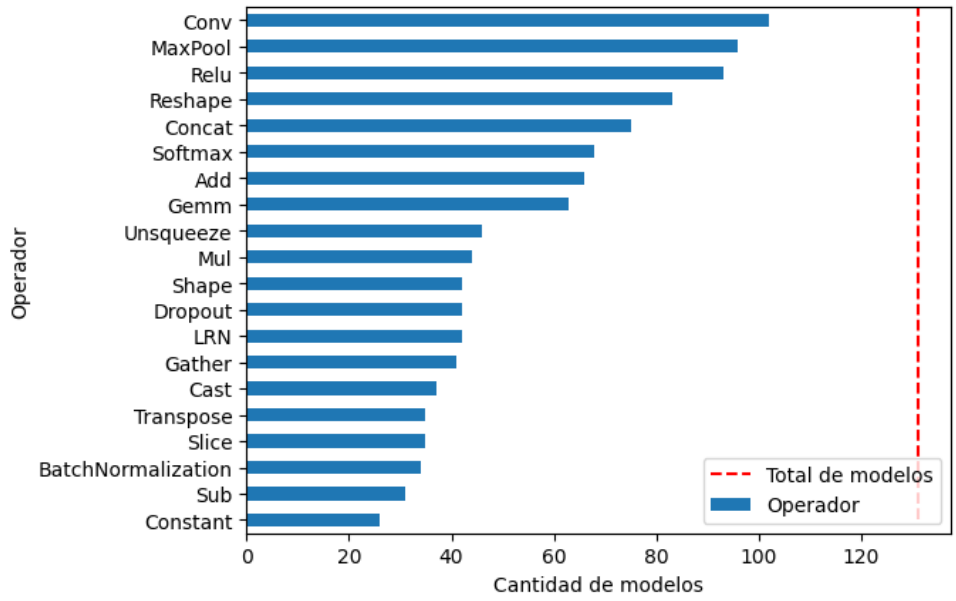

Figura 2: Cantidad de modelos en los que está contenido cada operador dentro del *Model Zoo* [6].

Como podemos ver en la figura, el operador más utilizado en los modelos es CONV, la convolución. Esto se debe a que es el componente principal de las CNNs (Convolutional Neural Networks), utilizado en el reconocimiento y procesamiento de imágenes. Más adelante, veremos que la convolución se puede expresar como una multiplicación de matrices, lo cual aumenta la importancia de tener una implementación eficiente de GEMM.

Analizando estos resultados, elegimos un conjunto de operadores, teniendo en cuenta para cada uno su prevalencia y la dificultad que conllevaría implementarlo.

3. Implementación

Como explicamos en la introducción, `onnx2code` recibe un modelo (en formato ONNX) y devuelve un archivo de C++ con una función `inference` que, como su nombre indica, realiza la inferencia del mismo. El siguiente es un ejemplo del código generado:

```
float intermediates[786432];

void inference(const float* weights, const float* inputs, float* outputs) {
    const float* T0 = inputs + 0;         // (1x512x512) input_2
    const float* T1 = inputs + 262144;    // (1x512x512) input_3
    float* T2 = outputs + 0;              // (1x512x512) add
    float* T4 = intermediates + 0;        // (512x512) model/dense_1/Tensordot/MatMul:0
    float* T6 = intermediates + 262144;   // (1x512x512) model/dense_1/Relu:0
    float* T8 = intermediates + 0;        // (512x512) model/dense/Tensordot/MatMul:0
    float* T10 = intermediates + 524288; // (1x512x512) model/dense/Relu:0
    const float* T12 = weights + 0;       // (512x512) model/dense_1/Tensordot/ReadVariableOp:0
    const float* T13 = weights + 262144; // (512x512) model/dense/Tensordot/ReadVariableOp:0

    GEMM_False_512_512_512_False(T1, T12, T4);
    Relu_262144(T4, T6);
    GEMM_False_512_512_512_False(T0, T13, T8);
    Relu_262144(T8, T10);
    Add_1x512x512_1x512x512(T10, T6, T2);
}
```

Figura 3: Código generado para un modelo que toma como entradas 2 matrices de tamaño 512×512, multiplica cada una por otra matriz, les aplica una función de activación (ReLU), y produce la suma entre ambos resultados.

En el código, podemos ver que en la primera parte se declara una serie de variables que se utilizarán como operandos en las funciones de la inferencia. Éstas son punteros que contienen *tensores*, objetos matemáticos que generalizan la noción de matriz y vector[3]. Los comentarios que están a la derecha de cada declaración indican las dimensiones de los tensores, junto con el nombre que toman en el archivo ONNX, a modo de referencia.

A pesar de que el modelo toma múltiples entradas, `inference` recibe un solo parámetro `inputs`, que debe apuntar a un bloque de memoria que contenga, de manera contigua, a todos los tensores que recibe el modelo. Los distintos valores son "extraídos" en las declaraciones, a través de los *offsets* correspondientes. Lo mismo ocurre para las `outputs` y los `weights`.

Por último, el código llama a una serie de funciones que se corresponden con las distintas **operaciones** (nodos) del modelo. Notemos que los nombres de las funciones contienen los tamaños de sus operandos, y algunos otros valores. Esto se debe a que, en vez de tener una única función para cada operador, generamos una función **para cada conjunto de parámetros usados en cada operador**. Esto nos permite aprovechar toda la información disponible estáticamente, sin tener que depender en las funciones de *inlining* del compilador, para generar código optimizado para esos parámetros específicos.

3.1. Utilización eficiente de la memoria

Durante la ejecución de un modelo, los operadores deben escribir su resultado a un tensor **destino** que funcionará entrada para un operador posterior. Estos tensores son denominados «**intermedios**» y deben estar disponibles al momento de ejecutar el operador. Para reducir el *overhead* de reservar memoria dinámica, los buffers para los tensores intermedios se reservan antes de ejecutar el modelo. Esta porción de memoria puede tomar un tamaño significativo, siendo en algunos casos órdenes de magnitud más

[3] Un vector es un tensor de rango 1 y una matriz uno de rango 2. Un tensor de rango $n+1$ se puede interpretar como una lista de tensores de rango n.

grandes que los modelos en sí. Este es el caso en redes convolucionales muy densas, es decir, con *muchas* capas.

El problema es que, si cada uno de los tensores intermedios recibe una sección exclusiva de memoria, el costo espacial empleado es muy alto. En algunos modelos resulta incluso infactible (el compilador no permite reservar tanta memoria). Por suerte, los tensores intermedios no tienen que co-existir en memoria; gracias a que la ejecución de los modelos es secuencial debido a la dependencia entre capas. Esta dependencia hace que sólo un operador esté en ejecución en cualquier momento dado, y sólo las entradas y salidas intermedias de ese operador sean necesarias. Si un modelo tiene forma de cadena, basta con alternar entre dos tensores en memoria. Lamentablemente, en general, la forma de los modelos es más compleja y nos obliga a buscar una alternativa.

A continuación mostramos una ilustración de la utilización de memoria para un modelo que asigna una porción de memoria exclusiva para cada tensor intermedio (estrategia *naïve*).

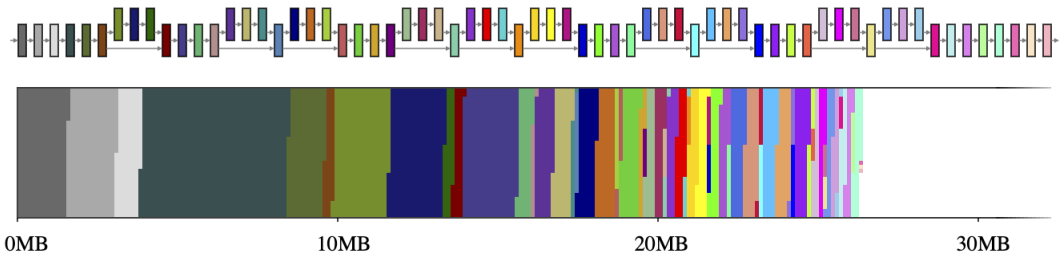

Figura 4: Asignación de memoria para MOBILENETV2 si cada tensor tiene su propio espacio en memoria. Ilustración tomada de [7].

Este problema es similar a uno que deben resolver los compiladores: el **problema de asignación de registros**. En ese caso, se debe asignar un registro a cada variable de forma tal que se utilice la mínima cantidad de registros o, al menos, se evite tener que guardar algunos valores en el *stack* (que tiene una latencia de acceso mucho mayor). Se tiene la restricción de que un registro no puede estar asignado a variables que se utilicen al mismo tiempo, es decir, que intersequen en sus *lifetimes*.

El principal inconveniente es que el problema de asignación de registros es **NP-completo** (coloreo de grafos se puede reducir a éste). Además, el problema que se nos presenta es **más general**: en lugar de asignar registros individuales, tenemos que asignar regiones de memoria con distintos tamaños, y minimizar el espacio total utilizado siguiendo las restricciones temporales.

Sin embargo, existen múltiples algoritmos aproximados que dan soluciones suficientemente buenas (y frecuentemente óptimas). Estas soluciones se describen en un blogpost [7] del equipo de *TensorFlow Lite*[4], que está basado en el paper [8]. Las heurísticas se dividen en 2 categorías generales:

- **Shared Memory Buffer Objects**: Estas técnicas se utilizan para entornos como WebGL (uno de los *targets* de TF Lite), donde cada tensor estará almacenado como textura, cuyo tamaño no puede cambiar después de su creación. Como estas restricciones no aplican a nuestro caso, elegimos la otra alternativa.
- **Memory Offset Calculation**: En este caso, se asume que la inferencia correrá en una CPU, y que se puede reservar una *pool* de memoria grande. Los tensores son *offsets* dentro de esta pool, y se deben encontrar la asignación de offsets que minimice el tamaño total.

[4]TensorFlow Lite es un conjunto de herramientas para desarrollar redes neuronales en dispositivos embebidos, móviles o de IoT. Estas plataformas suelen tener poca cantidad de memoria por lo que es crítico hacer un uso eficiente de la misma.

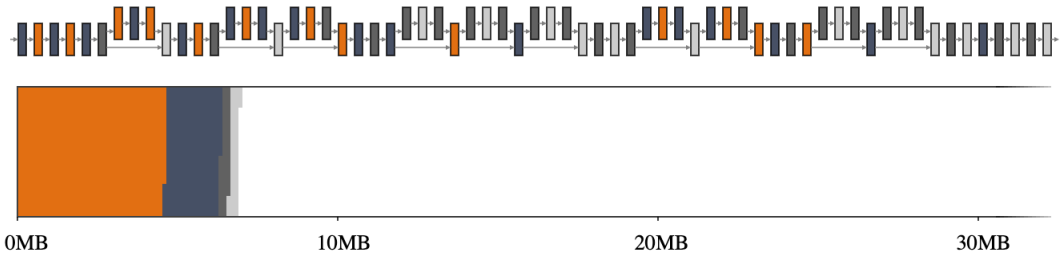

Figura 5: Asignación de memoria para MOBILENETV2 si los tensores comparten memoria. Ilustración tomada de [7]

.

Después de implementar un algoritmo *greedy* para asignar los vectores a offsets de memoria, logramos reducir el *footprint* lo suficiente como para correr modelos de mayor tamaño en el Zoo.

3.2. Estructura de los operadores

Recordemos que los **operadores** de los modelos de ML son los componentes que definen el cómputo que se realiza. La versión 1.14.0 del formato ONNX define 187 operadores, algunos de los cuales cuentan con múltiples variaciones que cambian significativamente su comportamiento. Parte de nuestro trabajo consistió en pre-seleccionar un conjunto abarcativo para `onnx2code`.

Como mencionamos anteriormente, para cada operador y conjunto de parámetros correspondiente, generamos una función que recibe los tensores de entrada y ejecuta el cómputo apropiado, escribiendo el resultado en otros tensores. El siguiente es otro extracto del código generado para el modelo de la sección anterior:

```
void Relu_262144(const float* __restrict__ A, float* __restrict__ OUT) {
    for(int i = 0; i < 262144; i++) {
        OUT[i] = A[i] > 0 ? A[i] : 0;
    }
}
```

Figura 6: Código generado para aplicar la función de activación ReLU a una matriz de tamaño 512×512.

Un detalle interesante en la *signature* de la función es el atributo `__restrict__`. Durante la implementación de algunos de los operadores observamos que, en varios loops que parecían simples de vectorizar, el compilador generaba código que no hacía uso de las instrucciones SIMD, o generaba ambos loops y saltaba a uno dependiendo de una condición. Después de investigar, aprendimos que esto se debía al posible *aliasing* entre los parámetros y que, si sabíamos que los punteros apuntaban a bloques de memoria distintos, podíamos solucionarlo utilizando `restrict`.

3.2.1. Atributo `restrict`

`restrict` es una *keyword* del estándar C99, implementada de forma no-estándar en varios compiladores de C++. Ésta provee un mecanismo para indicarle al compilador que, durante el *lifetime* de un puntero, éste es el único que tiene acceso a su sección de memoria (es decir, no hay *aliasing*).

Para ver los efectos de este atributo, podemos analizar una implementación simple de la suma entre 2 vectores de 10000 elementos:

```
void add(const float* A, const float* B, float* OUT) {
    for (int i = 0; i < 10000; ++i) {
        OUT[i] = A[i] + B[i];
```

```
    }
}
```

El loop de la función es un buen candidato para el uso de operaciones SIMD: si la arquitectura cuenta con la extensión **SSE**, se puede *unrollear*[5] el loop de forma tal que se ejecuten 4 operaciones al mismo tiempo al hacer uso de las instrucciones `movups` y `addps`.

Sin embargo, esa implementación es incorrecta para algunos parámetros. Si se llama a la función con `OUT = A + 1` (apunta al segundo elemento de `A`), cada iteración cambia el próximo elemento de A, así que ejecutarlas de a bloques de 4 no tiene el mismo efecto. Como el compilador debe producir código correcto para todos los parámetros, no puede utilizar el loop vectorizado. En cambio, en `gcc` versión 12.2 con flags `-O3 -msse`, genera código que primero verifica que haya suficiente distancia entre las direcciones de `A/B` y `OUT` y, de ser así, ejecuta la alternativa rápida.

```
add(float const*, float const*, float*):
    lea     rcx, [rdi+4]
    mov     rax, rdx
    sub     rax, rcx
    cmp     rax, 8
    jbe     .L5
    lea     rcx, [rsi+4]
    mov     rax, rdx
    sub     rax, rcx
    cmp     rax, 8
    jbe     .L5
    xor     eax, eax
.L3:
    movups  xmm0, XMMWORD PTR [rdi+rax]
    movups  xmm1, XMMWORD PTR [rsi+rax]
    addps   xmm0, xmm1
    movups  XMMWORD PTR [rdx+rax], xmm0
    add     rax, 16
    cmp     rax, 4096
    jne     .L3
    ret
.L5:
    xor     eax, eax
.L2:
    movss   xmm0, DWORD PTR [rdi+rax]
    addss   xmm0, DWORD PTR [rsi+rax]
    movss   DWORD PTR [rdx+rax], xmm0
    add     rax, 4
    cmp     rax, 4096
    jne     .L2
    ret
```

Como, en nuestro caso, los punteros de entrada siempre van a apuntar a secciones de memoria disjuntas, podemos usar `restrict` (o `__restrict__` en `g++`) para indicárselo al compilador. Luego, si recompilamos el código con las mismas opciones, obtenemos el resultado deseado:

```
add(float const* __restrict__, float const* __restrict__, float* __restrict__):
    xor     eax, eax
.L2:
    movups  xmm0, XMMWORD PTR [rsi+rax]
    movups  xmm1, XMMWORD PTR [rdi+rax]
    addps   xmm0, xmm1
    movups  XMMWORD PTR [rdx+rax], xmm0
    add     rax, 16
    cmp     rax, 4096
    jne     .L2
    ret
```

[5] El *unrolling* es una técnica para acelerar la ejecución de un loop al reemplazar el código del cuerpo, que ejecuta cada iteración individualmente, por uno que lo haga de "a bloques". Esto hace menos frecuente el chequeo de condición, que puede causar una *misprediction* (y perder ciclos de clock por el *flushing* del pipeline). La técnica es más eficiente todavía en arquitecturas con operaciones vectorizadas, ya que estas podrían correr múltiples iteraciones con una única instrucción.

3.2.2. Modularidad

La librería `onnx2code` tiene una **estructura modular**: separamos la implementación de cada operador (o, en algunos casos, conjunto de operadores similares) en una jerarquía de objetos con una interfaz bien definida: reciben la información del nodo (como el tipo de operador y sus parámetros) junto con sus entradas y salidas, y son capaces de devolver tanto la definición de la función correspondiente, como la invocación que debe contener `inference`. Es común que el mismo operador se utilice varias veces en el mismo modelo, por lo cual de-duplicamos las definiciones de funciones equivalentes.

Este enfoque modular para afrontar el proyecto nos brindó muchas ventajas a la hora del desarrollo, ya que pudimos usar un *branch*[6] para cada una de las operaciones, permitiendo implementarlas sin afectar el desarrollo de las demás (una práctica muy utilizada y muy útil a la hora de desarrollar este tipo de librerías).

Otra característica de nuestra estructura de operadores son las **variantes**: cada uno cuenta con múltiples implementaciones y, a la hora de generar el modelo, se selecciona una mediante un sistema de prioridades. Esto nos sirvió a la hora de evaluar el rendimiento de las distintas alternativas.

3.3. GEMM

Una de las operaciones más recurrentes dentro de la inferencia en las redes neuronales es la multiplicación de matrices (o GEMM por "General Matrix Multiplication"). Este tipo de operaciones son necesarias tanto en la multiplicación en sí misma, como también en la operación llamada convolución. Como se muestra en la figura 2, la convolución es de las operaciones más utilizadas en los modelos: ésta, junto con la multiplicación de matrices, representan alrededor del 90 % del tiempo de inferencia [10]. Por esta razón, dedicamos gran parte de nuestro esfuerzo a optimizar nuestra implementación del operador.

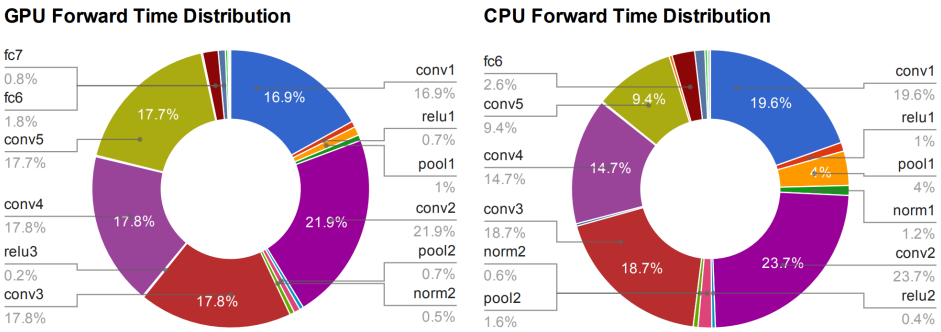

Figura 7: Análisis empírico sobre la proporción del tiempo de inferencia que toma cada nodo. Resultado tomado de [10].

En nuestra búsqueda por la eficiencia dentro de este tipo de operaciones encontramos un número importante de investigaciones recientes realizadas al respecto, las cuales se enfocan en optimizar al extremo las condiciones en las que se realizan las instrucciones y los accesos a memoria, modificando el orden y la metodología de las operaciones a realizar, con el objetivo de mejorar la utilización del caché, los distintos threads y los registros del procesador, entre otros recursos. Estos acercamientos logran una eficiencia importante en casos donde no es posible o aconsejable el uso de la GPU.

Para explicar el funcionamiento de nuestra implementación de GEMM, primero debemos introducir los distintos conceptos profundizados en estas investigaciones. Uno de los conceptos con mayor importancia será el *loop tiling*.

[6] El término *branch* se refiere a un mecanismo de los sistemas de control de versiones (como *git*), y no a la estructura de control de flujo.

3.3.1. Loop tiling

El *loop tiling* es una técnica de optimización aplicable a conjuntos de loops anidados, que consiste en subdividir el conjunto de iteraciones de uno (o más) de ellos en "bloques" o "*tiles*": ciclos internos con menos iteraciones. Esto permite correr los bloques en paralelo y mejora la localidad de los accesos a memoria (permitiendo un mejor aprovechamiento de la jerarquía de caché). Como `onnx2code` está diseñado para correr en 1 solo núcleo, solo nos afecta esta última ventaja.

Para entender bien este proceso, podemos enfocarnos en una operación más simple que la multiplicación de matrices: el producto externo. Dados dos vectores $a \in \mathbb{R}^n$ y $b \in \mathbb{R}^m$, su producto externo $a \otimes b = ab^t$ es una matriz $C \in \mathbb{R}^{n \times m}$ tal que $C_{ij} = a_i b_j$. Como veremos más adelante, esta operación formará parte de nuestra implementación de GEMM. Podemos computarla con el siguiente programa:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        C[i][j] = a[i] * b[j];
    }
}
```

Notemos que el arreglo b debe ser leído de la memoria n veces (una por cada iteración del ciclo externo). Esto no es un problema si, por ejemplo, b entra en el caché L1: solo la primera lectura tendrá problemas de latencia, y después las demás serán *hits*. Sin embargo, si es demasiado grande, es probable que haya *thrashing*, donde cada lectura reemplaza datos previamente guardados en el caché que deberán ser cargados de nuevo.

Podemos mitigar este problema *tilieando* el loop interno, de la siguiente manera:

```
for (int jj = 0; jj < m; jj += mb) {
    for (int i = 0; i < n; i++) {
        for (int j = jj; j < min(jj + jb, m); j++) {
           C[i][j] = a[i] * b[j];
        }
    }
}
```

Ahora, en cada iteración de `i`, se leerán a lo sumo `mb` valores de `b`, y se puede tomar un `mb` suficientemente bajo como para que entren en el caché. No obstante, esta transformación trajo un nuevo problema: debido al nuevo ciclo, el arreglo `a` será leído $\frac{m}{mb}$ veces. Podríamos intentar elegir un `mb` que mantenga un buen balance entre la cantidad de lecturas de a y el tamaño que se lee de b, o volver a usar *loop tiling*:

```
for (int ii = 0; ii < n; ii += nb) {
    for (int jj = 0; jj < m; jj += mb) {
        for (int i = ii; i < min(n, ii + ib); i++) {
            for (int j = jj; j < min(m, jj + jb); j++) {
                C[i][j] = a[i] * b[j];
            }
        }
    }
}
```

Ahora, los elementos de C se calculan en bloques de `ib` $\times$ `jb` elementos, y podemos elegir un tamaño que aproveche al máximo el caché.

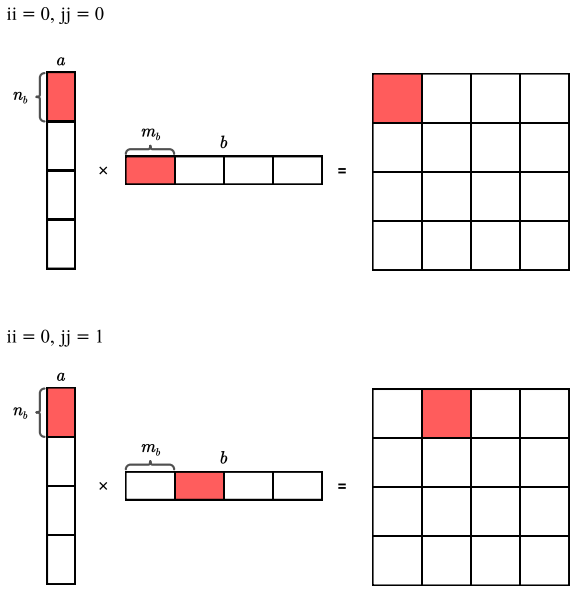

Figura 8: Ilustración de las 2 primeras iteraciones del algoritmo de producto externo con loop-tiling.

3.3.2. Estructura general

Nuestra rutina para GEMM está basada en las observaciones de [2] y [9], que describen el acercamiento que se utiliza en las librerías GOTOBLAS (ahora mantenida como OPENBLAS) y BLIS, una de las implementaciones más eficientes de BLAS[7]. Al igual que nuestro ejemplo del producto externo, consiste en particionar las operaciones de manera tal que se aproveche al máximo toda la jerarquía de caché.

Concretamente, se trata de una serie de ciclos anidados que realizan multiplicaciones entre matrices progresivamente más pequeñas, hasta llegar al punto en el que todos sus elementos puedan ser almacenados en los registros. Vamos a explicar el funcionamiento de cada uno, llamando $A \in \mathbb{R}^{m \times k}$ y $B \in \mathbb{R}^{k \times n}$ a las entradas y $C = A \times B \in \mathbb{R}^{m \times n}$ al resultado.

1. **Tiling de B**

   El loop más externo divide (implícitamente) a B en submatrices de la misma altura pero a lo sumo n_c elementos de ancho. Esto se debe a que en el siguiente paso se tomarán *paneles* de B que deben caber en el caché L3. Sin embargo, como éste suele ser de gran capacidad, los valores que asignamos a n_c son más grandes que el ancho de la mayoría de las matrices: en tal caso, este "loop" es de una sola iteración.

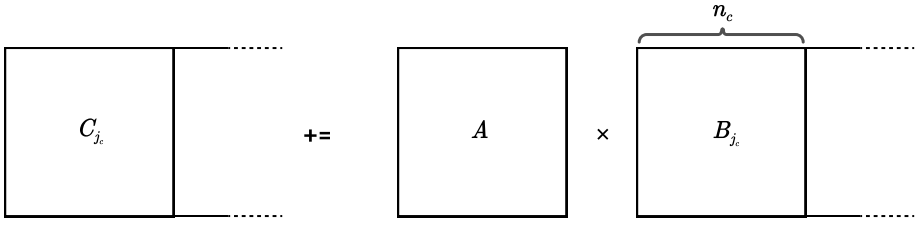

Figura 9: Cada iteración j_c de este ciclo computa la multiplicación entre A y la submatriz $B_{j_c} \in \mathbb{R}^{k \times n_c}$.

2. **Iteración de paneles de B_{j_c} y empaquetamiento a $\tilde{B}$**

   Este loop divide a B_{j_c} en *paneles* de k_c filas, y a la matriz A en unos de k_c columnas. Un panel es una

[7] BLAS (Basic Linear Algebra Subprograms) es una especificación para una API de operaciones de álgebra lineal.

submatriz con una dimensión “grande” y una “pequeña”. Las dimensiones del panel $B_{p_c} \in \mathbb{R}^{k_c \times n_c}$ deben permitir que éste entre en el caché L3.

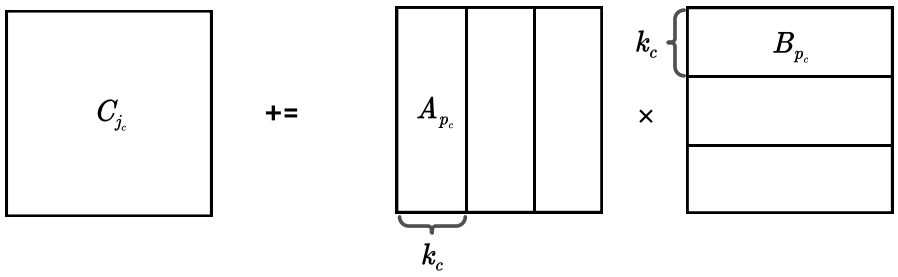

Figura 10: Cada iteración p_c de este ciclo representa una multiplicación *panel a panel*, denominada GEPP en la literatura, entre $A_{p_c} \in \mathbb{R}^{m \times k_c}$ y $B_{p_c} \in \mathbb{R}^{k_c \times n_c}$.

Además, *empaquetamos* los elementos de B_{p_c} en un buffer local $\tilde{B}$ de forma tal que todos los accesos en los pasos subsiguientes sean secuenciales. El *overhead* de la copia de memoria es compensado por la mejora en el patrón de acceso.

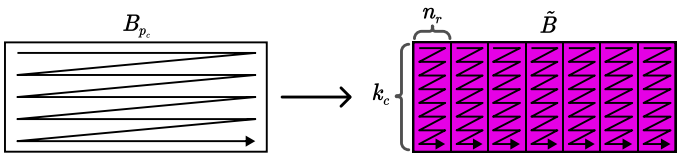

Figura 11: Empaquetamiento $B_{p_c} \to \tilde{B}$.

3. **Iteración de bloques de A_{p_c} y empaquetamiento en $\tilde{A}$**

   Este loop es análogo al anterior, pero itera *bloques* (submatrices con ambas dimensiones pequeñas) de A_{p_c}, cada uno con m_c filas.

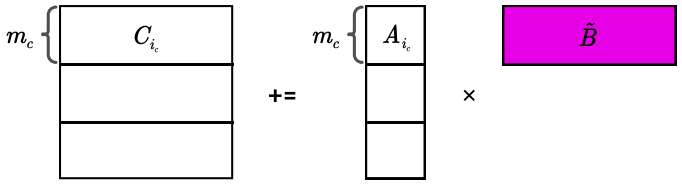

Figura 12: Cada iteración i_c de este ciclo es una multiplicación *bloque a panel*, también llamada GEBP, entre $A_{i_c} \in \mathbb{R}^{m_c \times k_c}$ y $\tilde{B} \in \mathbb{R}^{k_c \times n_c}$.

Nuevamente, empaquetamos los bloques en un buffer, denotado $\tilde{A}$, que se almacenará en el caché L2. Como en el caso anterior, ordenamos los elementos para optimizar la localidad de los accesos.

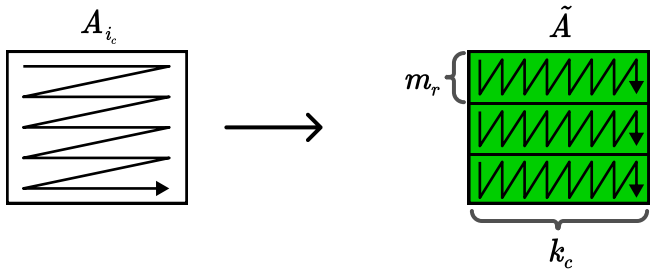

Figura 13: Empaquetamiento $A_{i_c} \to \tilde{A}$

4. **Iteración de *slivers* de $\tilde{B}$**

   Como explicamos, los loops anteriores nos dejan con secciones de las matrices de entrada cargadas en los 2 niveles de caché superiores. En este ciclo se divide al panel $\tilde{B}$ en *slivers* (también conocidos como *micro-paneles*) de n_r columnas. Debido al empaquetamiento previo, el sliver tiene el *layout* de una matriz *row-mayor*[8] de $k_c \times n_r$.

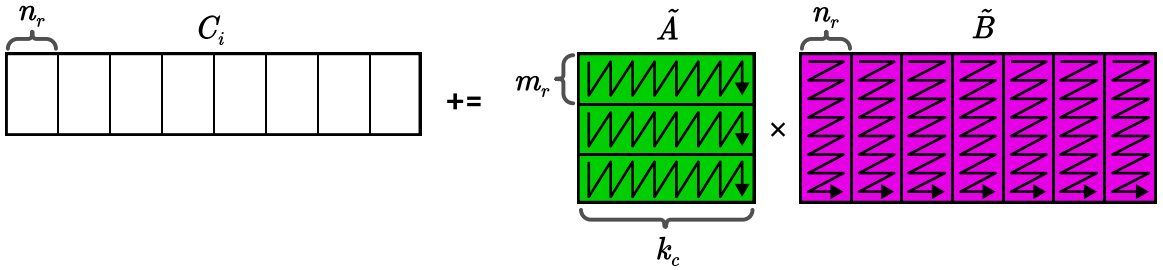

Figura 14: Cada iteración j_r de este ciclo multiplica el bloque $\tilde{A} \in \mathbb{R}^{m_c \times k_c}$ por un sliver $\tilde{B}_{j_r} \in \mathbb{R}^{k_c \times n_r}$.

5. **Iteración de *slivers* de $\tilde{A}$**

   Este ciclo es análogo al anterior: divide al bloque $\tilde{A}$ en slivers de m_r columnas, que están guardados en la memoria de forma *column-mayor*. Ambos slivers deberían entrar el caché L1, ya que cada uno de sus elementos será leído k_c veces en el *microkernel*.

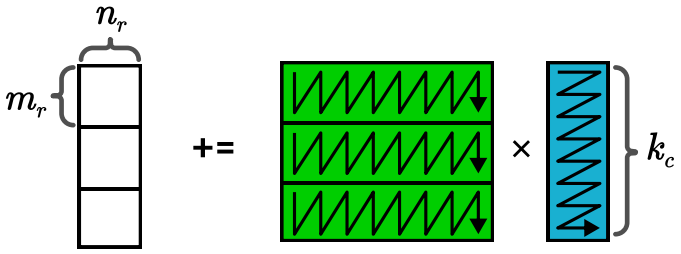

Figura 15: Cada iteración i_r de este ciclo realiza una multiplicación entre los slivers $\tilde{A}_{i_r} \in \mathbb{R}^{m_r \times k_c}$ y $\tilde{B}_{j_r} \in \mathbb{R}^{k_c \times n_r}$.

[8] Una matriz está en *row-mayor* cuando está guardada como una serie de filas contiguas en memoria.

3.3.3. Microkernel

El *microkernel* de la implementación de GEMM es la multiplicación entre los slivers $\tilde{A}_{i_r}$ y $\tilde{B}_{j_r}$. Se trata de un producto de matrices que, después de múltiples niveles de *tiling* y empaquetamientos, tienen un resultado pequeño (es de $m_r \times n_r$, y esos parámetros toman valores menores a 32).

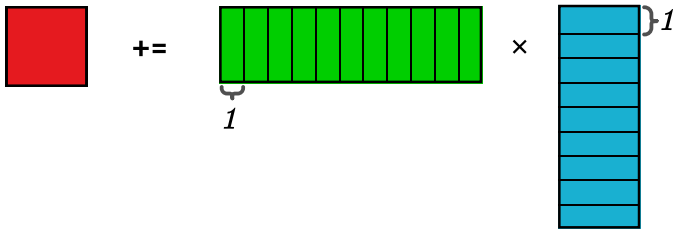

Figura 16: Ilustración de la operación en el microkernel

El algoritmo tradicional para multiplicar matrices consiste en iterar por las filas de una y las columnas de la otra, y computar el producto interno entre ellas para obtener, en cada iteración, un elemento del resultado. En la terminología de compiladores, se trata de las variantes IJK y JIK, llamadas así por los nombres que suelen tomar los índices de cada ciclo.

Nuestra multiplicación, por otro lado, se realiza mediante un ciclo KJI: esto implica computar el producto externo entre cada fila de $\tilde{A}_{i_r}$ y columna de $\tilde{B}_{j_r}$ y sumar todos los resultados.

$$
\begin{aligned}
AB \mathrel{+}= & \tilde{A}_{i_r} \cdot \tilde{B}_{j_r} \\
\mathrel{+}= & \left(\begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_{k_c} \end{array} \right) \cdot \left(\begin{array}{c} b_1 \\ \hline b_2 \\ \hline \vdots \\ \hline b_{k_c} \end{array} \right) \\
\mathrel{+}= & \sum_{i=0}^{k_c} a_i b_i^T
\end{aligned}
$$

Al igual que en el ejemplo anterior, utilizamos loop tiling para computar los productos externos de forma eficiente. Sin embargo, en este caso, las dimensiones son tan chicas que cada bloque interno, denominado *unit update*, puede ser computado en pocas instrucciones SIMD. Un producto externo individual tiene la siguiente forma:

```
for (int j = 0; j < nr; j += nu) {
        for (int i = 0; i < mr; i += mv) {
            unit_update(
                A_ir + i,
                B_jr + j,
                AB + i * nr + j
            );
        }
    }
}
```

Esta última función, `unit_update`, se optimiza profundamente en [9], donde desarrollan un modelo para los procesadores basado en teoría de encolamiento, y componen instrucciones vectoriales de *shuffle*, *FMA*[9], *load* y *broadcast* de forma tal que se maximice el *throughput* en distintas arquitecturas. No obstante,

[9] FMA, o *fused multiply-add*, es un conjunto de instrucciones que se agregó a la arquitectura x86. Son de la forma $c \leftarrow a \cdot b + c$, y resultan ser muy eficientes para el cálculo de sumas de productos externos.

las técnicas utilizadas estuvieron fuera de nuestro alcance: no logramos implementar código más eficiente que el que genera `gcc`.

A pesar de no replicar los métodos del trabajo, sí utilizamos sus observaciones para obtener criterios sobre los parámetros m_r, n_r, m_v y n_u (los que afectan al producto externo dentro del microkernel). La combinación ideal debería utilizar la mayor cantidad de registros SIMD posible, sin *spillear* valores al stack. Estos spills no suelen ser problemáticos en otros programas, gracias a la velocidad del caché L1, pero en nuestro caso generan dependencias de datos que inhiben el re-ordenamiento de la ejecución por parte de la CPU, y producen indirectamente una mayor cantidad de *misses* del *TLB* [9]. En la figura 17 podemos ver un microkernel con parámetros que evitan el spilling y utilizan 13 de los 16 registros `ymm` que posee la arquitectura x86-64 con la extensión AVX2.

```
void microkernel<4, 16, 4, 8>(int kc(rdi), float* a(rsi), float* b(rdx), float* ab(rcx)):
    [...]
    vmovups ymm8, YMMWORD PTR [rcx]             ; ymm8 <- ab[0:7]
    vmovups ymm4, YMMWORD PTR [rcx+32]          ; ymm4 <- ab[8:15]
    [...]
    vmovups ymm1, YMMWORD PTR [rcx+224]         ; ymm1 <- ab[56:63]
.L1:
    vmovups ymm0, YMMWORD PTR [rdx]             ; ymm0 <- b[0:7]
    vbroadcastss    ymm12, DWORD PTR [rax]      ; ymm12 <- a[0] (8 veces)
    vbroadcastss    ymm11, DWORD PTR [rax+4]    ; ymm11 <- a[1] (8 veces)
    [...]
    vfmadd231ps     ymm8, ymm0, ymm12           ; ymm8 <- ymm8 + b[0:7] * a[0]
    vfmadd231ps     ymm8, ymm0, ymm11           ; ymm7 <- ymm7 + b[0:7] * a[1]
    [...]
    vmovups ymm0, YMMWORD PTR [rdx+32]          ; ymm0 <- b[8:16]
    vfmadd231ps     ymm4, ymm0, ymm12           ; ymm4 <- ymm4 + b[8:16] * a[0]
    vfmadd231ps     ymm3, ymm0, ymm11           ; ymm3 <- ymm4 + b[8:16] * a[1]
    [...]
    add rsi, 16                                 ; a <- a + mr
    add rdx, 64                                 ; b <- b + nr
    cmp [...]
    jne .L1
    vmovups YMMWORD PTR [rsi], ymm8             ; ab[0:7] <- ymm8
    vmovups YMMWORD PTR [rsi+32], ymm4          ; ab[8:15] <- ymm4
    [...]
    ret
```

Figura 17: Extracto de código generado para la función `microkernel<mr=4, nr=16, mv=4, nu=8>(kc, A, B, AB)`, utilizando instrucciones AVX2 (`unit_update` fue *inlineada*). Se puede ver que, para estos parámetros, se obtiene una función que aprovecha la mayoría de los registros `ymm`, y no guarda ningún valor en la pila.

3.3.4. Ilustración completa de GEMM

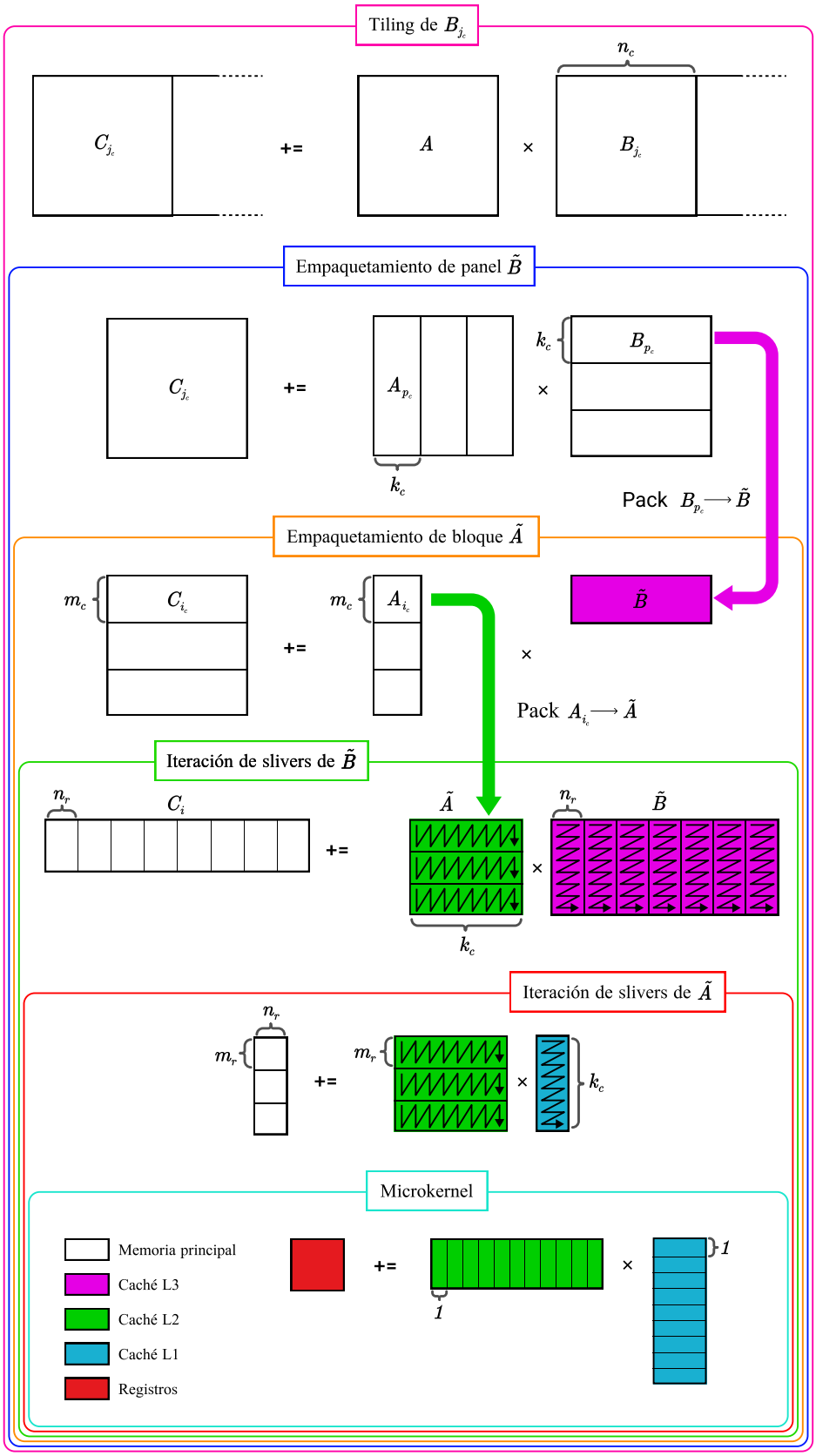

Figura 18: Ilustración de GEMM. Diseño inspirado en [9].

3.4. Otros Operadores

Mientras que nuestro enfoque principal fue GEMM, la mayoría de los modelos están compuestos por un conjunto diverso de operadores. A continuación describimos (no exhaustivamente) los que implementamos en `onnx2code`.

3.4.1. Elementwise

Estos son los operadores que solamente aplican una función a cada elemento del tensor de entrada. La categoría incluye a:

- RELU, TANH y SIGMOID: Tres funciones de activación comúnmente utilizadas en las redes neuronales para introducir un grado de no-linearidad.
- CLIP: Dados un máximo y un mínimo, *clipea* todos los valores de un tensor, asignando las cotas a aquellos que se encuentran fuera del intervalo.
- SUM: Suma una cantidad arbitraria de tensores elemento a elemento (a diferencia de ADD, que es un operador binario).

Todos estos operadores se implementan fácilmente mediante un único ciclo que aplica la función deseada a lo largo del tensor.

3.4.2. Broadcastable

Esta categoría incluye a los operadores binarios clásicos: la suma, la multiplicación, la resta y la división. Se llama así porque, cuando uno de los tensores de entrada es de distintas dimensiones/rango que el otro, el menor se *broadcastea* a lo largo del mayor. Esto implica "repetir" el segundo, con el objetivo de que los tamaños sean compatibles. Un ejemplo sería el siguiente:

$$\begin{bmatrix}1 & 3\\2 & 4\end{bmatrix} + \begin{bmatrix}1\\2\end{bmatrix} \underset{broadcast}{\longrightarrow} \begin{bmatrix}1 & 3\\2 & 4\end{bmatrix} + \begin{bmatrix}1 & 1\\2 & 2\end{bmatrix} = \begin{bmatrix}2 & 4\\4 & 6\end{bmatrix}$$

Las reglas generales para el broadcasting son complejas, y pueden ser exploradas en [5]. Afortunadamente, NUMPY posee la función `nditer`, que recibe dos tensores, y devuelve los accesos que se deben realizar a cada variable. Utilizamos esos accesos para generar un conjunto de loops anidados que realiza la operación correcta.

3.4.3. Reshapes

Hay un conjunto de operadores (como RESHAPE, SQUEEZE y UNSQUEEZE) que cambian las dimensiones de los tensores sin afectar los datos subyacentes. Como esta información dimensional es parte de la metadata del formato ONNX, nosotros no tuvimos que implementar dichas operaciones: si el modelo indica que un tensor U es la salida de un RESHAPE de T, simplemente asignamos la misma variable a ambos.

3.4.4. Convolución

Como adelantamos en la introducción, la convolución se puede convertir en una multiplicación de matrices mediante una técnica llamada **im2col**, *image to column*. El primer paso es transformar la imagen 3D (alto, ancho y canales) en un array 2D que podamos tratar como una matriz.

La idea es tomar cada patch que el kernel va a recorrer de la imagen 3D durante la convolución (teniendo en cuenta *strides*, *padding* y *dilations*) y copiarlo en una fila de una nueva matriz. Por este proceso es que la técnica se llama *im2col*, que se ilustra a continuación.

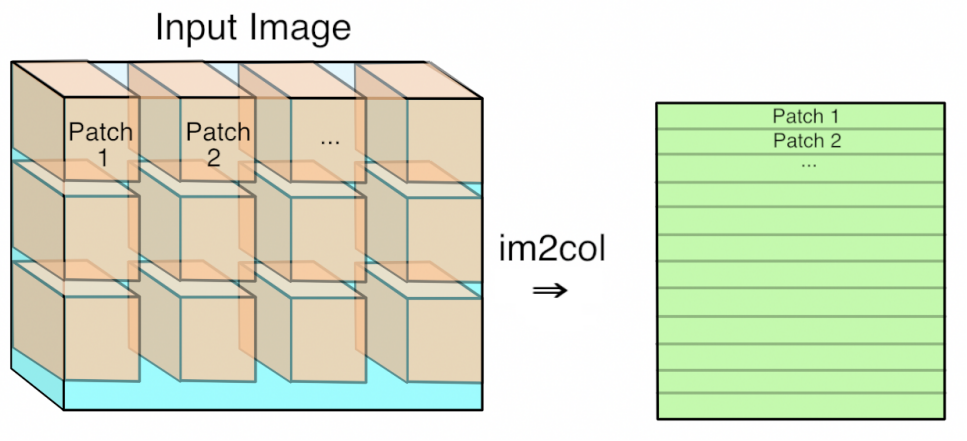

Figura 19: Transformación *im2col*. Ilustración tomada de [10].

Esta transformación, a primera vista, parece que genera una explosión en memoria. Si el stride es menor al ancho del kernel, estaremos copiando los mismos píxeles en múltiples filas (duplicados). La desventaja parece ser irremediable, pero como veremos durante la experimentación, el desperdicio de memoria es superado por las ventajas, principalmente por los accesos regulares a memoria.

Para continuar, el mismo procedimiento se puede realizar para el tensor de kernels, transformando los cubos 3D (kernels) en una matriz 2D mediante *im2col*. Afortunadamente no hay que realizar esta transformación en tiempo de ejecución ya que el orden en que se almacena este tensor por defecto (formato de almacenamiento de ONNX) coincide con el producido por *im2col*.

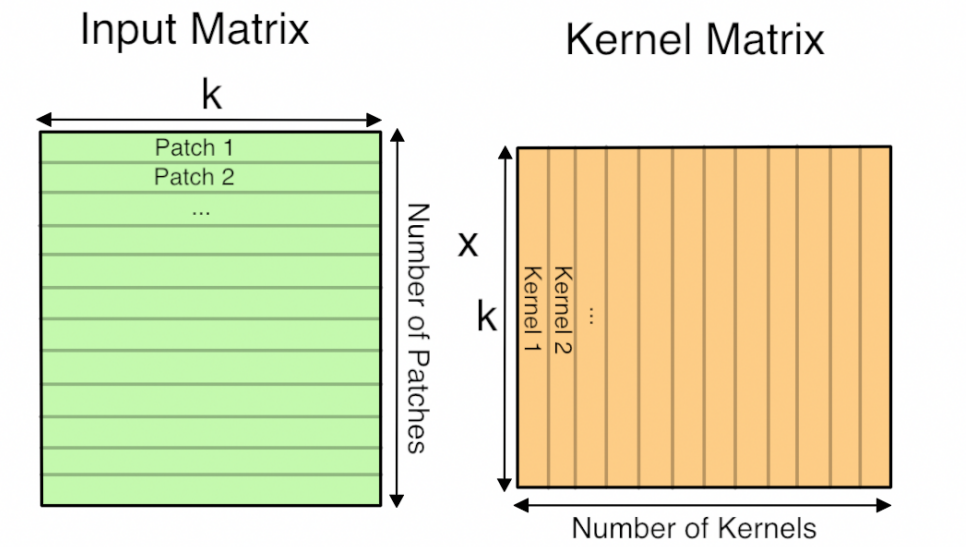

Figura 20: Multiplicación de matrices que produce el resultado de una convolución. Ilustración tomada de [10].

La multiplicación de estas dos matrices, entonces, va a producir el resultado de la convolución como si se hubiera hecho de la forma tradicional. Cada píxel (de cada kernel) de la convolución final es el resultado del producto interno entre el patch y el kernel correspondiente. Finalmente, el orden que toman los píxeles en memoria es el esperado: $P \times K$, donde P es el número de patches y K el número de kernels.

Durante décadas la investigación se centró en optimizar código para multiplicación de matrices grandes. Esta función, que toma la mayor parte del tiempo de cómputo (como vimos anteriormente),

crea un camino visible para optimizar no sólo esta operación tan común, pero también la convolución como acabamos de ver.

3.5. Testing

Desarrollar la librería de manera modular nos permite, entre otras cosas, hacer *unit testing* de forma mucho más simple. Para realizar dicha tarea utilizamos **pytest** y generamos una matriz de test para cada módulo, donde se producen distintas entradas y se ponen a prueba utilizando como referencia para las salidas correspondientes, la librería ONNX Runtime.

Por otro lado, también fue posible testear por separado los modelos que debería soportar la librería realizando tests de integración en los que se utiliza como entrada el modelo correspondiente, y la salida se compara nuevamente con la salida de ONNX Runtime. En el estado actual de la librería, se puede generar código y se verifican más de 10 modelos conocidos, entre ellos VGG 16, VGG 19, INCEPTIONV2 y SQUEEZENET1.1.

En total hay más de 800 tests entre operadores y modelos, lo que hace que correr todo el suite llegue a demorar una hora. Para agilizar el desarrollo utilizamos integración continua para que los tests se ejecuten en cada *push*, para detectar regresiones sin correr localmente todo el suite.

4. Experimentación

En esta sección detallaramos los experimentos que realizamos sobre la librería `onnx2code`. Para comenzar, evaluamos el performance de distintos parámetros de GEMM con el fin de encontrar la combinación óptima, comprobando las hipótesis que teníamos sobre su funcionamiento.

Una vez que obtuvimos los párametros óptimos para cada CPU, compararemos nuestra implementación de GEMM con distintos runtimes de ML. Posteriormente, contrastaremos el rendimiento de hacer inferencias completas en distintos modelos existentes.

Elegimos realizar los experimentos en distintos procesadores, principalmente de generaciones distintas, siendo nuestro objetivo entender cómo impactan los distintos parámetros y técnicas en cada una. A continuación se detallan las características de las CPUs elegidas:

Generación	CPU	Flags	Caché		
			L1	L2	L3
Skylake (6th)	Intel Core i7 6700K	AVX2	32KBytes	256KBytes	8MB
Comet Lake (10th)	Intel Core i7 10700K	AVX2	32KBytes	256KBytes	16MB
Ice Lake (10th)	Intel Core i7 1065G7	AVX512F	48KBytes	512KBytes	8MB
Tiger Lake (11th)	Intel Core i7 1165G7	AVX512F	48KBytes	1.25MB	12MB

Cuadro 1: Especificaciones de los procesadores utilizados en la experimentación. Las memorias caché L1 y L2 se muestran por core.

4.1. Parámetros de GEMM

En la sección donde describimos la estructura general de GEMM definimos diversos parámetros que afectan directamente cómo el compilador generará el código, y por ende su rendimiento. El objetivo es buscar los parámetros que mejor se ajustan a cada procesador particular, para posteriormente intentar corroborar que el comportamiento es el que esperamos.

Realizamos una búsqueda en grilla con parámetros que consideramos adecuados, eliminando configuraciones incompatibles (dimensiones no conformantes). Las listamos a continuación:

Parámetro	Valores
n_c	N (512)
k_c	64, 128, 256, 512
m_c	64, 128, 256, 512
m_r	2, 4, 8, 16, 32
n_r	2, 4, 8, 16, 32
m_v	2, 4, 8, 16
n_u	2, 4, 8, 16

Cuadro 2: Matriz de parámetros a evaluar. De las 6400 combinaciones posibles, sólo 3136 resultan compatibles. n_c tiene un solo valor porque en matrices de 512×512 no es necesario tilear B.

Para cada combinación de parámetros, se mide el tiempo que tarda en ejecutar la multiplicación entre dos matrices $\mathbb{R}^{512\times512}$. Se realizan 30 ejecuciones de calentamiento y posteriormente se promedian 300 ejecuciones, para reducir el ruido en las mediciones. Debido a la gran cantidad de combinaciones de parámetros y repeticiones, la búsqueda demora en promedio 8 horas.

4.2. Análisis de resultados

Una vez obtenidos los resultados el *benchmark*, realizamos un análisis exploratorio para determinar qué propiedades de los parámetros llevan a una buena implementación de GEMM.

4.2.1. Tamaño del microkernel

El patrón más significativo que encontramos es en los parámetros que controlan el tamaño del *microkernel* (m_r, n_r, m_v y n_u). Recordemos que este representa una multiplicación entre *slivers* de $\tilde{A}$ y $\tilde{B}$ de tamaños $m_r \times k_c$ y $k_c \times n_c$ (respectivamente), computada como suma de productos externos.

Durante la introducción de los microkernels conjeturamos que, para obtener un código eficiente, debemos utilizar un tamaño de producto externo (dado por $m_r \times n_r$) que maximice la cantidad de registros utilizados, pero evite el *spilling* al stack. Para confirmar esto, podemos analizar las 500 mejores combinaciones de parámetros en cada arquitectura.

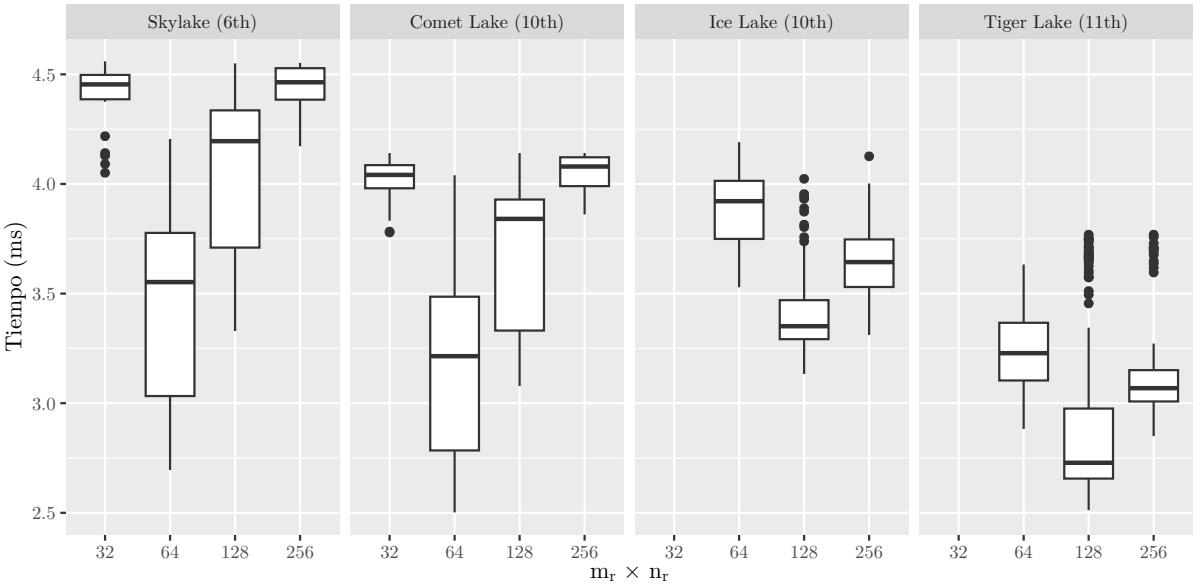

Figura 21: Distribución del tiempo que toma una multiplicación con respecto al tamaño del microkernel ($m_r \times n_r$), para cada arquitectura.

En la figura 21 podemos observar que los tamaños de microkernel más efectivos son: 64 para las arquitecturas con AVX2 (que cuenta con 16 registros `ymm` de 256 bits c/u), y 128 para aquellas con AVX512 (que posee 32 registros `zmm` de 512 bits c/u). Estos son los tamaños máximos que se pueden utilizar antes de que se requiera guardar valores en el stack.

4.2.2. Utilización del caché

Otro factor que afecta al rendimiento de nuestra implementación es el uso del caché. El algoritmo está diseñado para almacenar distintas estructuras en cada nivel de la jerarquía de caché y, si alguna excede el tamaño que posee la jerarquía, esperaríamos ver una degradación en el performance como producto del *thrashing*.

Para verificar esta hipótesis tomamos, del conjunto de datos anterior, sólo aquellos que tienen el microkernel de tamaño óptimo (64 o 128, dependiendo de la arquitectura). Primero, analizamos el comportamiento con respecto al caché L1: en este nivel se debe almacenar un sliver de $\tilde{B}$ en cada multiplicación del microkernel. Estos slivers tienen un tamaño de $k_c \times n_c \times 4$ bytes.

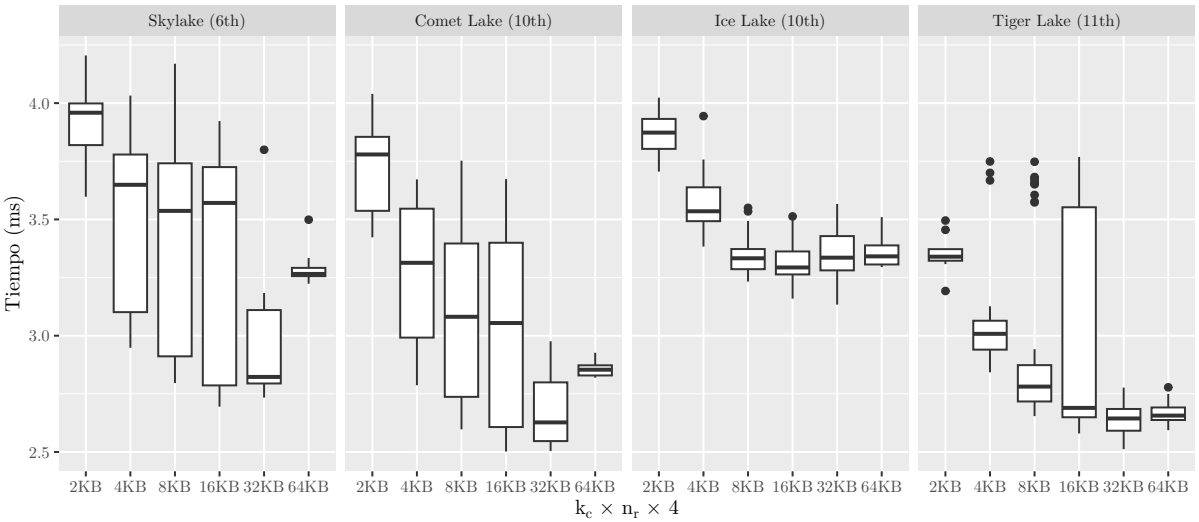

Figura 22: Distribución del tiempo que toma cada multiplicación con respecto al tamaño de cada sliver de $\tilde{B}$.

Teniendo en cuenta que *Skylake* y *Comet Lake* cuentan con 32KB de caché L1, es claro el impacto que tiene el tamaño del sliver: cuando este se pasa del tamaño del caché, la performance empeora. Para las otras arquitecturas (*Ice Lake* y *Tiger Lake*) se puede observar la misma tendencia (tienen 48KB de caché L1), aunque en un grado mucho menor.

También podemos estudiar el impacto que tiene el tamaño de bloque $\tilde{A}$. Nuestra implementación está diseñada para que éste se lea directamente del caché L2. Podemos estudiar su efecto de forma análoga al caso anterior, sabiendo que cada bloque ocupa $mc \times kc \times 4$ bytes.

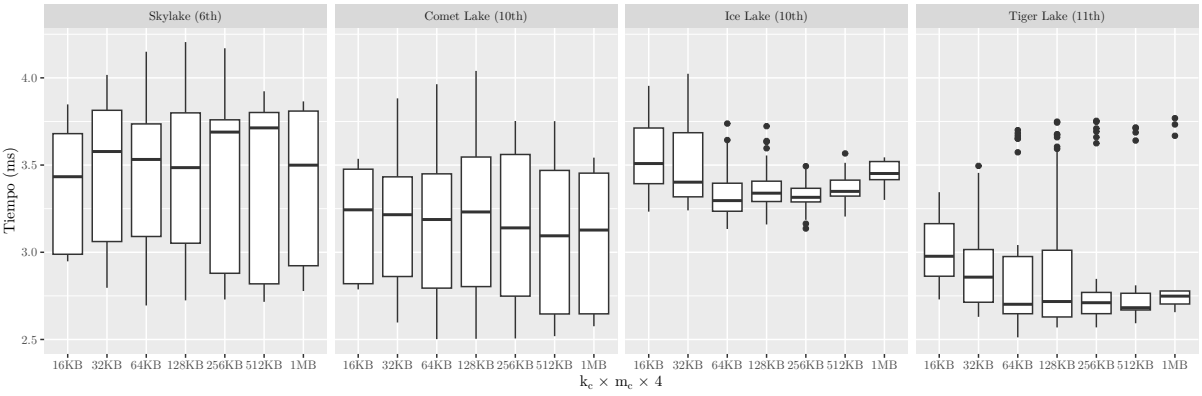

Figura 23: Distribución del tiempo que toma cada multiplicación con respecto al tamaño del bloque $\tilde{A}$.

En este caso, la tendencia no es tan clara: para ninguna de las arquitecturas vemos una diferencia significativa cuando el bloque toma un tamaño que deja de entrar en el caché L2. Suponemos que esto se debe a que las matrices que estamos multiplicando, de 512×512 elementos, no son lo suficientemente grandes para forzar muchos desalojos del caché. Lo mismo sucede para la caché L3 (en ese caso la matriz entra directamente). Por limitaciones técnicas, no evaluamos el rendimiento de matrices más grandes.

4.2.3. Parámetros óptimos

A modo de referencia, las siguientes son las mejores combinaciones que encontramos para nuestra implementación de GEMM:

Generación	k_c	m_c	m_r	n_r	m_v	n_u
Skylake (6th)	256	64	4	16	4	4
Comet Lake (10th)	256	64	4	16	4	8
Ice Lake (10th)	256	64	4	32	2	4
Tiger Lake (11th)	256	64	4	32	4	4

Cuadro 3: Parámetros óptimos que encontramos para diferentes procesadores

4.3. Comparación de GEMM

A continuación, compararemos el rendimiento de nuestra implementación de GEMM con un conjunto de runtimes de ML conocidos:

- **libxsmm** [3]: Esta librería, desarrollada por investigadores de *Intel*, genera código para la multiplicación de matrices optimizado para la CPU (y aprovechando todo el rango de instrucciones SIMD).
- **naïve**: Esta es la implementación clásica de múltiplicación de matrices, donde cada elemento del producto $A \times B$ se computa como el producto interno entre una fila de A y una columna de B.
- **ONNX Runtime** [1]: ONNX Runtime es una librería dedicada a la ejecución de modelos de ML desarrollada por *Microsoft*.
- **TensorFlow** [4]: Esta librería, creada por *Google*, es la más utilizada a la hora de crear, entrenar y correr modelos de inteligencia artificial.

Para comparar entre estas implementaciones, utilizamos multiplicaciones entre matrices cuadradas de tamaños progresivamente más grandes. Estas multiplicaciones (con tamaños iguales) son representativas del entorno de ML, ya que forman la base de las redes neuronales profundas. Al igual que en la sección anterior, corremos 30 iteraciones de calentamiento y promediamos entre otras 300 para obtener una buena estimación del tiempo que tarda cada una.

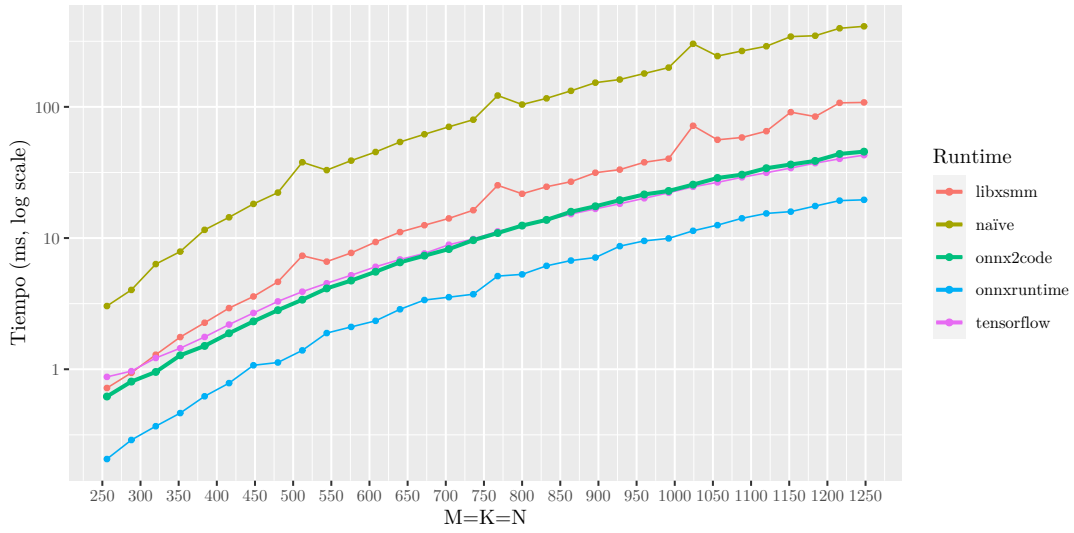

Figura 24: Tiempo que toma cada multiplicación $A \times B$ (con A y B cuadradas y del mismo tamaño) para distintos *runtimes*.

En este gráfico podemos ver que utilizar loop-tiling es ampliamente superior a la multiplicación *naïve*, y aproximadamente equivalente a la que usa *TensorFlow*. El bajo rendimiento de *libxsmm* es esperable, ya que esta librería está optimizada para matrices de tamaño chico. Por otro lado, el performance de *ONNX Runtime* nos indica que aún hay lugar para mejorar nuestra implementación.

4.4. Modelos

Por último, medimos el rendimiento de `onnx2code` para una serie de modelos presentes en el *Model Zoo* de ONNX. También medimos el de *ONNX Runtime*, y el de una versión de modificada de `onnx2code` en la que, en vez de emplear la técnica *im2col*, implementamos la convolución de forma *naïve*.

Modelo	Tamaño	`onnx2code` (*im2col*)	`onnx2code` (*conv-naive*)	`onnxruntime`
MNIST	26 KB	$0{,}329ms$	$0{,}972ms$	$0{,}037ms$
SUPER_RESOLUTION	240 KB	$1{,}7s$	$3{,}0s$	$40{,}2ms$
SQUEEZENET1.1	9 MB	$48{,}8ms$	$233{,}9ms$	$3{,}9ms$
EMOTION_FERPLUS	34 MB	$239{,}6ms$	$893ms$	$7{,}9ms$
INCEPTION-V2	44 MB	$344{,}5ms$	$1{,}7s$	$18{,}3ms$
RESNET50-CAFFE2-V1	98 MB	$659ms$	$2{,}8s$	$30{,}3ms$

Cuadro 4: Tiempo de inferencia para cada modelo, utilizando distintos *runtimes*.

Nuevamente, podemos ver que *ONNX Runtime* es imbatible a la hora de realizar una inferencia. Por otro lado, queda en evidencia la efectividad de la transformación *im2col* para las convoluciones: ese cambio provee mejoras sustanciales.

5. Conclusión

Aprendimos algunas lecciones durante el transcurso de esta investigación:

- El loop tiling es una técnica efectiva para la optimización de loops con una gran cantidad de iteraciones, y no siempre es aplicada automáticamente por los compiladores. Cualquier aplicación de álgebra lineal en búsqueda de un buen rendimiento debería explorarlo como alternativa.
- Los compiladores son muy efectivos a la hora de generar código optimizado cuando toda la información en una función es estática. Tal es el caso que no logramos obtener una mejor implementación para `unit_update`, aún teniendo en cuenta criterios de [9].
- No obstante, es importante proveer todo el contexto posible a los compiladores, mediante *hints* como los atributos `const` y `restrict`: éstos permiten realizar suposiciones que suelen llevar a código más eficiente.
- Sabiendo que la GPU es altamente superior a la hora de llevar a cabo los distintos cálculos necesarios para la inferencia en las redes neuronales, uno podría pensar que enfocarse en la optimización para la CPU no tendría ningún fruto. Sin embargo, teniendo en cuenta las múltiples investigaciones sobre la materia en cuestión presentadas en este trabajo, queda claro que el lugar para librerías como `onnx2code`, que permiten realizar inferencias de redes neuronales en contextos donde no es posible el uso de la GPU, existe y no es tan pequeño como podemos pensar en un principio.

6. Trabajo futuro

A lo largo del trabajo, identificamos algunas áreas que podrían ser exploradas para ampliar el alcance o mejorar el rendimiento de la librería. En ningún orden particular, éstas son:

- **Implementar más operadores**: ONNX ofrece una gran variedad de operadores que no implementamos en este trabajo. Muchos de esos operadores son tan prevalentes que permitirían la inferencia de diversos modelos nuevos. Por ejemplo, sólo implementando el operador LRN (Local Response Normalization) se volvería posible realizar la inferencia de 35 nuevos modelos.
- **Soportar modelos cuantizados**: Cuantizar modelos es una técnica muy útil para reducir drásticamente el tiempo de inferencia y la memoria utilizada por un modelo. Se basa en reducir la precisión (a `FP16`, `int8`, entre otros) de los parámetros, entradas y operaciones intermedias. Los cálculos en estos tipos de datos más acotados permiten aumentar el nivel de vectorización, entre otras ventajas. Esta transformación causa que el modelo pierda un par de puntos de precisión a cambio de utilizar menos recursos. Como `onnx2code` apunta a sistemas con pequeña capacidad de cómputo y memoria restringida es una buena idea incorporar esta transformación.
- **Abstracción común para tiling de MaxPool2D y Conv2D**: Las operaciones de convolución y *max-pooling* tienen una estructura extremadamente similar, que además es amigable a técnicas como loop tiling. Debido a la prevalencia de ambas, podría ser efectivo explorar una implementación similar a la de GeMM.
- **Utilizar pre-fetching en GeMM**: Durante nuestra desarrollo, intentamos hacer uso de las instrucciones de `prefetch`, que permiten indicar al procesador que ciertos valores se utilizarán próximamente, y por ende deben ser cargados a un nivel dado de caché. Sin embargo, todos nuestros pruebas resultaron en una degradación del rendimiento. Como el prefetch es efectivamente utilizado en múltiples implementaciones de BLAS, podría valer la pena revisitar esta idea.
- **Optimizar** `unit_update`: Como mencionamos anteriormente, intentamos optimizar el núcleo de la operación GeMM mediante las técnicas descriptas en [9], pero decidimos que estaba fuera de el alcance del trabajo. El paper propone generar microkernels que maximizan el throughput de instrucciones, reemplazando cargas de memoria por shuffles.

7. Bibliografía

[1] ONNX Runtime developers. *ONNX Runtime*. https://onnxruntime.ai/. Version: x.y.z. 2021.

[2] Robert van de Geijn y Kazushige Goto. «Anatomy of high-performance matrix multiplication Kazushige Goto, Robert A. van de Geijn ACM Transactions on Mathematical Software (TOMS), 2008». En: *ACM Transactions on Mathematical Software* 34 (mayo de 2008), Article 12. DOI: 10.1145/1356052.1356053. URL: https://www.cs.utexas.edu/users/flame/pubs/GotoTOMS_final.pdf.

[3] Alexander Heinecke y col. «LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation». En: nov. de 2016, págs. 981-991. DOI: 10.1109/SC.2016.83. URL: http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/poster_files/post137s2-file2.pdf.

[4] Martín Abadi y col. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: https://www.tensorflow.org/.

[5] NumPy. *NumPy - General Broadcasting Rules*. URL: https://numpy.org/doc/stable/user/basics.broadcasting.html#general-broadcasting-rules.

[6] ONNX. *GitHub - onnx/models: A collection of pre-trained, state-of-the-art models in the ONNX format*. URL: https://github.com/onnx/models.

[7] *Optimizing TensorFlow Lite Runtime Memory*. en. URL: https://blog.tensorflow.org/2020/10/optimizing-tensorflow-lite-runtime.html (visitado 22-02-2023).

[8] Yury Pisarchyk y Juhyun Lee. *Efficient Memory Management for Deep Neural Net Inference*. 2020. eprint: arXiv:2001.03288. URL: https://arxiv.org/pdf/2001.03288.pdf.

[9] Richard Veras y col. «Automating the Last-Mile for High Performance Dense Linear Algebra». En: (nov. de 2016). URL: https://arxiv.org/pdf/1611.08035.pdf.

[10] Pete Warden. *Why GEMM is at the heart of deep learning*. URL: https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning.