Universidad Zaragoza

Trabajo Fin de Grado

Aceleradores Hardware para Visión por Computador
Hardware Accelerators for Computer Vision

Autor

Alberto Álvarez Aldea

Directores

Dra. Ana Cristina Murillo Arnal

Dr. Darío Suárez Gracia

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2017

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Alberto Álvarez Aldea,

con nº de DNI 72994106H en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Aceleradores Hardware para Visión por Computador

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 22 de Junio de 2017

Fdo: 

AGRADECIMIENTOS

A mis padres y mi hermana por su apoyo incondicional,
a Lucía por su tiempo y su paciencia,
a Darío y Ana Cris por su dedicación y ayuda.

Aceleradores Hardware para Visión por Computador

RESUMEN

La visión por computador es una rama de la inteligencia artificial que estudia el análisis y procesado automático de imágenes con la finalidad de obtener información de interés para distintas tareas. Gracias al bajo coste de los sensores de visión y a la capacidad de cálculo de los microprocesadores, el uso de visión por computador se ha extendido de manera extraordinaria en los últimos años.

La extracción de características en imágenes y las redes neuronales convolucionales (CNNs) son dos algoritmos muy importantes para muchas de las aplicaciones con más impacto de visión por computador. Estos algoritmos requieren altas prestaciones computacionales y en múltiples ocasiones deben ejecutarse en sistemas embebidos, por lo que requieren un consumo de energía mínimo. El uso masivo de estos algoritmos junto con las prestaciones requeridas, y la necesidad de un uso reducido de energía y el fin del escalado de la tecnología CMOS, han motivado que se desarrollen múltiples coprocesadores de propósito específico (aceleradores *hardware*), en especial para CNNs.

El principal objetivo de este trabajo consiste en el estudio y la extensión de aceleradores *hardware* diseñados para ejecutar CNNs, con la finalidad de que sean también capaces de ejecutar algoritmos de extracción de características de forma eficiente sin reducir las prestaciones de las CNNs. Para ello, en primer lugar se ha realizado una caracterización detallada de los algoritmos en procesadores de propósito general, para analizar los requisitos computacionales y cuellos de botella. Dada la complejidad de la tarea de integrar nuevos algoritmos en un acelerador, la siguiente parte de este trabajo consiste en proponer una metodología para la integración de nuevos algoritmos en aceleradores *hardware*. Para validar la metodología propuesta, este trabajo presenta los resultados de aplicarla para evaluar la integración del detector de características ORB en el acelerador de CNNs PuDianNao. Estos experimentos muestran que la integración de ORB en dicho acelerador consigue un aumento del rendimiento de 309.4× y una reducción del consumo energético de 335.9×, con respecto a un procesador de propósito general de última generación. Por último, se propone una mejora de dicho acelerador que, como muestran los experimentos, aumenta el rendimiento del algoritmo de extracción de ORB en 2×.

Índice

Capítulo 1

Introducción

1.1. Motivación

La visión por computador se enmarca dentro de la inteligencia artificial y su fin es el análisis y procesado automático de imágenes buscando obtener información de interés para distintas tareas, por ejemplo, detectar obstáculos y señales para navegación autónoma de vehículos o identificar personas mediante análisis de caras.

Gracias al bajo coste de los sensores de visión y a la capacidad de cálculo de los microprocesadores, el uso de visión por computador se ha extendido de manera extraordinaria en los últimos años. Estos avances han producido la proliferación de técnicas complejas de aprendizaje automático en aplicaciones de visión, dando lugar a resultados inimaginables hace unos años.

Los algoritmos de extracción de características y las redes neuronales convolucionales (CNNs) son dos de los ingredientes principales en la mayoría de los algoritmos básicos de las aplicaciones de reconstrucción automática en 3D y reconocimiento visual automático. Estos algoritmos requieren altas prestaciones computacionales, entre las que cabe destacar: los complejos cálculos matemáticos de las técnicas de *deep learning*, o la elevada transferencia de datos ,debido a la frecuencia de procesado de imágenes necesaria, para los algoritmos de realidad virtual y navegación autónoma. Además, en múltiples ocasiones deben ejecutarse en sistemas embebidos por lo que requieren un consumo de energía mínimo.

El uso masivo de estos algoritmos junto con las prestaciones requeridas, la necesidad de un uso reducido de energía y el fin del escalado de la tecnología CMOS [1], han motivado que se desarrollen múltiples coprocesadores de propósito especifico (aceleradores *hardware*) tanto en la academia [2, 3] como en la industria [4]. La mayoría de estos aceleradores están enfocados hacia los algoritmos de *deep learning* debido al auge de este campo.

1.2. Objetivos y tareas

El principal objetivo de este proyecto consiste en el estudio y la extensión de aceleradores *hardware* diseñados para ejecutar redes neuronales convolucionales (CNNs), con la finalidad de que sean también capaces de ejecutar algoritmos de extracción de características de forma eficiente. En concreto, el objetivo es la aceleración de la extracción de características (reducción del tiempo de computo, consumo energético, ancho de banda consumido, ...), sin reducir las prestaciones de las CNNs.

Para conseguir este objetivo, en este proyecto se han planteado las siguientes tareas, con el alcance y objetivo descrito a continuación:

- Estudio del estado del arte tanto a nivel de algoritmos de visión por computador y CNNs, como de aceleradores *hardware* para este propósito. Esto permitirá la selección de los algoritmos a acelerar y los aceleradores *hardware* a tomar como punto de partida.

- Caracterización de los algoritmos seleccionados para analizar su comportamiento y determinar las limitaciones computacionales. Esto permitirá asegurar que los algoritmos pueden ser ejecutados mejorando las prestaciones en el acelerador seleccionado.

- Establecimiento de una metodología para estudiar la viabilidad de la integración de algoritmos en aceleradores.

- Adaptación del algoritmo de extracción de características ORB, al acelerador *hardware* para *machine learning* PuDianNao.

- Modificación del acelerador *hardware* PuDianNao para mejorar todavía más la ejecución del algoritmo de ORB.

Las principales contribuciones de este trabajo consisten en:

- Caracterización y análisis de CNNs y de ORB.

- Establecimiento de una metodología para estudiar la viabilidad de la integración de algoritmos en aceleradores.

- Integración de ORB en el acelerador PuDianNao consiguiendo una aceleración de 309.4× y un ahorro energético de 335.85× respecto a un procesador de propósito general de última generación.

- Presentación de dos propuestas de modificación de PuDianNao para la aceleración de ORB. Una propuesta que obtiene una aceleración de 1.18× y un ahorro energético de 1.06×, y otra propuesta que obtiene una aceleración de 2.11× y un ahorro energético de 1.11×.

1.3. Descripción del contenido

El capítulo 2 describe el estado del arte correspondiente a los algoritmos de visión por computador y de los aceleradores *hardware* para dichos algoritmos. El capítulo 3 presenta la caracterización y el análisis de los algoritmos descritos en el capitulo anterior. El capítulo 4 presenta una metodología para estudiar la viabilidad de la integración de algoritmos en aceleradores. El capítulo 5 describe la integración del ORB en PuDianNao empleando la metodología propuesta y presenta dos propuestas de mejora del acelerador para incrementar todavía más las prestaciones de ORB. El capítulo 6 presenta las principales conclusiones extraídas y el trabajo futuro a desarrollar.

Capítulo 2

Trabajo relacionado

Este capítulo resume los conceptos teóricos más importantes para este trabajo. Por un lado, los algoritmos de visión por computador estudiados (2.1), por otro, el estado del arte en técnicas de aceleración *hardware* (2.2).

2.1. Algoritmos de visión por computador

Como se ha comentado anteriormente, este trabajo se centra en algoritmos de extracción de características locales y CNNs, ya que son dos de los ingredientes principales en las técnicas de visión por computador más utilizadas hoy en día por numerosas aplicaciones.

2.1.1. Características locales de imágenes

La detección de características locales es un proceso que consiste en la localización de puntos característicos y distintivos de una imagen. Esta técnica permite desde tareas de reconocimiento de objetos a la reconstrucción de entornos mediante la búsqueda de correspondencias entre varias imágenes. En la figura 2.1 se muestra un ejemplo de correspondencias encontradas de manera automática.

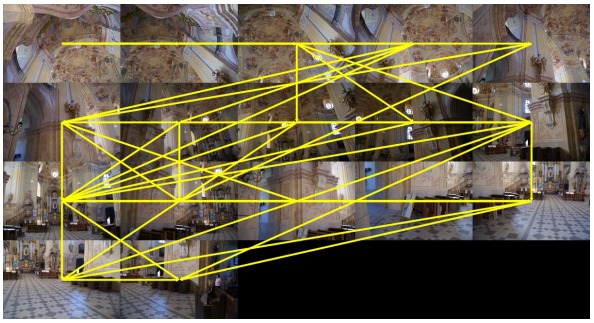


Figura 2.1: Correspondencias (marcadas con rectas) entre múltiples imágenes de la misma escena usando características locales (puntos marcados en cada imagen).[1]

A lo largo de los años se han desarrollado múltiples algoritmos para la detección y descripción de características. Desde los primeros extractores de *esquinas* [5] hasta el conocido SIFT [6], que proponía detectores mucho más invariantes que los anteriores y dio un gran impulso al reconocimiento visual automático, hasta versiones más eficientes como SURF [7], FAST [8] u ORB [9] que permiten la ejecución en tiempo real.

Oriented FAST and rotated BRIEF(ORB) ha sido un punto de inflexión en los últimos años, debido a que ofrece una precisión similar al detector SIFT pero con un coste computacional dos órdenes de magnitud menor. ORB es ampliamente utilizado debido a que ha facilitado la realización de aplicaciones como SLAM [10] o reconstrucción densa de espacios en 3D en tiempo real [11]. El algoritmo puede dividirse en las siguientes seis fases (detalladas en el anexo A), división que usaremos para los análisis posteriores:

1. Conversión de color a escala de grises
2. Creación de pirámides
3. Uso del detector FAST para localizar puntos de interés
4. Refinamiento mediante Harris
5. Obtención de la orientación
6. Cálculo de los descriptores de cada punto característico

2.1.2. Redes neuronales convolucionales

Las redes neuronales son un paradigma de procesamiento inspirado en como el cerebro procesa la información, aplicado en el ámbito de la inteligencia artificial y el aprendizaje automático. Una red neuronal está compuesta de un conjunto de elementos interconectados (neuronas) que procesan la información hasta alcanzar una respuesta. Las redes neuronales se han utilizado desde hace mucho tiempo para múltiples aplicaciones de aprendizaje automático [12]. Recientemente, se han obtenido grandes avances en este tipo de técnicas gracias al campo del *deep learning* [13].

Las *Deep Neural Networks* son redes neuronales que contienen una gran cantidad de capas ocultas. Un tipo específico de estas redes neuronales, son las *Convolutional Neural Networks* (CNNs). Estas redes están siendo ampliamente utilizadas en muchos campos, definiendo un nuevo estado del arte en numerosas aplicaciones de visión por computador, por ejemplo el modelo conocido como *AlexNet* [14] es una de las primeros propuestas en mostrar una mejora impresionante en reconocimiento de objetos utilizando CNNs.

[1]Fuente imagen: `http://www.imagingshop.com/images/sharpstitch/image-matching.jpg`

Las CNNs están formadas por diferentes capas: *Convolutional Layer*, *Pooling Layer*, *Normalization Layer* y *Fully-Connected Layer*. Estas redes reciben como entrada una imagen que se procesa a través de varias capas del tipo *Convolutional*, *Pooling* y *Normalization*, para normalmente acabar por alguna capa *Fully-Connected*. La figura 2.2 muestra el procesado de una imagen en una CNN.

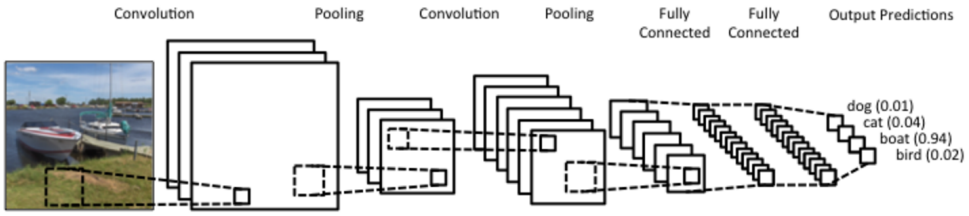


Figura 2.2: Ejemplo de clasificación mediante una red neuronal convolucional. El flujo de ejecución va de izquierda a derecha. En ese orden, se encuentran dos capas *convolutional* y dos de *pooling*, posteriormente dos capas *fully-connected* y finalmente los resultados de la clasificación.[2]

A continuación, se describe la capa de tipo *Convolutional* ya que es la que consume prácticamente la totalidad del tiempo de ejecución y tiene un gran peso en este trabajo, mientras que las otras capas se describen en el anexo B. Hay que notar que los análisis de este trabajo se centran en la parte de evaluación de una imagen a través de la red (*forward-pass*), que son los que se realizan en tiempo de evaluación, no en los pasos iterativos necesarios durante el entrenamiento (*backward-pass*).

Convolutional Layer. Es la capa principal de las CNNs, y como su nombre indica está basada en la convolución. La ecuación 2.1 muestra la definición matemática de la convolución de dos funciones (f y g), la cual se denota con el símbolo $*$ y consiste en la integral del producto de ambas funciones después de desplazar una de ellas una distancia (τ).

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau \tag{2.1}$$

La convolución consiste en la aplicación de diferentes filtros (*kernels*) a las entradas de la capa. Estos filtros de aplican con un *stride*(S) y un *padding*(P). En la figura 2.3 se muestra la convolución de una imagen de forma más gráfica.

[2]Fuente imagen: https://www.clarifai.com/technology
[3]Fuente imagen: http://xrds.acm.org/blog/2016/06/convolutional-neural-networks-cnns-illustrated-explanation/

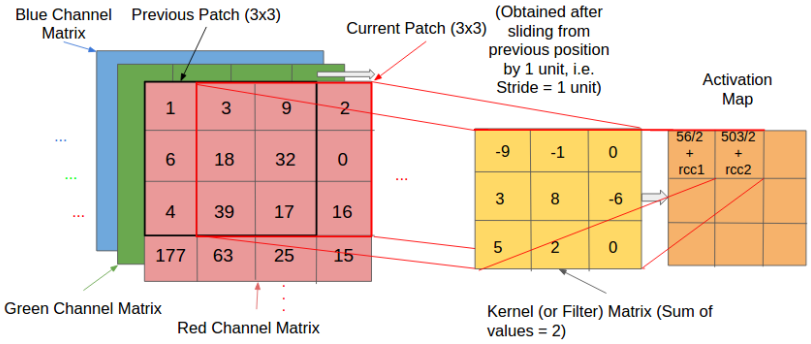


Figura 2.3: Convolución de una imagen con tres canales (RGB). A la izquierda se muestran los tres canales, resaltando dos *patches* (*previous* y *current*). En amarillo se muestra el *kernel* utilizado, y en naranja el resultado parcial después de aplicarlo sobre los dos *patches*.[3]

2.2. Aceleradores Hardware

El alto coste computacional de los algoritmos de visión por computador ha hecho que surjan múltiples aceleradores para aplicaciones de este tipo, y a lo largo de los años se han ido presentando diferentes aceleradores para extracción de características y *machine learning* [15]. Con la proliferación de las CNNs este fenómeno se ha extendido y ha significado un punto de expansión para el desarrollo de aceleradores. Desde entonces se han propuesto diversos diseños, siendo los más famosos los pertenecientes a la familia DianNao, formada por: DianNao [16], PuDianNao [3] y DaDianNao [17, 18]. La figura 2.4 muestra el árbol genealógico de la familia.

Estos aceleradores tienen la finalidad de obtener alta eficiencia en su dominio específico mediante la simplificación del control y la extracción masiva de paralelismo. A alto nivel su organización básica consiste en una segmentación de varias capas con dos *buffers* al inicio y uno al final que se comunican con el procesador y la memoria principal a través de un DMA.

El primer acelerador fue DianNao, el cual está enfocado concretamente en la aceleración de CNNs. DaDianNao, es una versión posterior enfocada a la aceleración

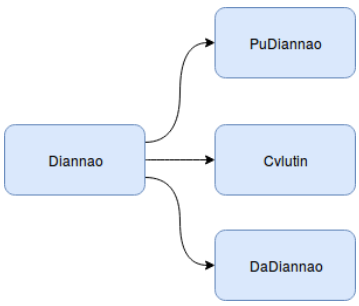
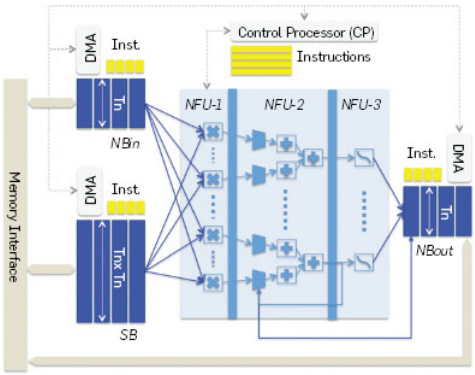
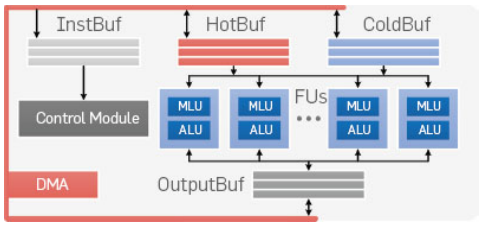


Figura 2.4: Árbol genealógico de la familia DianNao y aceleradores derivados

para entornos de supercomputación. Posteriormente, los mismos autores presentaron un acelerador que combina la aceleración de CNNs y de múltiples algoritmos de *machine learning*, PuDianNao. Sobre la arquitectura de DianNao original se han realizado múltiples mejoras, como la eliminación de las multiplicaciones por cero en Cvltin [2]. La figura 2.5 muestra la arquitectura de DianNao y PuDianNao.



(a) Diannao original



(b) PuDiannao

Figura 2.5: Arquitectura de los aceleradores de la familia *X*DianNao

Capítulo 3

Caracterización y análisis de ORB y CNN

A la hora de trabajar con un acelerador *hardware* es crítico conocer cual es el componente del procesador que limita el rendimiento de la aplicación para que el acelerador mejore sus prestaciones.

Este capítulo presenta la caracterización y análisis de los algoritmos en los que se centra el estudio: algoritmo de extracción de características ORB y descripción/clasificación de una imagen con redes neuronales convolucionales, CNNs. El capítulo se divide en tres secciones. La sección 3.1 describe las cargas de trabajo estudiadas, desarrolladas y utilizadas. La sección 3.2 explica y describe la metodología utilizada para la realizar la caracterización. La sección 3.3 presenta y analiza los resultados obtenidos.

3.1. Diseño y desarrollo de los *benchmarks*

Existen múltiples conjuntos de *benchmarks* (programas de prueba) para la evaluación de algoritmos de visión por computador y aprendizaje automático, como el San Diego Benchmark Suite [19], CortexSuite [20], o Fathom [21]. Sin embargo, estos conjuntos no incluyen *benchmarks* para evaluar tanto la extracción de características mediante ORB como las CNNs. Por ello, se decidió crear un *benchmark* para cada algoritmo.

Los criterios de diseño en la realización de los programas de prueba fueron: asegurar la representatividad de los resultados, su fácil reproducibilidad, uso de los mismos conjuntos de entrada que a su vez son representativos, y la mínima actividad del sistema operativo y de entrada/salida. En ambos *benchmarks* se separa la fase de carga de imágenes de la de cálculo y luego se ejecuta la parte representativa durante 1 millón de veces. Finalmente, el *benchmark* de CNNs permite la evaluación de 2 modelos de redes estándar: CaffeRef [22] y AlexNet [22]. Ya que ambos algoritmos son fácilmente paralelizables a nivel de imagen, se decidió que ambos *benchmarks* utilizaran

únicamente un hilo. Siguiendo el criterio de representatividad, los *benchmarks* emplean las bibliotecas más utilizadas tanto en investigación como en la industria; OpenCV para la parte de visión[23] y Caffe para las CNN [24]. Los *benchmarks* pueden descargarse de `github.com/albert17/benchmarks`.

3.2. Metodología

A la hora de caracterizar una carga de trabajo es necesario fijar dos aspectos, por un lado la plataforma donde se ejecutarán los experimentos y por otro la metodología y las métricas a emplear para analizar los resultados.

3.2.1. Entornos de ejecución: Skylake & Jetson

La ejecución de los *benchmarks* desarrollados se ha realizado en dos entornos representativos: una plataforma de computación de altas prestaciones, Skylake, y un sistema empotrado de bajo consumo, Jetson. Los detalles de ambos entornos se muestran en la tabla 3.1.

	Skylake	**Jetson**
Sistema Operativo	CentOS 6.1	Ubuntu 14.04
CPU	Intel Skylake	ARM A-15
Número Cores	4	4
Multihilo	si, 2	no
Cache nivel L1	4x64KB	4x64KB
Cache nivel L2	4x256KB	1x2MB
Cache nivel L3	1x8MB	-
Memoria Principal (RAM)	64 GB	2 GB
Disco	1TB HD	16GB HD

Tabla 3.1: Plataformas consideradas como entornos de evaluación.

3.2.2. Métricas utilizadas

En la metodología propuesta se ha seguido una aproximación de arriba a abajo en la que primero se han analizado los tiempos de ejecución totales y después se ha caracterizado los tipos de instrucciones, para terminar profundizando en los distintos componentes de un microprocesador que podían limitar el rendimiento. Para ejecutar los *benchmarks*, se han tomado múltiples imágenes como entrada y se ha observado que no había variabilidad en los resultados.

Para la obtención de las métricas se utilizan tanto medidas de tiempos como contadores *hardware*, ya que por construcción los *benchmarks* no sobrecargan el sistema

operativo y permiten ver los recursos *hardware* más empleados, que serán los candidatos a acelerar. Para poder acceder a los contadores *hardware* ha sido necesario el uso de las herramientas *perf* [25] y *perfmon* [26], junto con los manuales técnicos oficiales de los procesadores [27, 28].

Tiempo de ejecución. Esta es la métrica más básica empleada y analiza la duración de la ejecución de los *benchmarks*. Indica si es necesario acelerar una aplicación o no.

Distribución de instrucciones. Una característica fundamental de un programa es la distribución de instrucciones, ya que permite empezar a acotar las razones de la falta de rendimiento. Por ejemplo: si hay muchas accesos a memoria o muchas operaciones con enteros.

Análisis microarquitectónico. Para determinar los cuellos de botella dentro de los elementos del procesador (*frontend*, *backend* y memoria, ...) se ha empleado la metodología propuesta por Intel[29], la cual ha sido adaptada para su utilización en el procesador ARM. Tanto la metodología como su adaptación a ARM pueden encontrarse detalladas en el Anexo C.

La figura 3.1 muestra los componentes de un procesador agrupados en el *frontend* y *backend*. El *frontend* se encarga de leer, decodificar y preparar la ejecución de instrucciones y el *backend* se encarga de ejecutar y retirar las instrucciones.

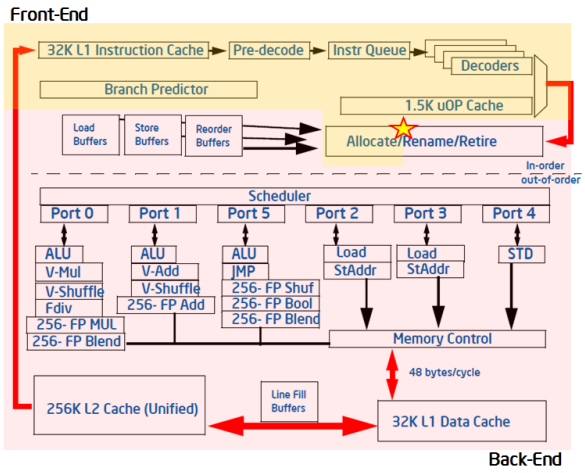


Figura 3.1: Diagrama microarquitectónico de un procesador con sus componentes agrupados en 2 bloques: *backend* y *frontend*. La estrella en la lógica de asignación/jubilación representa la frontera entre el *backend* y el *frontend*.

3.3. Resultados

3.3.1. Tiempo de Ejecución

La figura 3.2 muestra el tiempo de ejecución consumido en el procesado de una imagen en un pase *forward* de una CNN (utilizando dos modelos distintos) y en la extracción de características ORB en ambas plataformas. Como puede observarse el tiempo de ejecución de las CNNs es mucho mayor que el de ORB, ya que las CNN realizan muchas más operaciones para procesar una imagen, siendo los tiempos menores en la estación de trabajo (Skylake).

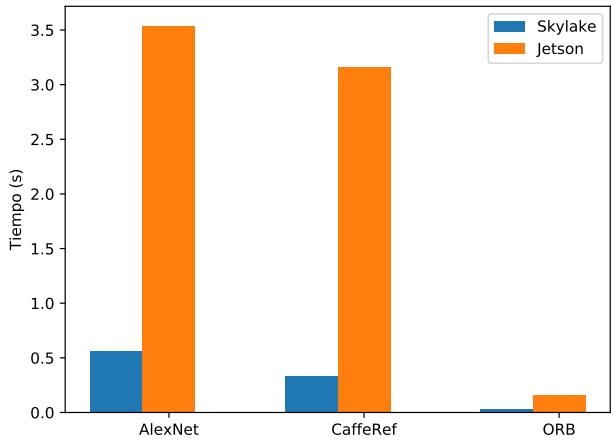


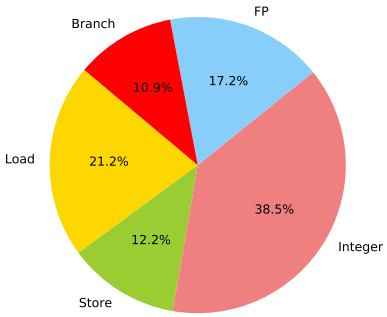
Figura 3.2: Tiempo de ejecución procesando una imagen con distintos algoritmos y plataformas

Conclusión. Pese a que el tiempo en ORB parece pequeño, 0.16 segundos en Jetson, debe tenerse en cuenta la frecuencia de ejecución de dicha tarea que suelen necesitar las aplicaciones relacionadas. Por ejemplo, un sistema de realidad aumentada puede fácilmente requerir extraer características en 90 imágenes por segundo (y no solo extraer, sino el procesado posterior de las mismas), mientras que Jetson únicamente podría analizar 6. Esto confirma que es necesaria la utilización de aceleradores dedicados en multitud de entornos.

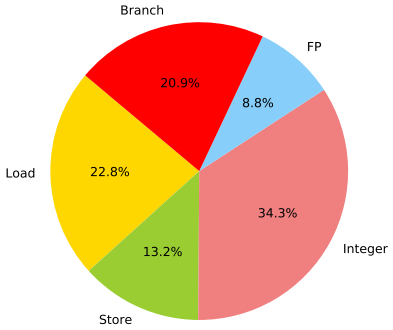
3.3.2. Distribución de instrucciones

La figura 3.3 muestra la distribución de instrucciones: *Load*, *Store*, *Integer* (enteros), *FP* (coma flotante), *Branch* (saltos), utilizadas en la ejecución de los dos modelos de CNN (*AlexNet* y *CaffeRef*) y de ORB en las dos plataformas estudiadas. La distribución de instrucciones es similar en ambas plataformas por lo que podemos considerarla

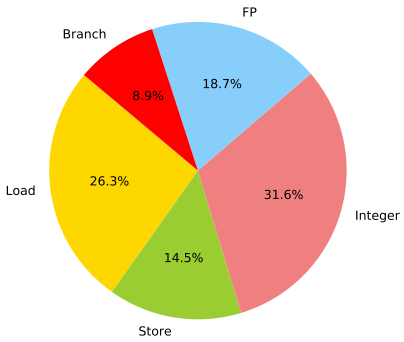
independiente de la arquitectura. Otra diferencia significativa, es que ORB no hace prácticamente uso de operaciones en coma flotante sino que casi la totalidad son en enteros, lo cual se debe a que al inicio convierte las imágenes a escala de grises y opera con ellas el resto del algoritmo.



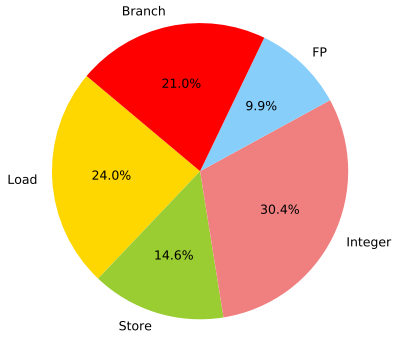
(a) *AlexNet* en Skylake



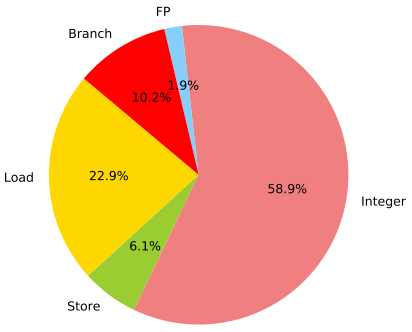
(b) *AlexNet* en Jetson



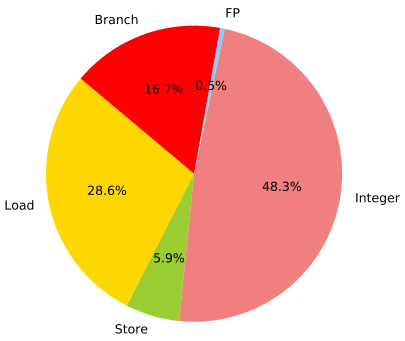
(c) *CaffeRef* en Skylake



(d) *CaffeRef* en Jetson



(e) *ORB* en Skylake



(f) *ORB* en Jetson

Figura 3.3: Distribución de instrucciones para todas las combinaciones de algoritmos y plataformas consideradas.

Conclusión. Como la distribución de instrucciones es similar en los algoritmos de procesado de ORB y CNN, esto permite entrever que un acelerador de CNNs podría llegar a ejecutar ORB de forma satisfactoria.

3.3.3. Análisis microarquitectónico

Visto que el tiempo de ejecución es relativamente grande para muchos casos de uso y que la distribución de instrucciones es bastante homogénea, es necesario profundizar en los componentes del procesador para encontrar las partes a acelerar. El modelo empleado en el análisis se basa en una clasificación multinivel de los ciclos de ejecución del procesador.

Nivel 1: Distribución de ciclos de ejecución. El objetivo es ver la cantidad de ciclos en los que el procesador retira instrucciones (trabajo útil) y la cantidad de ciclos en los que no se ha hecho trabajo útil. Este nivel está dividido en cuatro categorías:

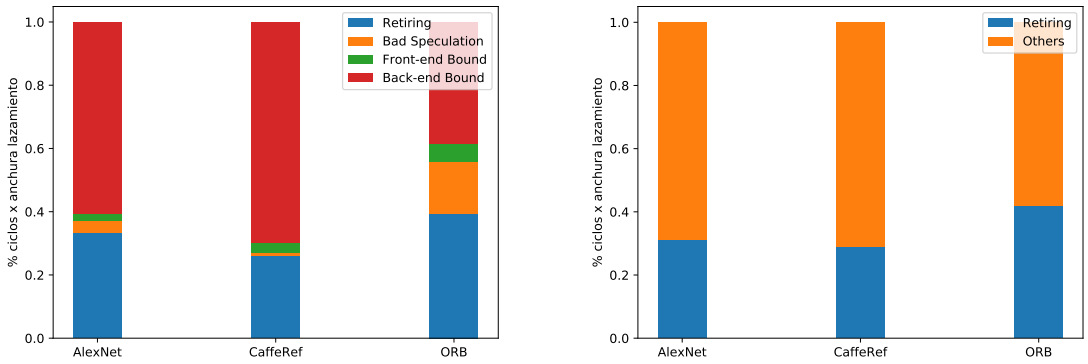
- *Retiring*: % ciclos en los que se han retirado o completado instrucciones útiles.
- *Frontend Bound*: % ciclos donde no se completan instrucciones porque no llegan a la fase de ejecución.
- *Backend Bound*: % ciclos perdidos mientras las instrucciones se están ejecutando.
- *Bad Speculation*: % ciclos perdidos en fallos de predicción y limpieza del *pipeline*.

La figura 3.4 muestra la clasificación de este nivel. Como puede observarse, el mayor cuello de botella en todos los casos se encuentra en el *backend* del procesador. En el caso de Jetson, el procesador ARM A-15 no dispone de los contadores hardware necesarios para distribuir los ciclos y por eso sólo hay 2 categorías. Con independencia de la plataforma y del *benchmark*, *Retiring* tiene un valor pequeño (entre 0.26 y 0.42) por lo que ambas plataformas no utilizan bien buena parte de los recursos y la mayor parte de perdida de prestaciones se debe al *backend* (entre 0.38 y 0.69).

Nivel 2: *Back-end*. Al bajar un nivel en el análisis es necesario centrarse en el *Backend*, cuyo análisis se muestra en la figura 3.5. Este análisis distingue dos categorías:

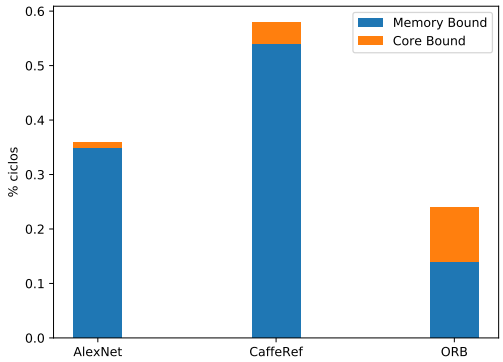
- *Memory Bound*: Retrasos del procesador debido a la jerarquía de memoria.
- *Core Bound*: Retrasos del procesador debido a unidades funcionales.

La memoria afecta más a CNN que a ORB por el tamaño de las redes respecto al de las imágenes y el *core* afecta más a ORB, ya que tiene más intensidad aritmética.



(a) Skylake (b) Jetson

Figura 3.4: Clasificación de los ciclos consumidos en el nivel más alto (*Top level*) durante la ejecución de las aplicaciones. Este nivel tiene cuatro categorías: *Retiring*, *Bad Speculation*, *Frontend* y *Backend*.

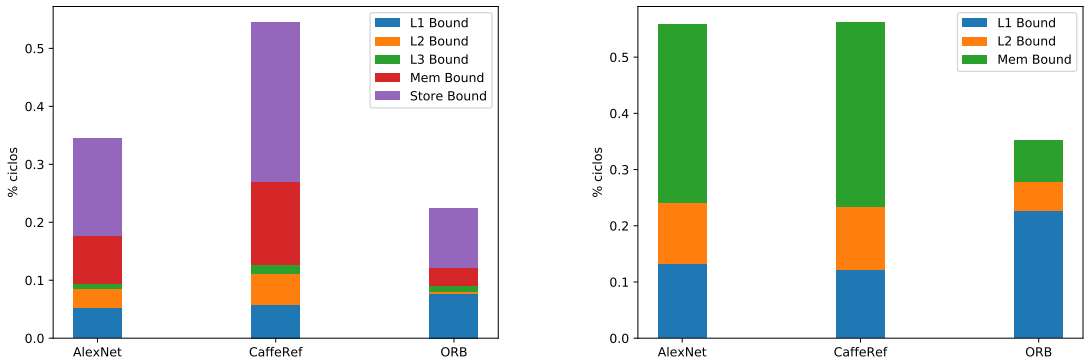


(a) Skylake

Figura 3.5: Clasificación de los ciclos consumidos en el *backend level* durante la ejecución de las aplicaciones. Este nivel tiene dos categorías: *Memory Bound* y *Core bound*.

Nivel 3: Memoria. El siguiente nivel se centra en los accesos a memoria, en el se profundiza en los ciclos que componen la limitación *Memory Bound*. El resumen de este análisis se puede ver en la figura 3.6. Este nivel se divide en cinco categorías:

- *L1 Bound*: % ciclos perdidos por espera de datos de la memoria cache de nivel 1.
- *L2 Bound*: % ciclos perdidos por espera de datos de la memoria cache de nivel 2.
- *L3 Bound*: % ciclos perdidos por espera de datos de la memoria cache nivel 3.
- *MEM Bound*: % ciclos perdidos por espera de datos desde memoria principal.
- *Store Bound*: % ciclos perdidos por espera de la escritura de datos.



(a) Skylake (b) Jetson

Figura 3.6: Clasificación de los ciclos consumidos en el nivel de memoria (*Memory level*) durante la ejecución de las aplicaciones. El nivel se divide en cinco categorías: *L1 Bound*, *L2 Bound*, *L3 Bound*, *MEM Bound*, *Store Bound*.

Conclusión. Como puede observarse las redes neuronales están limitadas en gran parte por la memoria principal, mientras que en ORB la limitación se encuentra en la cache de primer nivel y las escrituras en memoria. Por lo tanto un acelerador capaz de ejecutar CNN y ORB necesita de un gran ancho de banda en memoria y muchas unidades funcionales para maximizar el rendimiento.

Capítulo 4

Metodología para integrar y evaluar nuevos algoritmos en aceleradores

Este capítulo presenta la metodología propuesta en este trabajo para estudiar si es viable integrar un algoritmo en un acelerador *hardware*.

El uso de aceleradores permite aumentar en varios órdenes de magnitud el rendimiento y la eficiencia energética de algoritmos a cambio de perder flexibilidad en la programación. La mayor limitación a la hora de hacer estudios con aceleradores es que en general no se dispone ni de un entorno de simulación, ni de la definición de la arquitectura para construir uno, ni de un compilador para generar código desde un lenguaje de alto nivel. Por lo tanto, establecer con celeridad si un acelerador puede ejecutar un algoritmo determinado se vuelve una tarea muy compleja, pero tan compleja como necesaria.

Por fortuna, los aceleradores *hardware* suelen prescindir de las características más complejas de los procesadores, como la ejecución fuera de orden y la especulación. Por lo tanto, los modelos analíticos, como los que se proponen en este trabajo, son mucho más sencillos y más precisos que los de un procesador de propósito general.

Durante este trabajo se estudiaron e intentaron aplicar otras opciones para realizar la integración, como el uso de herramientas específicas como Aladdin [30] o el uso de simuladores microarquitectónicos como gem5 [31] o gem5-aladdin [32]. Sin embargo, tuvieron que ser desechadas por la incapacidad de simular un acelerador *hardware*, el bajo realismo y el coste de desarrollo.

4.1. Metodología

4.1.1. Visión general

La figura 4.1 muestra el diagrama de la metodología propuesta. El primer paso, **caracterización**, consiste en caracterizar el algoritmo seleccionado para ver si su

ejecución tiene margen de mejora con respecto a un procesador de propósito general. Para ello se realizaran múltiples análisis como se ha visto en el capítulo 3. El segundo paso, **selección**, comprueba si se puede emplear un acelerador existente o si es necesario realizar uno nuevo. Completado este paso se deberá integrar el algoritmo en el acelerador, **integración**. Este paso consiste en programar en el ensamblador del acelerador el algoritmo elegido o en emularlo. En general, se intentará dividir el algoritmo en fases para simplificar la programación y comprobar si puede haber solapamiento entre fases, y así aumentar el *throughput*. El siguiente paso, **cuantificación**, realiza modelos temporales y energéticos de la ejecución. Para aumentar la precisión de los modelos, en especial del energético, y permitir evaluar el impacto de los datos de entrada en tiempo y energía ,se construirá una herramienta con el instrumentador dinámico de binarios Intel Pin [33]. Esta herramienta se utilizará para determinar el uso de cada componente del acelerador, ya que estos contadores ayudan a determinar con relativa precisión el consumo. Finalmente, a la vista de los resultados se podrán realizar propuestas de mejora sobre el acelerador, **mejora**.

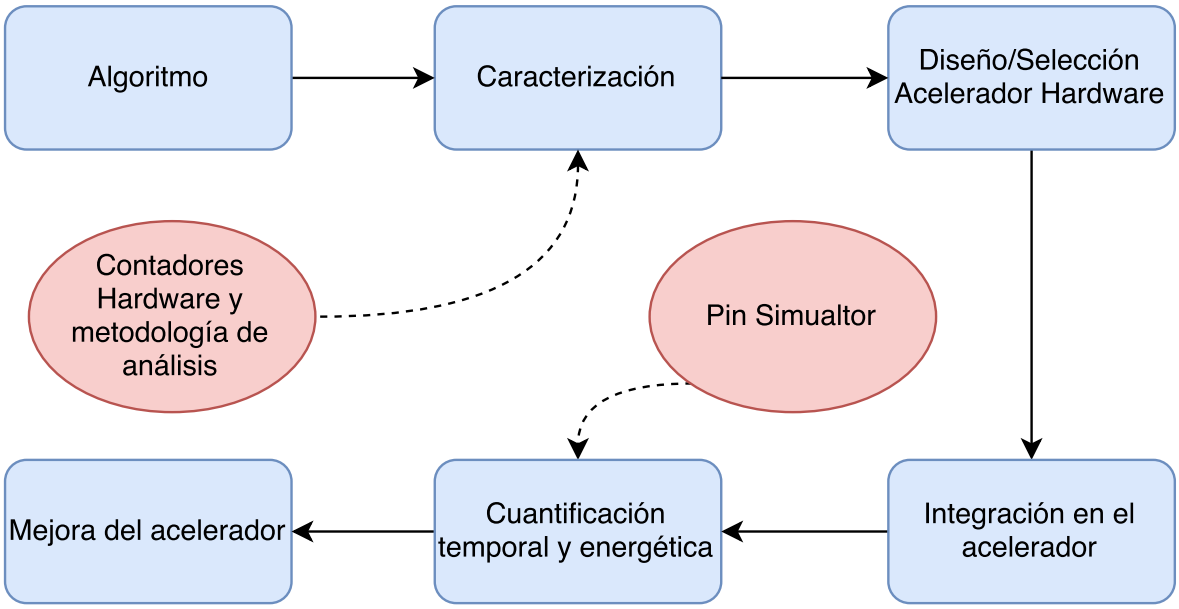


Figura 4.1: Metodología diseñada para la integración de algoritmos en aceleradores y mejora de los mismos

4.1.2. Estimación de tiempo y energía

Tiempo. A la hora de realizar estimaciones temporales con aceleradores, se puede asumir que estos últimos funcionan como coprocesadores, o huéspedes, de una CPU principal quien se encarga de enviar de manera continuada datos al acelerador para que este los pueda consumir y devolver los resultados.

En un acelerador bien diseñado y balanceado, el tiempo (T) se calcula como el tiempo de ciclo (T_c) multiplicado por el número de ciclos que requerirá la ejecución del algoritmo, asumiendo que el coste de computo (C) es mayor que el de transferencia de datos (M) desde el procesador. Además, se desprecian las latencias iniciales (L_i) y finales(L_f) dado que no se consideran ejecuciones de cargas pequeñas. En este caso particular, se asume que se va a trabajar con un gran volumen de imágenes.

$$T = T_c \cdot (L_i + max(M, C) + L_f) = T_c \cdot C \tag{4.1}$$

Energía. La energía (E) se calcula como la suma de dos componentes: energía dinámica y estática. La primera depende de la actividad del acelerador, es decir, del número de operaciones que realiza cada uno de los elementos *hardware*. A este consumo dependiente de la actividad, se le añade la segunda componente, energía estática, que se produce sólo por estar encendido el acelerador:

$$E = E_d \cdot N_{op} + P_s \cdot T \tag{4.2}$$

Donde E_d representa la energía dinámica media por operación, N_{op} el número de operaciones, P_s la potencia estática y T el tiempo de ejecución.

Para obtener N_{op} se utiliza el instrumentador binario Pin ya que permite ejecutar los binarios nativos sobre Skylake y obtener el número de operaciones. El cálculo de E_d y P_s requiere o bien conocer los datos del *layout* del acelerador, que no están disponibles, o realizar una estimación de los mismos como se ha hecho en este trabajo. Se ha empleado ALADDIN [30] para la obtención del consumo de las unidades funcionales y CACTI [34] para la obtención del consumo de las unidades de memoria. La figura 4.2 ilustra este proceso.

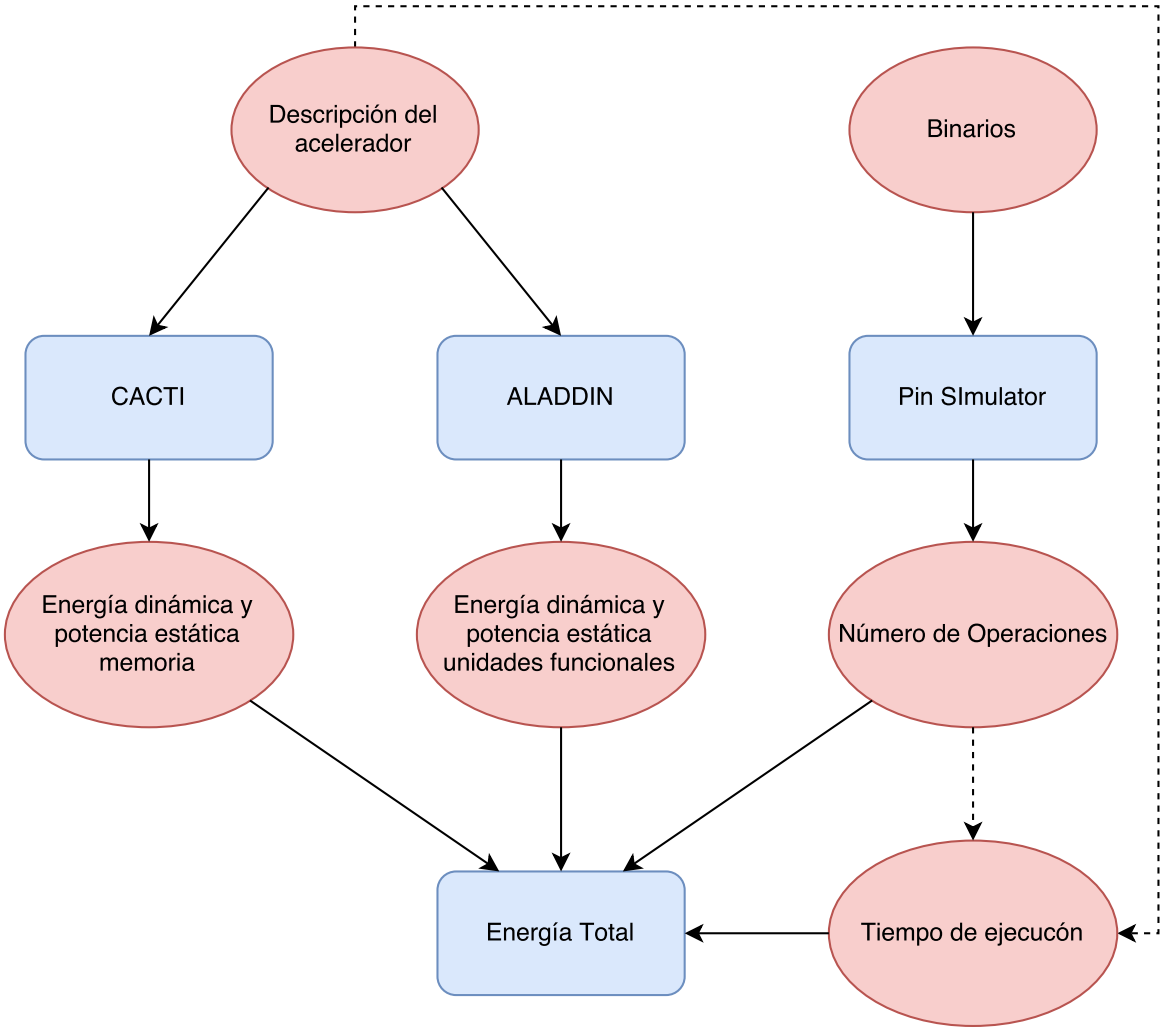


Figura 4.2: Proceso y herramientas para el modelado del consumo energético

Capítulo 5

Integración de ORB en PuDianNao

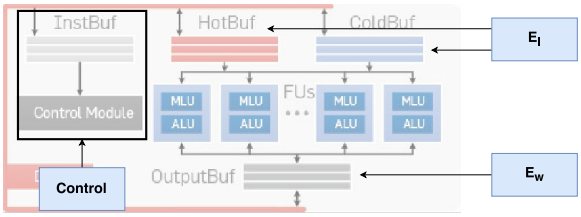
Este capítulo verifica la metodología propuesta en el capítulo 4 integrando ORB en PuDianNao, uno de los aceleradores para *machine learning* publicados más flexible, pero cuya información descrita disponible es limitada. La sección 5.1 muestra el caso de uso al añadir ORB a PuDianNao y la sección 5.2 describe dos mejoras en el diseño del acelerador para aumentar el rendimiento de ORB corriendo sobre PuDianNao. En ambas se presentan los resultados obtenidos en términos de tiempo y energía.

5.1. Integración de ORB en PuDianNao

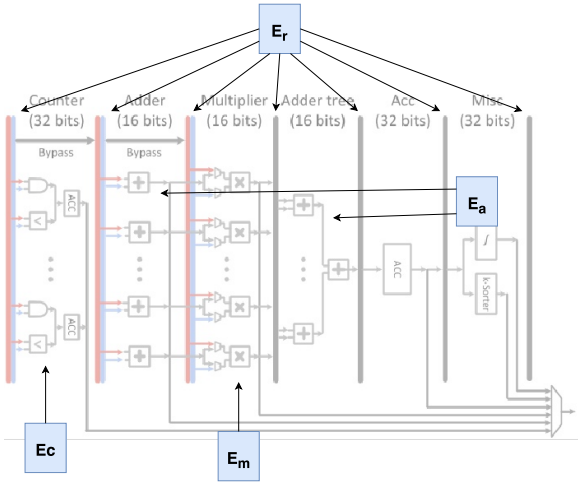
Como ya se ha descrito anteriormente, PuDianNao acelera la ejecución tanto de CNNs como de otros algoritmos de aprendizaje automático. Además, la referencia original de PuDianNao da algo de información sobre su repertorio de instrucciones lo que permite realizar modelos analíticos [3].

Este acelerador presenta dos *buffers* de entrada de datos y uno de *salida*, los cuales son capaces de lidiar con la alta demanda de datos. Además, posee una gran cantidad de unidades funcionales que permiten extraer el paralelismo y asumir la alta intensidad de computo. Por ello, se ha decidido incorporar ORB a este acelerador, el cual se describe con detalle en el anexo D.

Antes de entrar en detalles de implementación, se describen los términos que aparecen en las ecuaciones de las distintas etapas de ORB. Para los cálculos energéticos estos son: la potencia estática (P_s), la energía dinámica del multiplicador (E_m), del sumador (E_a), del comparador (E_c) y de los registros (E_r), y la energía dinámica de lectura (E_l) y escritura (E_w). Además, se utilizan el número de lecturas (N_l), escrituras (N_w), registros (N_r) y operaciones (N_{op}). La figura 5.1 ilustra la correspondencia de estos términos sobre PuDianNao.



(a) Energía relativa a memoria y control



(b) Energía relativa a las unidades funcionales

Figura 5.1: Representación de las métricas energéticas utilizadas sobre la arquitectura de PuDianNao

5.1.1. Integración

Como se ha explicado en el capítulo 2, el algoritmo se divide en seis etapas que se detallan en el anexo A. Se ha programado cada una de ellas siguiendo los detalles de la arquitectura de PuDianNao y de ese código se han extraído los modelos analíticos que se muestran a continuación. Este capítulo realiza un resumen de la integración y no muestra el detalle que aparece en el anexo E, donde está el pseudo-código del bloque de instrucciones de cada etapa para facilitar la comprensión.

Conversión a escala de gris

Este bloque de instrucciones se ejecuta en función del tamaño de la imagen (NxM), el número de ciclos por bloque (CPB), el ancho del acelerador (W), el tiempo de ciclo del procesador (T_c) y el número de colores por pixel (3).

El tiempo de ejecución de esta etapa viene dado por:

$$T = T_c \cdot CPB \cdot \frac{N \cdot M}{W} \tag{5.1}$$

El consumo energético viene dado por:

$$\begin{aligned} E = N_l \cdot N \cdot M \cdot E_l + N_w \cdot N \cdot M \cdot E_w + 3 \cdot N \cdot M \cdot (E_a + E_m) \\ + \frac{T}{T_c} \cdot N_r \cdot E_r + P_s \cdot T \end{aligned} \tag{5.2}$$

Pirámides

Este bloque de instrucciones se ejecuta en función del tamaño de la imagen (NxM), el número de niveles de la pirámide (L), el número de ciclos por bloque (CPB), el ancho del acelerador (W) y el tiempo de ciclo del procesador (T_c).

El tiempo de ejecución de esta etapa viene dado por:

$$T = T_c \cdot CPB \cdot \sum_{i=0}^{L-1} \frac{N \cdot M}{W \cdot 2^i} \tag{5.3}$$

El consumo energético viene dado por:

$$E = \sum_{i=0}^{L-1} \frac{N \cdot M}{2^i} \cdot E_l + \sum_{i=0}^{L-1} \frac{N \cdot M}{2^{i+1}} \cdot E_w + \frac{T}{T_c} \cdot (E_a + N_r \cdot E_r) + P_s \cdot T \tag{5.4}$$

FAST

Este bloque de instrucciones se ejecuta en función del tamaño de la imagen (NxM), el desplazamiento de la ventana (s), el número de niveles de la pirámide (L), el número de ciclos por bloque (CPB), el ancho del acelerador (W) y el tiempo de ciclo del procesador (T_c).

El tiempo de ejecución de esta etapa viene dado por:

$$T = T_c \cdot \sum_{i=0}^{L-1} CPB \cdot \frac{\left\lfloor \frac{N}{2^i \cdot s} \right\rfloor \left\lfloor \frac{M}{2^i \cdot s} \right\rfloor}{W} \tag{5.5}$$

Para el consumo energético también influyen el número de candidatos a punto característicos (C) que encuentra el algoritmo. El consumo viene dado por:

$$\begin{aligned} E = T_c \cdot \sum_{i=0}^{L-1} CPB \cdot (E_a + E_c + E_l) \cdot \left\lfloor \frac{N}{2^i \cdot s} \right\rfloor \left\lfloor \frac{M}{2^i \cdot s} \right\rfloor \\ + C \cdot E_w + \frac{T}{T_c} \cdot N_r \cdot E_r + P_s \cdot T \end{aligned} \tag{5.6}$$

Harris

Este bloque de instrucciones se ejecuta en función de el número de candidatos a punto característico (C), el tamaño de la ventana utilizada para el algoritmo (p^2), el número de ciclos por bloque (CPB), el ancho del acelerador (W) y el tiempo de ciclo del procesador (T_c).

El tiempo de ejecución de esta etapa viene dado por:

$$T = T_c \cdot \frac{CPB \cdot C \cdot p^2 + C}{\left\lfloor \frac{w}{4} \right\rfloor} \tag{5.7}$$

Para el consumo energético también influyen el número de puntos característicos (K) que encuentra el algoritmo. El consumo viene dado por:

$$E = (CPB \cdot C \cdot p^2 + C) \cdot (E_a + E_m) + C \cdot E_l + K \cdot E_w + \frac{T}{T_c} \cdot N_r \cdot E_r + P_s \cdot T \tag{5.8}$$

Orientación

Este bloque de instrucciones se ejecuta en función de el número puntos característicos (K), el tamaño de la ventana utilizada para el algoritmo (p), la distancia de la ventana (d), el número de ciclos por bloque (CPB), el ancho del acelerador (W) y el tiempo de ciclo del procesador (T_c).

El tiempo de ejecución de esta etapa viene dado por:

$$T = T_c \cdot \frac{CPB \cdot p + CPB \cdot d \cdot p}{W} \tag{5.9}$$

El consumo energético viene dado por:

$$E = K \cdot (E_l + E_w) + CPB \cdot p + CPB \cdot d \cdot p \cdot (E_a + E_m) + \frac{T}{T_c} \cdot N_r \cdot E_r + P_s \cdot T \tag{5.10}$$

Descriptores

Este bloque de instrucciones se ejecuta en función de el número puntos característicos (K), el tamaño del descriptor utilizado por el algoritmo (n), el número de ciclos por bloque (CPB), el ancho del acelerador (W) y el tiempo de ciclo del procesador (T_c).

El tiempo de ejecución de esta etapa viene dado por:

$$T = T_c \cdot \frac{2 \cdot CPB \cdot K \cdot n}{W} \tag{5.11}$$

El consumo energético viene dado por:

$$E = (K \cdot n) \cdot (E_w + E_l + 3 \cdot E_a + E_m) + \frac{T}{T_c} \cdot E_r + P_s \cdot T \tag{5.12}$$

5.1.2. Resultados

La ejecución del algoritmo de ORB en PuDianNao supone una aceleración de 309.43× y una reducción del consumo energético de 335.85× como se ilustra en las figuras 5.2 y 5.3.

5.2. Mejoras sobre PuDianNao

Con la finalidad de incrementar las prestaciones de la ejecución del algoritmo de ORB se han propuesto dos mejoras para el acelerador. En ambas se considera como restricción que las prestaciones de las CNNs no pueden verse reducidas, pero solo la primera considera que las prestaciones de los demás algoritmos de *machine learning* soportados por el acelerador tampoco pueden verse reducidas.

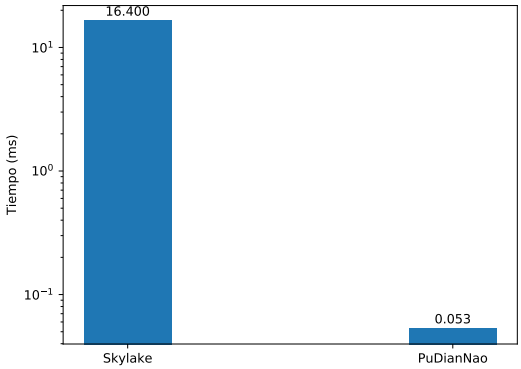


Figura 5.2: Tiempo medio de ejecución al extraer ORB en una imagen en Skylake y en PuDianNao

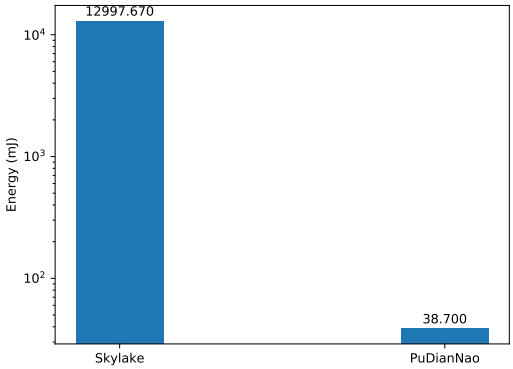


Figura 5.3: Energía media consumida al extraer ORB en Skylake y en PuDianNao

5.2.1. Propuesta 1: Realimentación etapa de sumadores (MLU-2)

Esta propuesta mantiene las prestaciones de las redes neuronales convolucionales y del resto de algoritmos de *machine learning* soportados por el acelerador.

La estructura del acelerador obliga a que cuando se quieren sumar más de dos valores ($output = a + b + c$), se deban realizar realizar la combinación de sumas dos a dos ($mem\_tmp = a + b; output = mem\_tmp + c$) con el consecuente sobrecoste tanto temporal como energético, debido a que se deben escribir y leer en memoria los resultados intermedios.

Esta propuesta soluciona este problema mediante la utilización de los registros de salida de la etapa de suma como entrada de la misma. Para ello se añaden multiplexores a la entrada de los sumadores, a los que se les conectan los registros de entrada y salida de la etapa de suma, como se muestra en la figura 5.4.

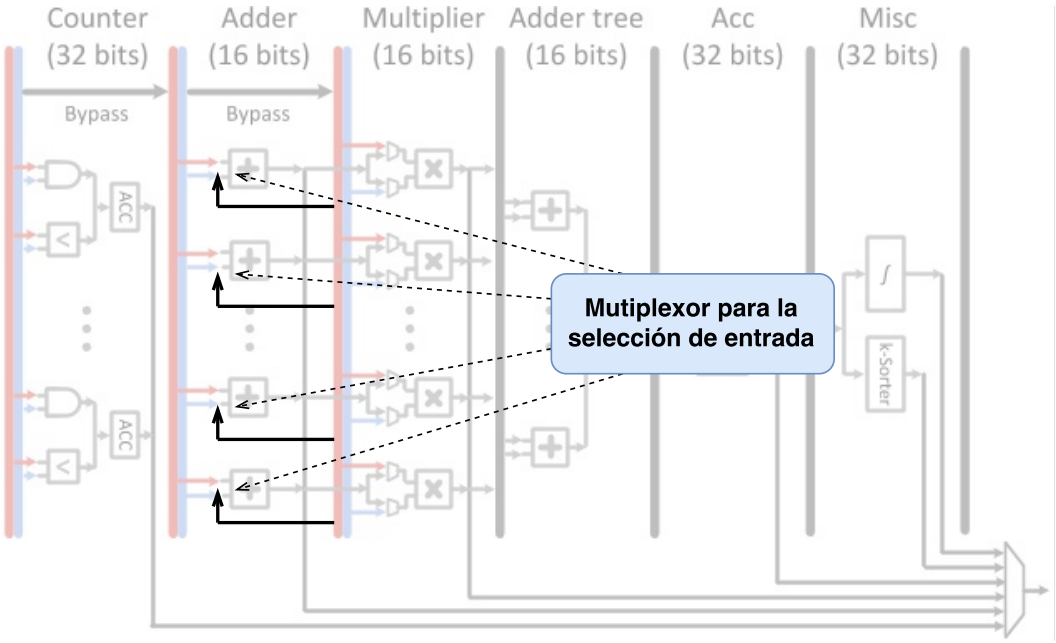


Figura 5.4: Modificación de PuDiaNao mediante la adición de múltiples entradas en la etapa MLU-2 para la aceleración de la conversión de colores

Esta modificación supone una mejora de las prestaciones, ya que disminuyen los accesos memoria y se aprovecha más el *pipeline* del acelerador, disminuyendo los ciclos por bloque (CPB). Esta propuesta requiere la adición de tantos multiplexores como anchura (W) tiene el acelerador, lo cual se considera despreciable en cuanto a aumento del área del acelerador. Esta propuesta consigue una aceleración de 1.18×, y una disminución del consumo energético de 1.06×.

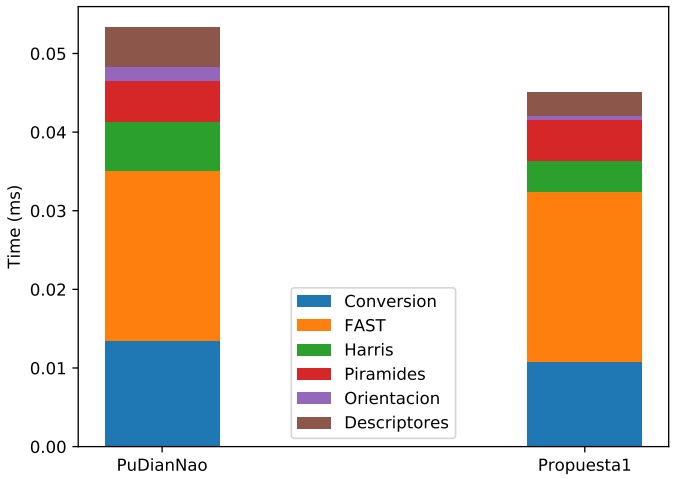


Figura 5.5: Tiempo medio de ejecución al extraer ORB en una imagen en PuDianNao y con la realimentación en la etapa de sumadores

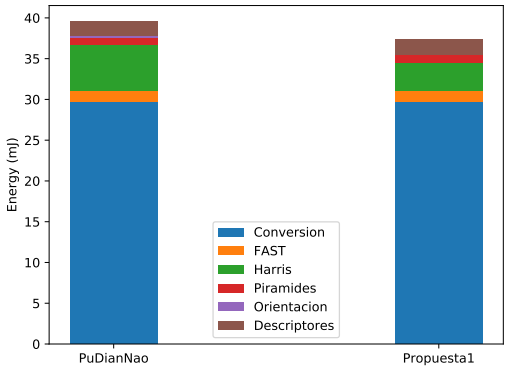


Figura 5.6: Energía media consumida al extraer ORB en una imagen en PuDianNao y con la realimentación en la etapa de sumadores

5.2.2. Propuesta 2: Reordenación de etapas

Esta propuesta mantiene las prestaciones de las CNNs, pero no verifica las consecuencias del nuevo diseño sobre el resto de algoritmos de *machine learning* soportados por el acelerador.

Al igual que en el caso anterior, el orden de las etapas del acelerador no permite el encadenamiento de algunas operaciones, por ejemplo una multiplicación seguida de una suma $(a * b + c)$. Esto obliga a atravesar varias veces todas las etapas del segmentado $(mem\_tmp = a * b; output = mem\_tmp + c)$ conllevando un coste en tiempo no desdeñable y además se añade la sobrecarga en energía por tener que leer y escribir los valores intermedios, mem_tmp.

Empleado como punto de partida la propuesta anterior, esta propuesta permite encadenar operaciones cambiando el orden entre las etapas MLU-1 (comparadores), MLU-2 (sumadores) y MLU-3 (multiplicadores). El resultado consiste en las etapas en orden MLU-3, MLU-2, MLU-1 y puede visualizarse en la figura 5.7.

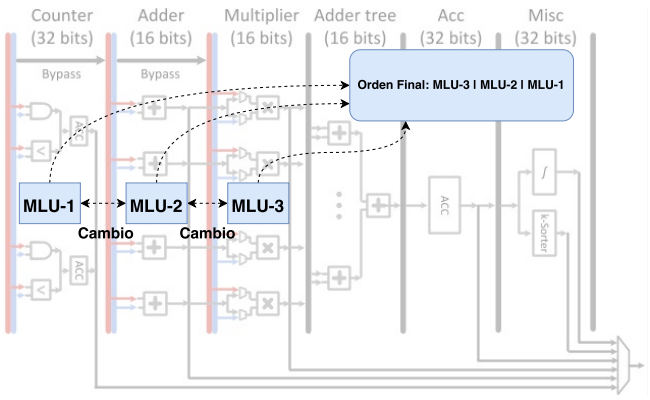


Figura 5.7: Modificación del orden de etapas en el acelerador PuDianNao

Esta modificación supone una mejora en múltiples, ya que disminuyen los accesos memoria y se aprovecha más el *pipeline* del acelerador, disminuyendo los ciclos por bloque (CPB). Consigue una aceleración de 2.11× en cuanto a tiempo, y una disminución del consumo energético de 1.11×.

A la vista de ambas figuras, 5.8 y 5.9, se aprecia que las mejoras reducen el tiempo de ejecución en mucha mayor medida que la energía. Esto se debe a que el número de operaciones, excepto las escrituras a memoria, son las mismas en los tres casos por lo que la energía dinámica es muy similar. Además, al ser la frecuencia del acelerador no muy alta (1 GHz) la potencia estática es baja y la reducción de consumo estático no se aprecia en el total.

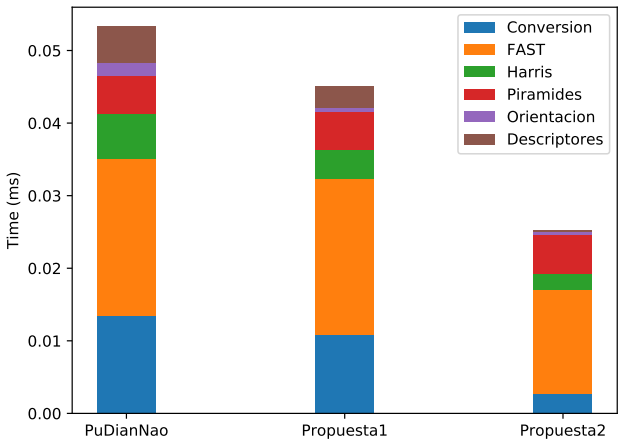


Figura 5.8: Tiempo medio de ejecución al extraer ORB en una imagen en PuDianNao y en las dos propuestas de modificación.

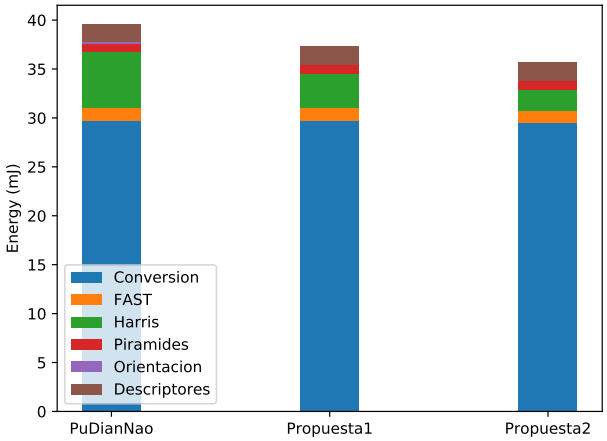


Figura 5.9: Energía media consumida al extraer ORB en una imagen en PuDianNao y en las dos propuestas de modificación.

Capítulo 6

Conclusiones y trabajo futuro

Conclusiones

Como se ha comentado a lo largo de esta memoria, las CNNs se han convertido en la herramienta clave en muchas aplicaciones relacionadas con visión por computador, mejorando el estado del arte en múltiples campos de investigación. Debido a su alto coste computacional y la gran cantidad de datos que manejan, en los últimos años se han propuesto muchos aceleradores hardware para CNNs y en la actualidad las principales empresas tecnológicas como Google, Apple o Microsoft están investigando en ellos.

Este trabajo considera el hecho de que existen otro tipo de algoritmos relevantes en visión por computador, los cuales sería útil poder acelerar en los mismos tipos de *hardware* específico. En particular, se centra en la detección de características locales (un paso clave, por ejemplo, en sistemas de realidad virtual).

Para investigar este problema, a lo largo de este trabajo se han realizado tareas muy distintas: definición y análisis de metodologías, modelos analíticos, adaptación de algoritmos, experimentación, diseño de *hardware* o desarrollo de *software*. Además, se han estudiado y combinado conceptos de dos disciplinas de gran impacto actualmente, tanto en investigación como en la industria: por un lado de visión por computador, para entender bien los algoritmos de extracción de características y redes neuronales, y por otro lado el análisis y diseño de hardware específico, muy importante ante el fin del escalado de la tecnología CMOS y la demanda de altas prestaciones y bajo consumo energético de muchas aplicaciones.

Como conclusiones generales, cabe destacar que se han completado satisfactoriamente todas las tareas propuestas para alcanzar los objetivos planteados:

- Caracterización y análisis de los algoritmos de procesado de imagen con redes neuronales convolucionales y de extracción de características ORB.

- Establecimiento de una metodología para estudiar la viabilidad de la integración

de algoritmos en aceleradores.

- Integración del algoritmo de ORB en el acelerador PuDianNao, consiguiendo una aceleración de 309.43× y una reducción del consumo energético de 335.85×.

- Presentación de dos propuestas de modificación del acelerador PuDianNao para una mayor aceleración del algoritmo de ORB. Una propuesta que obtiene una aceleración de 1.18× y una reducción del consumo energético de 1.06×, y una propuesta que obtiene una aceleración de 2.11× y una reducción del consumo energético de 1.11×.

Los principales problemas encontrados durante la realización de estas tareas han sido:

- La comprensión de los algoritmos de visión, debido a la base matemática que requieren.

- La complejidad de las bibliotecas profesionales utilizadas en los algoritmos estudiados (OpenCV y Caffe). Ha resultado complicado desarrollar los *benchmark* para evaluación empleando dichas bibliotecas, debido a su gran tamaño y numero de dependencias.

- Utilización de los contadores *hardware* de los procesadores para realizar la caracterización de los algoritmos, dada la poca información disponible en los manuales de los procesadores y las limitaciones de las herramientas.

- Desarrollo del código de ejecución de ORB en PuDianNao y su modelo analítico de tiempo de ejecución y consumo energético, dada la limitada información en la publicación de PuDianNao.

Como conclusiones más especificas relativas al problema estudiado, cabe resaltar que como se ve a lo largo de este trabajo, el diseño de *hardware* especifico consigue unas prestaciones muy superiores a las que es capaz de obtener un procesador de propósito general, a cambio de reducir la flexibilidad. Para poder aumentar esta flexibilidad, es fundamental establecer una metodología que permita el estudio de la viabilidad de la integración de algoritmos en aceleradores. Como se ha demostrado en este trabajo, pueden obtenerse mejoras de prestaciones de dos ordenes de magnitud ejecutando aplicaciones en aceleradores *hardware* existentes para otros algoritmos distintos.

Trabajo Futuro

Como extensión directa de este trabajo se podría realizar el análisis de otras tareas relacionadas, como por ejemplo: otros algoritmos de extracción de características, cálculos geométricos del procesado de características locales para buscar correspondencias o diferentes modelos de redes neuronales.

En cuanto a posibles variaciones o propuestas de diferentes metodologías, se podría intentar desarrollar un modelo energético más preciso o comparar con más aproximaciones existentes.

Bibliografía

[1] J. M. Shalf and R. Leland. Computing beyond moore's law. *Computer*, 48(12):14–23, Dec 2015.

[2] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 1–13. IEEE, 2016.

[3] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 369–381. ACM, 2015.

[4] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[5] Chris Harris and Mike Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Citeseer, 1988.

[6] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[7] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. *Computer vision–ECCV 2006*, pages 404–417, 2006.

[8] Edward Rosten, Reid Porter, and Tom Drummond. Faster and better: A machine learning approach to corner detection. *IEEE transactions on pattern analysis and machine intelligence*, 32(1):105–119, 2010.

[9] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2564–2571. IEEE, 2011.

[10] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

[11] Thomas Whelan, Renato F Salas-Moreno, Ben Glocker, Andrew J Davison, and Stefan Leutenegger. Elasticfusion: Real-time dense slam and light source estimation. *The International Journal of Robotics Research*, page 0278364916669237, 2016.

[12] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, and Lawrence D Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems (NIPS)*, pages 396–404, 1990.

[13] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems (NIPS)*, pages 1097–1105. Curran Associates, Inc., 2012.

[15] Feng-Cheng Huang, Shi-Yu Huang, Ji-Wei Ker, and Yung-Chang Chen. High-performance sift hardware accelerator for real-time image feature extraction. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(3):340–351, 2012.

[16] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for

ubiquitous machine-learning. In *ACM Sigplan Notices*, volume 49, pages 269–284. ACM, 2014.

[17] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.

[18] Tao Luo, Shaoli Liu, Ling Li, Yuqing Wang, Shijin Zhang, Tianshi Chen, Zhiwei Xu, Olivier Temam, and Yunji Chen. Dadiannao: A neural network supercomputer. *IEEE Transactions on Computers*, 66(1):73–88, 2017.

[19] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. Sd-vbs: The san diego vision benchmark suite. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 55–64. IEEE, 2009.

[20] Shelby Thomas, Chetan Gohkale, Enrico Tanuwidjaja, Tony Chong, David Lau, Saturnino Garcia, and Michael Bedford Taylor. Cortexsuite: A synthetic brain benchmark suite. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 76–79. IEEE, 2014.

[21] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David M. Brooks. Fathom: Reference workloads for modern deep learning methods. *CoRR*, abs/1608.06581, 2016.

[22] Jeff Donahue. Caffemodel: Caffenet. 2014.

[23] G. Bradski. The opencv library. *Dr. Dobb's Journal of Software Tools*, 2000.

[24] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[25] Linux. Perf: Linux profiling with performance counters. 2009.

[26] Stephane Eranian. Perfmon2. 2001.

[27] ARM Holdings. Cortex-a15 technical reference manual. 2012.

[28] Intel Corporation. Intel 64 and ia-32 architectures optimization reference manual. 2015.

[29] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 35–44. IEEE, 2014.

[30] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 97–108. IEEE, 2014.

[31] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[32] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. Co-designing accelerators and soc interfaces using gem5-aladdin. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[33] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[34] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20, HP Labs, April 2008.

[35] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 806–813, 2014.

[36] Evan Shelhamer. Caffemodel: Alexnet. 2014.

[37] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. *Computer vision–ECCV 2006*, pages 430–443, 2006.

[38] Paul L Rosin. Measuring corner properties. *Computer Vision and Image Understanding*, 73(2):291–307, 1999.

[39] Vijay Janapa Reddi, Alex Settle, Daniel A Connors, and Robert S Cohn. Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, page 22. ACM, 2004.

Lista de Figuras

Lista de Tablas

Anexos

Anexo A

Algoritmo ORB

Conversión a escala de grises

Habitualmente las imágenes son capturadas por las cámaras en formato RGB, no obstante ORB requiere que las imágenes se encuentren en formato de escala de grises. Para realizar la conversión, se realiza la suma de cada uno de los colores (R, G, B) multiplicados por sus respectivas constantes (K_r, K_g, K_b).

$$Y = R \cdot K_r + G \cdot K_g + B \cdot K_b \tag{A.1}$$

Creación de las pirámides

El algoritmo utilizado en la detección es FAST, el cual no produce características *multi-scale*. Por ello es necesario realizar un escalado piramidal de la imagen y aplicar FAST en cada uno de los niveles de la pirámide.

El escalado piramidal consiste en reducir la imagen a una cuarta parte de forma consecutivamente durante N veces, siendo N el número de niveles de la pirámide. La figura A.1 ilustra este procedimiento.

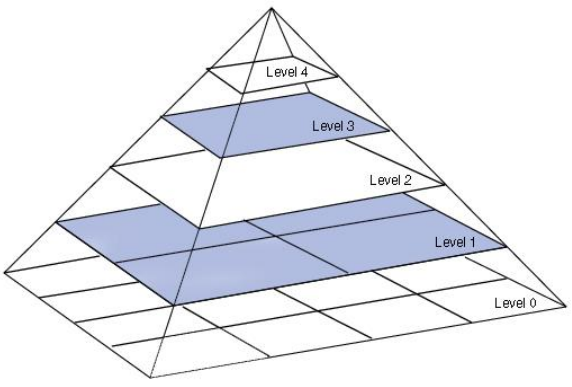


Figura A.1: Escalado piramidal de cinco nivéles[1]

Uso del detector FAST

Para la detección de los candidatos a puntos característicos se utiliza el detector FAST, puesto que es el único que permite la ejecución en tiempo real. Este detector verifica si un píxel es un punto oscuro (d), claro (b) o similar (s), comparandoló con un número N de píxeles a su alrededor, como se ilustra en la figura A.2.

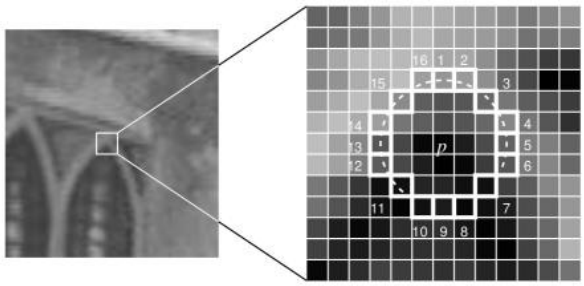


Figura A.2: Detector FAST

La ecuación A.2 muestra como se decide a que conjunto pertenece un píxel. Para cada uno de los píxeles vecinos ($I_{p\to x}$), se compara si el píxel actual (I_p) más o menos un *threshold* (t) es mayor o menor. Si el píxel se clasifica en el mismo conjunto para M vecinos, entonces se considera como un candidato a punto característico.

$$S_{p\to x} = \begin{cases} d, & I_{p\to x} \leq I_p - t \\ s, & I_{p\to x} - t < I_p - t < I_{p\to x} + t \\ b, & I_p + t \leq I_{p\to x} - t. \end{cases} \tag{A.2}$$

Refinamiento mediante Harris

Dado que FAST no es capaz de obtener una métrica para los puntos característicos obtenidos, se utiliza el detector de esquinas de Harris para obtener este valor. Este algoritmo consiste en el cálculo de los gradientes en los puntos candidatos obtenidos por FAST, utilizando una ventana de desplazamiento que permite calcula la variación de la intensidad.

Denotando las intensidades pixélicas como I, la ecuación A.3 muestra el cambio producido E tras un desplazamiento en x e y:

$$E_{u,v} = \sum_{x,y} w_{x,y}[I_{x+u,y+v} - I_{x,y}]^2 \simeq \sum_{x,y} u^2 I_x^2 + 2uvI_xI_y + v^2 I_y^2 \tag{A.3}$$

Utilizando una notación matricial se puede expresar como:

$$E_{u,v} \simeq \begin{bmatrix} u & v \end{bmatrix} \sum_{x,y} w_{x,y} \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \tag{A.4}$$

[1]Fuente: `http://cs.brown.edu/courses/csci1430/2011/results/proj1/georgem/`

De la ecuación anterior se puede obtener M, la matriz simétrica 2x2.

$$M = \sum_{x,y} w_{x,y} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \tag{A.5}$$

Finalmente, se puede calcular la métrica (*score*) para el candidato a punto característico.

$$R = det(M) - k \cdot trace(M))^2 \tag{A.6}$$

Obtención de la orientación

Para la obtención de la orientación, se utiliza la intensidad de los centroides[38]. Esta técnica asume que la intensidad del punto es un *offset* de su centro, cuyo vector puede utilizarse para calcular la orientación.

Los momentos de un *patch* se definen como:

$$m_{p,q} = \sum_{x,y} x^p y^p I_{x,y} \tag{A.7}$$

Con esos momentos se encuentra el centroide:

$$C = \begin{pmatrix} \frac{m_{10}}{m_{00}} & \frac{m_{01}}{m_{00}} \end{pmatrix} \tag{A.8}$$

Finalmente, con el vector del centro del punto al centroide se calcula la orientación:

$$\theta = atan2(m_{01}, m_{10}) \tag{A.9}$$

Generación de los descriptores

ORB utiliza el descriptor BRIEF, el cual consiste en una cadena de bits construida por la realización de un *test* binario sobre diferentes puntos. El *test* binario τ, donde $p(x)$ es la intensidad de un *patch* en el punto x, viene definido por:

$$\tau(p; x, y) := \begin{cases} 1, & p(x) < p(y) \\ 0, & p(x) \geq p(y) \end{cases} \tag{A.10}$$

La *feature* queda descrita como un vector de n *tests* binarios:

$$f_n(p) := \sum_{1 \leq i \leq n} 2^{i-1} \tau(p; x_i, y_i) \tag{A.11}$$

ORB genera los descriptores de una forma más eficiente, para cada conjunto de *features* de n *tests* binarios localizados en (x_i, y_i) define una matriz 2 x n:

$$S = \begin{pmatrix} x_1 & , .., & x_n \\ y_1 & , .., & y_n \end{pmatrix} \tag{A.12}$$

Usando la orientación θ del *patch* y su correspondiente matriz de rotación R_θ, se construye una versión *steered*(dirigida) S_θ de S:

$$S_\theta = R_\theta \cdot S \tag{A.13}$$

Ahora, el operador de *steered* (dirigido) BRIEF pasa a ser:

$$g_n(p, \theta) := f_n(p) | (x_i, y_i) \in S_\theta \tag{A.14}$$

Anexo B

Redes neuronales convolucionales

Este anexo presenta un breve resumen de los tipos de capas utilizadas en las CNNs consideradas en este trabajo.

Convolutional Layer

Es la capa principal de las CNNs, y como su nombre indica está basada en la convolución. La ecuación B.1 muestra la definición matemática de la convolución de dos funciones (f y g), la cual se denota con el símbolo $*$ y consiste en la integral del producto de ambas funciones después de desplazar una de ellas una distancia (τ).

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau \tag{B.1}$$

La convolución consiste en la aplicación de diferentes filtros (*kernels*) a las entradas de la capa. Estos filtros de aplican con un *stride*(S) y un *padding*(P). En la figura B.1 se muestra la convolución de una imagen de forma más gráfica.

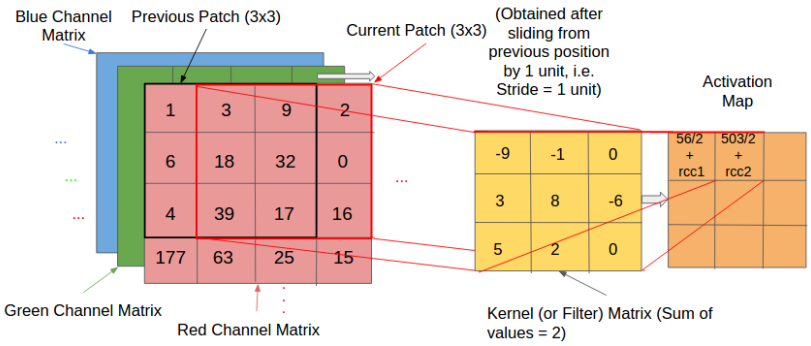


Figura B.1: Convolución de una imagen con tres canales (RGB). A la izquierda se muestran los tres canales, resaltando dos *patches* (*previous* y *current*). En amarillo se muestra el *kernel* utilizado, y en naranja el resultado parcial después de aplicarlo sobre los dos *patches*.[1]

[1]Fuente imagen: http://xrds.acm.org/blog/2016/06/convolutional-neural-networks-cnns-illustrated-explanation/

Pooling Layer

La capa de *pooling* tiene como finalidad de reducir progresivamente el tamaño de los elementos de la red y prevenir el sobre-entrenamiento (*overfitting*). Se introducen periódicamente entre varias capas convolucionales y se utiliza la operación MAX para la reducción. Normalmente, se realiza con filtros (*kernels*) de tamaño 2x2 aplicados con un desplazamiento (*stride*) de 2, reduciendo el tamaño en un 75 %.

Normalization Layer

Esta capa tiene la finalidad de mantener los valores de las neuronas dentro de un rango determinado. Se considera que las aportaciones a los resultados de la red son mínimos.

Full-Connected Layer

La capa *Fully-Connected* es la última capa de la red y tiene conexiones a todas las activaciones en la capa anterior. Las activaciones se suelen calcular con una multiplicación de matrices seguida de la suma del *bias*.

Anexo C

A Top-Down Method for Performance Analysis and Counters Architecture

Analizar el comportamiento de una aplicación en un procesador de propósito general es una tarea muy difícil y costosa. El incremento de la complejidad de la microarquitecutra, la diversidad de las cargas y la gran información necesaria, han hecho que esta tarea se vuelva casi imposible.

Esta metodología pretende hacer posible la caracterización de aplicaciones y detección de cuellos de botella en procesadores fuera de orden. Esta metodología propone una análisis de arriba a abajo que usando los contadores *hardware*, obtiene una serie de métricas de forma jerárquica que permiten profundizar hasta las causas de la limitación de la aplicación.

El *pipeline* de las CPU fuera de orden tiene dos partes principales: el *frontend* y el *backend*. El *frontend* es responsable de cargar las instrucciones desde memoria y pasarlas al *backend*. El *backend* se encarga de planificar, ejecutar y retirar las instrucciones que ha recibido. En la figura C.1 se muestra la microarquitectira del procesador Intel Ivy Bridge señalando *frontend* y *backend*.

C.1. Descripción de las métricas

La metodología utiliza un modelo de arriba a abajo definiendo una jerarquía que permite profundizar en los cuellos de botella de la aplicación. En la figura C.2 se visualiza la jerarquía definida por la metodología.

C.1.1. Top Level Breakdown

Es necesaria realizar una primera clasificación de la actividad del *pipeline*. Se definen cuatro categorías: Frontend Bound, Backend Bound, Bad Speculation, Frontend Bound.

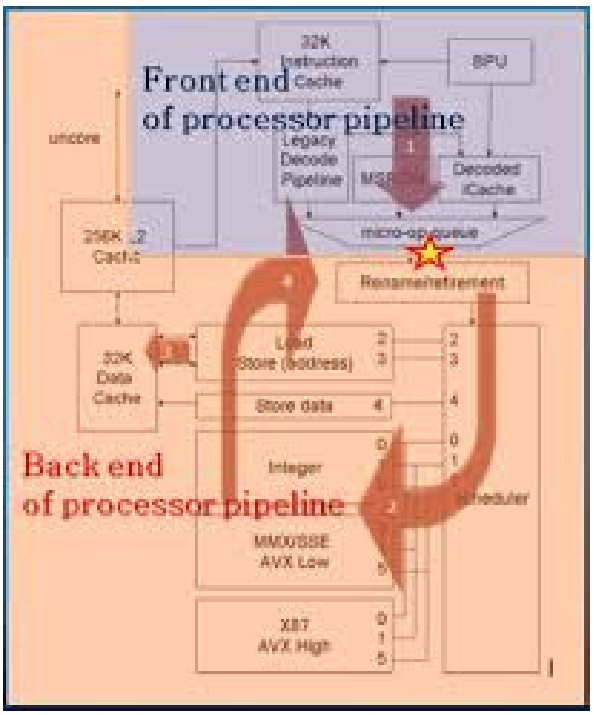


Figura C.1: Diagrama microarquitectónico del procesador Intel Ivy Bridge. En azul se muestra el *frontend* del procesador y rojo el *backend*.

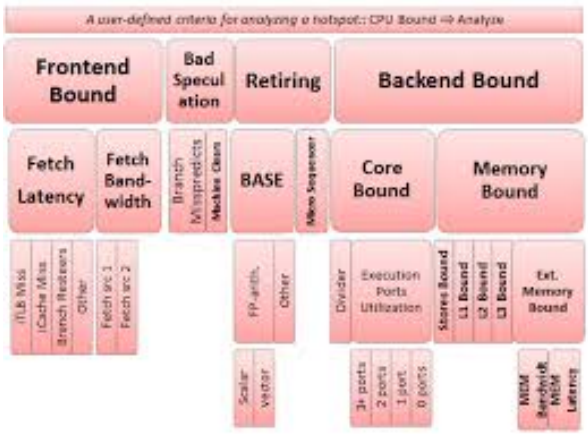


Figura C.2: Jerarquía definida por la metodología de análisis. De arriba a abajo se visualizan los niveles de menor a mayor especificidad.

El flujo para la clasificación se visualiza en la figura C.3.

C.1.2. Frontend Bound Category

El *frontend* hace referencia a la parte del *pipeline* donde el predictor de saltos predice la siguiente instrucción, la cual se carga desde la cache de instrucciones y es pasada al *backend*. *Frontend Bound* representa cuando el *frontend* no alimenta con suficientes instrucciones al *backend*.

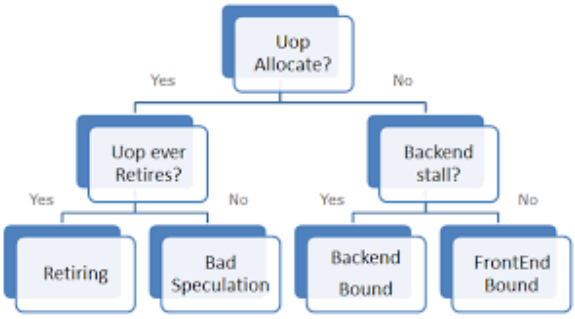


Figura C.3: Flujo para la clasificación de una instrucción en el primer nivel

La métrica se define como:

$$FrontendBound = \frac{InstruccionesNoRepartidas}{Ciclos \cdot AnchoInstrucciones} \tag{C.1}$$

C.1.3. Bad speculation Category

Bad Speculation refleja la actividad del *pipeline* malgastada debido a especulaciones incorrectas. Esto incluye los ciclos en los que se lanzan instrucciones que nunca serán retiradas y los ciclos que en los que el *pipeline* esta bloqueado debido a la recuperación necesaria por una especulación fallida.

La métrica se define como:

$$BadSpeculation = \frac{InsLanzadas - InsRetiradas + CiclosRecuperacion}{Ciclos \cdot AnchoIns} \tag{C.2}$$

C.1.4. Retiring Category

Retiring refleja las instrucciones que han sido retiradas respecto a las que podrían haberlo sido en ese número de ciclos. Cuanto mayor es el valor de *Retiring*, mayor es el aprovechamiento de la CPU, ya que se están ejecutando más instrucciones por ciclo.

La métrica se define como:

$$Retiring = \frac{InstruccionesRetiradas}{Ciclos \cdot AnchoInstrucciones} \tag{C.3}$$

C.1.5. Backend Bound Category

Backend Bound refleja las instrucciones que no pueden continuar su ejecución en el *backend* debido a falta de recursos y deben esperar para poder avanzar.

La métrica se define como:

$$BackendBound = 1 - (FrontendBound + BadSpeculation + Retiring) \tag{C.4}$$

Esta métrica se divide en otras dos métricas: *Memory Bound* y *Core Bound*. La primera, hace referencia a los ciclos de *stall* debidos al subsistema de memoria, mientras

que la segunda, indica los ciclos de *stall* debidos a la ocupación de las unidades funcionales.

La métrica Memory Bound se define como:

$$MemoryBound = \frac{StallsMemoria}{Ciclos} \tag{C.5}$$

La métrica Core Bound se define como:

$$BackendBound = \frac{StallsEjecucion}{Ciclos} \tag{C.6}$$

C.1.6. Memory Bound Breakdown

La métrica *Memory Bound* que acaba de ser descrita puede descomponerse en múltiples niveles, tantos como la jerarquía de memoria tenga. La metodología esta definida para el caso habitual de Intel, en el que hay cuatro niveles de memoria (L1, L2, L3 y memoria principal). Además de las lecturas en memoria, se contempla la porción de la métrica que ocupan las escrituras en memoria (*Store Bound*).

La métrica L1 Bound se define como:

$$L1Bound = \frac{StallsMemoria - StallsFallosL1}{Ciclos} \tag{C.7}$$

La métrica L2 Bound se define como:

$$L2Bound = \frac{StallsFallosL1 - StallsFallosL2}{Ciclos} \tag{C.8}$$

La métrica L3 Bound se define como:

$$L3Bound = \frac{StallsFallosL2 - StallsFallosL3}{Ciclos} \tag{C.9}$$

La métrica Mem Bound se define como:

$$MemBound = \frac{StallsFallosL3}{Ciclos} \tag{C.10}$$

La métrica Store Bound se define como:

$$StoreBound = \frac{StallsEscrituras}{Ciclos} \tag{C.11}$$

C.2. Adaptación para un procesador ARM A-15

La metodología y métricas que acaban de ser descritas fueron propuestas para procesadores Intel. Sin embargo, por las características de este proyecto ha sido también necesario realizar la caracterización en un procesador ARM A-15. Dado que este procesador no dispone de los mismos contadores *hardware*, ha sido necesaria la adaptación de la metodología para este caso.

C.2.1. Top Level

En el primer nivel no ha sido posible obtener todas las métricas debido a que el procesador no tiene los contadores *hardware* necesarios. Se ha obtenido la métrica *Retiring*, y las otras tres (*Frontend Bound*, *Backend bound*, *Memory bound*) se han agrupado como *Others*.

La métrica *Retiring* se define como:

$$Retiring = \frac{InstruccionesRetiradas}{Ciclos \cdot AnchoInstrucciones} \tag{C.12}$$

La métrica *Others* se define como:

$$Others = 1 - Retiring \tag{C.13}$$

C.2.2. Backend Level

En este nivel ha sido imposible la obtención de las métricas *Core Bound* y *Memory Bound*, debido a que este procesador no posee ningún contador de *stalls*.

C.2.3. Memory Level

Lo deseado enn este nivel sería que el procesador tuviera contadores de *stall*, pero como ya se ha dicho no es el caso. Sin embargo, se ha diseñado un procedimiento alternativo que permite obtener todas las métricas menos *Store Bound*, es decir: *L1 Bound*, *L2 Bound* y *Mem Bound* (este procesador no tiene nivel L3 de memoria).

La métrica *L1 Bound* se define como:

$$L1Bound = \frac{Instrucciones}{AnchoInstrucciones \cdot Ciclos} \cdot \frac{LatenciaL1 \cdot AccesosL1}{Ciclos} \tag{C.14}$$

La métrica *L2 Bound* se define como:

$$L2Bound = \frac{Instrucciones}{AnchoInstrucciones \cdot Ciclos} \cdot \frac{LatenciaL2 \cdot AccesosL2}{Ciclos} \tag{C.15}$$

La métrica *Mem Bound* se define como:

$$MemBound = \frac{Instrucciones}{AnchoInstrucciones \cdot Ciclos} \cdot \frac{LatenciaMem \cdot AccesosMem}{Ciclos} \tag{C.16}$$

Anexo D

PuDianNao: A Polyvalent Machine Learning Accelerator

PuDianNao es un acelerador *hardware* perteneciente a la familia DianNao que acelera la ejecución de múltiples algoritmos de *machine learning*. Como se ilustra en la figura D.1, PuDianNao consiste en varias unidades funcionales (FUs), tres *buffers* de datos (HotBuf, ColdBuf y OutputBuf), un *buffer* de instrucciones (Instbuf), un modulo de control y un DMA.

D.1. Unidades funcionales

Las unidades funcionales (FUs) son las unidades de ejecución básicas de PuDianNao. Concretamente, cada FU consiste en dos partes, una unidad de *machine learning* (MLU) y una unidad aritmético lógica (ALU).

D.1.1. Unidad de *machine learnign* (MLU)

La MLU esta diseñada para soportar múltiples e importantes operaciones básicas comunes en las técnicas de *machine learning*. Como se ilustra en la figura D.2, el *pipeline* de la MLU está dividido en seis etapas (Counter, Adder, Multiplier, Adder tree, Acc y Misc).

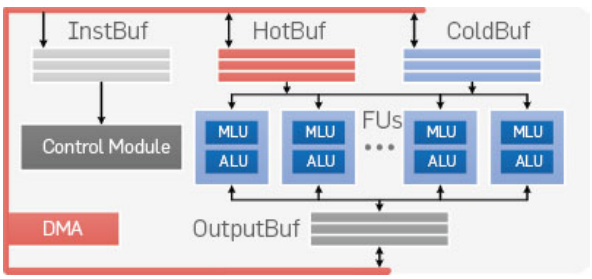


Figura D.1: Arquitectura del acelerador PuDianNao

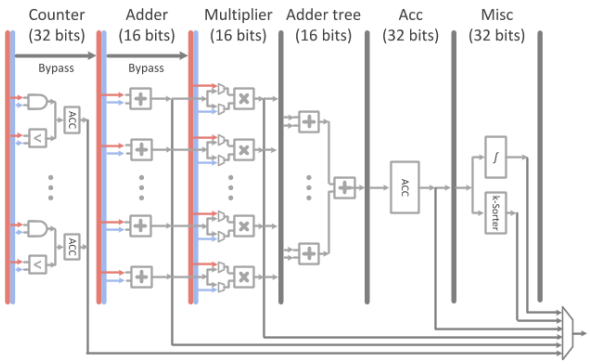


Figura D.2: *Pipeline de la MLU del acelerador PuDianNao*

En la etapa **Counter**, cada par de entradas alimentan una puerta AND o son comparadas por una unidad de comparación, añadiendo el valor obtenido a un acumulador. Después, los resultados de los acumuladores son enviados directamente al *output buffer* en lugar de la siguiente etapa. Además, se puede realizar un *bypass* de esta etapa.

En la etapa **Adder**, cada par de entradas alimentan a un sumador. Los resultados pueden ser enviados directamente al *output buffer* o a la siguiente etapa. Además, se puede realizar un *bypass* de esta etapa.

En la etapa **Multiplier** cada par de entradas alimenta a un multiplicador. Las entradas pueden ser salidas de la etapa anterior o directamente datos de los *Input Buffers*. Los resultados pueden ser enviados directamente al *output buffer* o a la siguiente etapa.

La etapa **Adder tree** realiza la adición del resultado de todos los multiplicadores. Cuando las dimensiones que desean sumarse son mayores al tamaño del *adder tree*, se acumulan los resultados parciales en la etapa **Acc**. Después, los resultados pueden ser enviados directamente al *output buffer* o a la siguiente etapa.

La etapa **Misc** integra dos módulos, interpolación linear y *k-sorter*. Esta etapa no es utilizada en este trabajo por lo que no se dan más detalles de la misma.

D.1.2. Unidad aritmético lógica (ALU)

Además de las operaciones básicas que acaban de ser descritas, en ocasiones se requieren otras operaciones no soportadas por la MLU (e.g., división, asignación condicional). Aunque es infrecuente, ejecutar estas operaciones en la CPU principal requeriría un gran movimiento de datos y sobrecostes por la sincronización. Por ello, cada FU contiene una pequeña ALU que contiene un sumador, un multiplicador, un divisor y una conversor de 16 a 32 bits y viceversa.

D.2. Buffers de datos

Para el aprovechamiento de la localidad de muchos algoritmos de *machine learning*, el acelerador cuenta con tres *buffers on-chip* separados: HotBuf (8KB), ColdBuf (16KB) y OutputBuf (8KB). El HotBuf almacena los datos de entrada que tienen un reuso corto, el ColdBuf almacena los datos de entrada que tienen un reuso largo y el OutputBuf almacena los datos de salida y los resultados temporales.

Los tres *buffers* están conectados al mismo DMA. Además, se utilizan SRAMs de un puerto para el HotBuf y ColdBuf, que reducen el área y el consumo energético. Debido a que las FUs pueden leer o escribir en el OutputBuf, se necesita una SRAM de doble puerto.

Anexo E

Bloques de instrucciones de ORB en PuDianNao

Conversión de RGB a escala de grises

1. Carga y lectura de píxeles del canal Red en el Cold Buffer
2. Utilización de la etapa MLU-3 para multiplicarlos por la constante K_r
3. Escritura del resultado en el Ouput Buffer (G_1)
4. Carga y lectura de píxeles del canal Blue en el Cold Buffer
5. Utilización de la etapa MLU-3 para multiplicarlos por la constante K_b
6. Escritura del resultado en el Ouput Buffer (G_2)
7. Carga y lectura de píxeles del canal Green en el Cold Buffer
8. Utilización de la etapa MLU-3 para multiplicarlos por la constante K_g
9. Escritura del resultado en el Ouput Buffer (G_3)
10. Carga y lectura de G_1 en Hot Buffer y de G_2 en Cold Buffer
11. Utilización de la etapa MLU-2 para la adición de G_1 y G_2, produciendo G_{12}
12. Escritura del resultado en el Ouput Buffer
13. Carga y lectura de G_12 en Hot Buffer y de G_3 en Cold Buffer
14. Utilización de la etapa MLU-2 para la adición de G_{12} y G_3, produciendo los píxeles finales

Creación de las pirámides

1. Carga y lectura de píxeles en el Cold Buffer
2. Utilización de MLU-4 para realizar la suma de todos los píxeles
3. Utilización de la división para la obtención de la media
4. Escritura del resultado en el Ouput Buffer

Uso del detector FAST

1. Carga y lectura de píxeles en el Cold Buffer
2. Carga y lectura de threshold, máscaras, y candidatos en Hot Buffer
3. Utilización de la etapa MLU-2 para sumar los píxeles y el threshold
4. Escritura y lectura de los resultados
5. Utilización de la etapa MLU-1 para comparar los resultados con los candidatos
6. Escritura y lectura de los resultados
7. Utilización de la etapa MLU-1 para comparar los resultados con las máscaras
8. Utilización de la etapa MLU-4 para sumar el resultado y acumularlo
9. Escritura del resultado en el Ouput Buffer
10. Carga y lectura de píxeles en el Cold Buffer
11. Carga y lectura de threshold, máscaras, y candidatos en Hot Buffer
12. Utilización de la etapa MLU-2 para restar los píxeles y el threshold
13. Escritura y lectura de los resultados
14. Utilización de la etapa MLU-1 para comparar los resultados con los candidatos
15. Escritura y lectura de los resultados
16. Utilización de la etapa MLU-1 para comparar los resultados con las máscaras
17. Utilización de la etapa MLU-4 para sumar el resultado y acumularlo
18. Escritura del resultado en el Ouput Buffer

Refinamiento mediante Harris

1. Carga y lectura de píxeles en el Cold Buffer
2. Utilización de la etapa MLU-2 para sumar los píxeles el eje x
3. Utilización de la etapa MLU-2 para sumar los píxeles el eje y
4. Utilización de la etapa MLU-3 para multiplicar eje x
5. Utilización de la etapa MLU-4 para acumular eje x
6. Utilización de la etapa MLU-3 para multiplicar eje y
7. Utilización de la etapa MLU-4 para acumular eje y
8. Utilización de la etapa MLU-3 para multiplicar eje x e y
9. Utilización de la etapa MLU-4 para acumular eje x e y
10. Utilización de las etapas MLU-2 y MLU-3 para obtener el score

Calculo de la orientación

1. Carga y lectura de píxeles en el Cold Buffer
2. Uso de MLU-2 para realizar la suma de píxeles
3. Uso de MLU-3 para multiplicar los resultados
4. Uso de MLU-4 para acumular los resultados anteriores
5. Uso de MLU-2 para realizar la suma de píxeles
6. Uso de MLU-3 para multiplicar los resultados
7. Uso de MLU-4 para acumular los resultados anteriores
8. Escritura de la orientación en Output Buffer

Generación de los descriptores

1. Carga y lectura del *pattern* en el Cold Buffer
2. Utilización de MLU-3 para multiplicar por el ángulo
3. Escritura y lectura de los resultados

4. Utilización de MLU-2 para la suma de los valores anteriores
5. Escritura y lectura de los resultados
6. Utilización de MLU-3 para multiplicion parcial
7. Escritura y lectura de los resultados
8. Utilización de MLU-2 para la suma de los valores y generacion de la dirección
9. Escritura del resultado en el Output Buffer
10. Carga y lectura de píxeles en el Hot Buffer
11. Utilización MLU-3 para multiplicar por el desplazamiento.
12. Utilización MLU-1 para comparar los dos *pixels*.
13. Escritura del resultado en el Output Buffer

Anexo F

Distribución Temporal de las Tareas

El desarrollo de este trabajo se ha llevado a cabo durante ocho meses, controlando el tiempo empleado en cada una de las tareas. En la figura F.1 se muestra un Diagrama de Gantt que representa el esfuerzo dedicado desglosado por semanas y tareas.

Este trabajo se puede organizar principalmente en cuatro bloques de tareas: (1) Estudio del estado del arte de los algoritmos de visión por computador y los aceleradores *hardware* existentes para este propósito; (2) Caracterización de los algoritmos para determinar cuales son los componentes a analizar; (3) Propuesta de una metodología para estudiar la viabilidad de integrar un algoritmo en un acelerador; (4) Integración de ORB en el acelerador PuDianNao; (5) Documentación del proyecto.

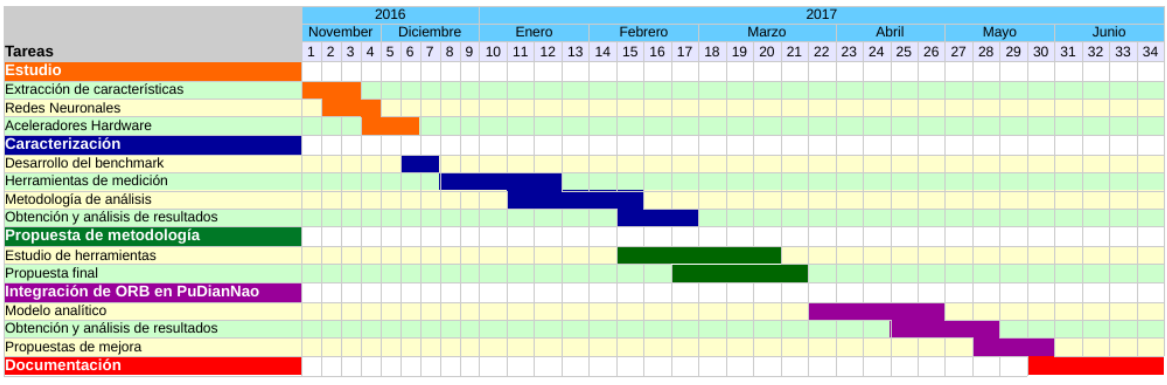


Figura F.1: Diagrama de Gantt del proyecto. Muestra las tareas y subtareas desarrolladas distribuidas en años, meses y semanas.

Para completar la planificación del proyecto, se muestra en la figura F.2 el esfuerzo dedicado a cada una de las tareas.

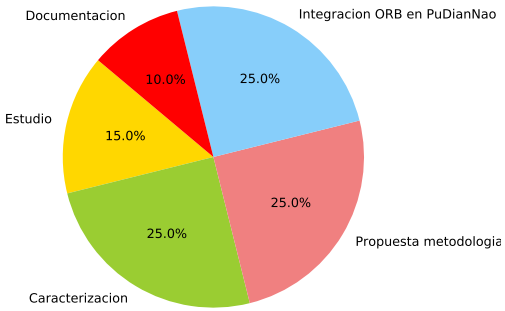


Figura F.2: Esfuerzo dedicado a cada una de las tareas