TRABAJO FIN DE GRADO

UCAM

UNIVERSIDAD CATÓLICA
DE MURCIA

ESCUELA UNIVERSITARIA POLITÉCNICA

Departamento de Ciencias Politécnicas
Grado en Ingeniería Informática

Implementación y evaluación del modelo DLRM en acelerador ASIC simulado mediante STONNE

Autor:
D. Nicolás Meseguer Iborra

Director:
Dr. D. José Luis Abellán Miguel

Murcia, Junio de 2021

TRABAJO FIN DE GRADO

UCAM

UNIVERSIDAD CATÓLICA
DE MURCIA

ESCUELA UNIVERSITARIA POLITÉCNICA

Departamento de Ciencias Politécnicas
Grado en Ingeniería Informática

Implementación y evaluación del modelo DLRM en acelerador ASIC simulado mediante STONNE

Autor:
D. Nicolás Meseguer Iborra

Director:
Dr. D. José Luis Abellán Miguel

Murcia, Junio de 2021

Dr. D. José Luis Abellán Miguel profesor de la UCAM.

CERTIFICA: que el Trabajo Fin de Grado titulado "Implementación y evaluación del modelo DLRM en acelerador ASIC simulado mediante STONNE" que presenta D. Nicolás Meseguer Iborra, para optar al título oficial de Grado en Ingeniería informática, ha sido realizado bajo su dirección.

A su juicio reúne las condiciones necesarias para ser presentado en la Universidad Católica San Antonio de Murcia y ser juzgado por el tribunal correspondiente.

Murcia, a 31 de Mayo de 2021.

Agradecer en primer lugar a mi madre, por estar siempre apoyándome a hacer lo que más me gusta, a mi familia por ayudarme durante estos años de carrera y finalmente, al Dr. D. José Luis Abellán Miguel y a nuestros colaboradores, el Dr. D. Manuel E. Acacio Sánchez y D. Francisco Muñoz Martínez, que me han acompañado durante el desarrollo del presente trabajo.

Índice

Abstract xxv

Resumen xxvii

1. Introducción 29

1.1. Motivación 30

1.2. Definición 31

1.3. Objetivos propuestos 32

1.3.1. Objetivos Generales 32

1.3.2. Objetivos Específicos 32

2. Estado del arte 35

2.1. Conceptos relevantes del dominio de aplicación 35

2.2. Relación con proyectos con la misma funcionalidad 47

2.3. Estudio de viabilidad 49

2.3.1. Alcance del proyecto 49

2.3.2. Estudio de la situación actual 50

2.3.3. Estudio y valoración de las alternativas de solución 50

2.3.4. Selección de la solución 52

3. Metodología de desarrollo del software 55

3.1. Scrum 57

3.1.1. Implementación de Scrum 58

3.1.2. Roles en el equipo Scrum 58

3.1.3. Eventos de cada sprint 58

3.1.4. Herramientas de Scrum 59

4. Tecnologías y herramientas utilizadas 61

4.1. Infraestructura software 61

4.2. Infraestructura hardware 63

5. Estimación de recursos y planificación **65**
5.1. Preparación preliminar 65
5.2. Estimación por puntos de historia 68
5.3. Planificación temporal del proyecto 71
5.4. Valoración de la dedicación y coste económico 73

6. Desarrollo del contenido del proyecto **77**
6.1. Sprint 1 77
6.1.1. Sprint planning 77
6.1.2. Sprint review 79
6.1.3. Sprint Retrospective 82
6.2. Sprint 2 84
6.2.1. Sprint planning 85
6.2.2. Sprint review 87
6.2.3. Sprint retrospective 91
6.3. Sprint 3 94
6.3.1. Sprint planning 94
6.3.2. Sprint review 96
6.3.3. Sprint retrospective 102
6.4. Sprint 4 104
6.4.1. Sprint planning 105
6.4.2. Sprint review 106
6.4.3. Sprint retrospective 125

7. Despliegue y prueba de la implementación **129**
7.1. Plan de pruebas 129
7.2. Escalabilidad y mantenimiento/soporte 137
7.3. Plan de formación de usuarios 138

8. Conclusiones **141**
8.1. Objetivos alcanzados 141
8.2. Conclusiones del trabajo y personales 142
8.3. Vías futuras 143

9. Referencias **145**

10. Anexos **151**

10.1. Manual de instalación *STONNE* 151

10.1.1. Compilar el *frontend* de *STONNE* 152

10.1.2. Compilar las librerías de conexión a *STONNE* 152

10.2. Manual de usuario del modelo *DLRM* 153

10.2.1. Cambiar la arquitectura aceleradora 156

10.3. *Scripts* pruebas *STONNE* 157

10.4. Ejecuciones realizadas sobre el modelo *DLRM* 158

10.4.1. Grupo de evaluación 158

10.4.2. Grupo 1 159

10.4.3. Grupo 2 161

10.4.4. Grupo 3 162

10.5. Artículo redactado para *SARTECO* 164

Índice de figuras

1. Motivación sistemas de recomendación (Ajitsaria, 2018). 29
2. ML vs DL (IONOS, 2020). 35
3. Neurona Biológica vs Artificiales (Meng, 2020). 36
4. Modelo matemático de una neurona (Requena, 2021). 36
5. NN vs DNN (Ironhack, 2021). 38
6. Arquitectura del modelo DLRM (Naumov y cols., 2019). 39
7. Formato CSR convertido a Multi-hot Encoding. 41
8. DLRM Workload of a sample (Naumov y cols., 2019). 42
9. Arquitectura de *STONNE* (Martínez, Abellán, Acacio, y Krishna, 2020). 43
10. Arquitectura aceleradora *MAERI*. 44
11. Arquitectura aceleradora *SIGMA*. 45
12. Arquitectura híbrida propuesta para DLRM. 46
13. Arquitectura de *Centaur* (Hwang, Kim, Kwon, y Rhu, 2020). 48
14. Actividad de los *frameworks* de *DL* (Hale, 2019). 53
15. Arquitectura híbrida inspirada en *APU*. 54
16. Most used agile metodologies *The State of Agile*, 2019 (Petrova, 2019). 57
17. Historias de usuario (Martin, 2018). 66
18. Estimación de póquer (Manager, 2021). 66
19. Wrike: Product Backlog. 68
20. Planificación temporal. 74
21. Diagrama de Gantt. 75
22. Historias de Usuario para el Sprint 1. 78
23. Tablero para el Sprint 1. 79
24. Ejemplo de una operación *GEMM Sparse*. 81
25. Gráfico de *burn down* para el Sprint 1. 85
26. Historias de Usuario para el Sprint 2. 86
27. Tablero para el Sprint 2. 87

28. *Frameworks* para *deep-learning* escritos en *Python* (Costa, 2020). 87
29. Salida *Embedding DLRM vs benchmark*. 90
30. Salida *DLRM* nativo *vs benchmark*. 91
31. Gráfico de *burn down* para el Sprint 2. 94
32. Historias de Usuario para el Sprint 3. 95
33. Tablero para el Sprint 3. 96
34. EmbeddingBags en *DLRM vs STONNE*. 99
35. Ejecución del modelo *DLRM* en *STONNE* con *MAERI* y *SIGMA*. . 101
36. Modelo *DLRM* nativo *vs STONNE*. 102
37. Gráfico de *burn down* para el Sprint 3. 104
38. Historias de Usuario para el Sprint 4. 105
39. Tablero para el Sprint 4. 106
40. Ciclos de ejecución, grupo evaluación. 110
41. Número de lecturas, grupo evaluación. 111
42. Número de escrituras, grupo evaluación. 111
43. Ciclos de ejecución, grupo 1. 116
44. Número de lecturas, grupo 1. 116
45. Número de escrituras, grupo 1. 117
46. Ciclos de ejecución, grupo 2. 118
47. Número de lecturas, grupo 2. 119
48. Número de escrituras, grupo 2. 119
49. Ciclos de ejecución, grupo 3. 121
50. Número de lecturas, grupo 3. 122
51. Número de escrituras, grupo 3. 123
52. Gráfico de *burn down* para el Sprint 4. 127
53. Logotipo de *STONNE* (Martínez y cols., 2020). 151
54. Salida test de *STONNE*. 153
55. Salida test de *DLRM*. 155
56. Uso de *SIGMA* como fichero de configuración. 156
57. Archivos de configuración en *STONNE*. 156
58. Configuración variable de entorno STONNE. 157

Índice de tablas

1. Características de los *frameworks* evaluados. 52
2. Características de las arquitecturas aceleradoras. 53
3. Metodología Tradicional vs Ágil. 55
4. Product Backlog: Historias de usuario (Wrike, 2021). 67
5. Estimaciones realizadas para el proyecto. 71
6. Estimaciones realizadas para el proyecto. 73
7. Estimación económica del proyecto. 76
8. Duración final Sprint 1. 84
9. Versiones de *frameworks* y librerías. 88
10. Duración final Sprint 2. 93
11. Archivo de configuración para las arquitecturas.
RR - Red de reducción. RD - Red de distribución. CM - Controlador de memoria 100
12. Duración final Sprint 3. 104
13. Variables modificables. 107
14. Grupo de prueba para evaluar el modelo. *Embedding = EmbeddingBag 109
15. Cargas de trabajo sobre el modelo DLRM. 115
16. Duración final Sprint 4. 127
17. Plan de pruebas. 129
19. Caso de prueba 02. 130
18. Caso de prueba 01. 130
20. Caso de prueba 03. 131
21. Caso de prueba 04. 131
22. Caso de prueba 05. 132
23. Caso de prueba 06. 132
24. Caso de prueba 07. 133
25. Caso de prueba 08. 133
26. Caso de prueba 09. 134

27. Caso de prueba 10. 134
28. Caso de prueba 11. 135
29. Caso de prueba 12. 135
30. Caso de prueba 13. 136
31. Caso de prueba 14. 137

Abstract

Recommender systems based on Deep Learning (DL) are a booming area of research nowadays. Today's systems must be able to generate an output (e.g. the probability of clicking on a certain ad) in the most accurate way (re-display or not) and within a strict timeframe for digital companies to maximize their profits.

However, when these trained models are deployed on Datacenters with a large volume of data to be processed (hundreds of thousands of users requesting recommendations), the computational time and the demand for computational resources (computational units, memory consumption, etc.) increases considerably. As a result, the need arises to create new accelerator architectures specialized for processing recommender systems.

In this work, we will characterize the performance of the DLRM (Facebook) recommender system, based on deep neural networks, on a hybrid unified accelerator architecture (architecture for sparse computation and another one for dense computation) that we have proposed to accelerate this workload. For this purpose, the STONNE simulator will be used, which allows to implement and accelerators for DL will be used.

The evaluation of DLRM has been performed using workloads similar to those expected in a real environment. In particular, from our analysis it has been found that DLRM can require up to 1.3 GBs, which is 74 million memory accesses for reading and 284,000 for writing data, and the execution time is often dominated by data fetches that are needed following irregular access patterns. This analysis will make it possible to detect execution bottlenecks to design and implement optimizations on the proposed accelerator.

Keywords: Recommender systems, Deep Learning, Accelerator architectures, Reconfigurable HW, STONNE.

Resumen

Los sistemas de recomendación basados en técnicas de *Deep Learning* (DL) es un área en constante evolución. Los sistemas actuales han de ser capaces de generar una salida (por ejemplo, la probabilidad de hacer clic sobre un determinado anuncio) de la manera más precisa (volver a mostrarlo o no) y dentro de un estricto margen de tiempo para que las empresas digitales maximicen sus beneficios.

Sin embargo, cuando estos modelos ya entrenados se despliegan sobre *Datacenters* con un gran volumen de datos a procesar (cientos de miles de usuarios solicitando recomendaciones), el tiempo de cómputo y la demanda de recursos computacionales (unidades de cómputo, consumo de memoria, etc.) aumenta considerablemente. Como resultado de esto, surge la necesidad de crear nuevas arquitecturas aceleradoras especializadas en procesar sistemas de recomendación.

En este trabajo se va a realizar una caracterización de la ejecución del sistema de recomendación DLRM (Facebook), basado en redes neuronales profundas, sobre una arquitectura aceleradora unificada híbrida (arquitectura para cómputo *sparse* y otra para cómputo *dense*) que hemos propuesto para acelerar esta carga de trabajo. Para ello, se usará el simulador STONNE que permite implementar y evaluar aceleradores para DL.

La evaluación de DLRM se ha realizado mediante cargas de trabajo similares a las esperadas en un entorno real. En particular, de nuestro análisis se ha comprobado que DLRM puede requerir de hasta 1,3 *GBs*, lo que supone 74 millones de accesos a memoria para lectura y 284.000 para escritura de datos, y el tiempo de ejecución está muchas veces dominado por las búsquedas de datos que se necesitan siguiendo patrones de acceso irregulares. Este análisis permitirá detectar cuellos de botella en la ejecución para diseñar e implementar optimizaciones sobre el acelerador propuesto.

Palabras clave: Sistemas de recomendación, Deep Learning, Arquitecturas Aceleradoras, Hardware reconfigurable, STONNE.

1 Introducción

¿Cómo *YouTube* sabe qué vídeo podría interesarme ver a continuación?, ¿por qué *Netflix* me recomienda series y/o películas que podrían interesarme cuando nunca las he buscado?, en ambos casos se usa un modelo de recomendación basado en aprendizaje máquina (*Machine Learning* o *ML*). Los sistemas de recomendación ayudan a los usuarios a encontrar contenido similar y/o atractivo en un corpus grande (GoogleDevelopers, 2018).

A día de hoy, uno de los modelos de recomendación más preciso es el modelo *Deep Learning Recommendation Model* (Naumov y cols., 2019) (*DLRM*), que ha sido propuesto y es actualmente usado por *Facebook* para sus campañas de anuncios. El modelo DLRM ofrece anuncios personalizados a los usuarios, para lograrlo toma como entrada de datos variables categóricas y variables continuas, realiza una serie de operaciones y calcula el porcentaje de que un usuario haga clic en un determinado anuncio. Todos estos términos serán detallados a lo largo de este trabajo.

Dado que existe la necesidad de procesar una cantidad ingente de datos para ofrecer recomendaciones personalizadas mediante *DLRM* a millones de usuarios dentro de un estricto margen de tiempo (muchas veces en tiempo real), se hace necesario ofrecer a las empresas que usen este modelo el soporte computacional más eficiente, reduciendo al máximo su tiempo de ejecución, el consumo de recursos computacionales (unidades de cómputo, memoria, red o disco), así como el consumo energético.

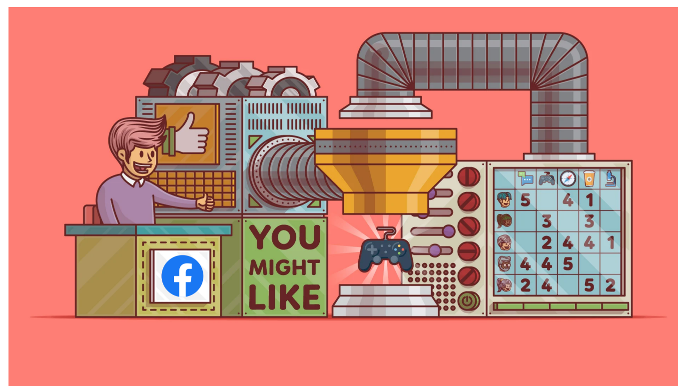

Figura 1: Motivación sistemas de recomendación (Ajitsaria, 2018).

1.1 Motivación

Actualmente los sistemas de recomendación han emergido como una herramienta clave para abordar tareas de personalización y recomendación. Los sistemas de recomendación actuales han de ser capaces de recomendar ítems a millones de usuarios en cualquier momento, por lo que aspectos fundamentales como el tiempo de ejecución o el consumo de energía no pueden caer en el olvido. De lo contrario, se podrían producir grandes pérdidas (campañas de anuncios, pérdida de consumidores, etc.).

En un día, se consumen en promedio un total de un billón de horas en vídeos de *YouTube* y el 70 % de estas horas pertenecen a recomendaciones generadas por el algoritmo de recomendación (Rodriguez, 2018). Dada la gran cantidad de tiempo que se emplea en ejecutar estos algoritmos, es imprescindible lograr que su ejecución sea lo más eficiente posible para reducir su tiempo de ejecución, así como minimizar el consumo de energía y memoria de las plataformas de cómputo sobre las que se ejecutan.

Desde una perspectiva algorítmica (SW), inicialmente los sistemas de recomendación hacían uso de técnicas como *content filtering* (Karbhari, 2017) y posteriormente evolucionaron a *collaborative filtering* (Schafer, 2007) (ambas técnicas serán explicadas en la Sección 2.3.3). Haciendo uso de algunos métodos como *k-vecinos* ó *asociación por grupos* se han logrado obtener buenos resultados, pero, no ha sido hasta hace poco cuando los sistemas de recomendación han empezado a hacer uso de técnicas de aprendizaje profundo (*deep learning* ó DL) por la potencia que ofrecen.

Los modelos de recomendación actuales difieren de otros modelos de redes neuronales basados en *DL*, por la necesidad de tratar con ingentes cantidades de datos categóricos, como por ejemplo, la categoría de los vídeos que el usuario ha visualizado, el género de las películas, tipo de anuncios, etc (más adelante nos referiremos a esta variable categórica como *sparse features* ó *sparse data*). Es por ello que los modelos de recomendación necesitan hacer uso de cientos de *gigabytes* para realizar recomendaciones precisas. Como consecuencia, el tiempo necesario para procesar toda esta cantidad de datos puede llegar a ser

abrumador, por lo que es necesario aunar distintas estrategias de optimización a todos los niveles de la pila computacional (desde la definición del modelo hasta la plataforma de cómputo sobre la que se ejecuta).

Esta metodología de optimización está basada en un co-diseño SW/HW creando plataformas especializadas para alcanzar la máxima eficiencia computacional del modelo ejecutado (al modelo ejecutado se le denomina comúnmente la carga de trabajo). Un ejemplo de esto sería la arquitectura desarrollada, propuesta y utilizada por *Google* actualmente en sus *CPDs*, *Tensor Processing Unit* o *TPU* (Jouppi, 2017), que es un chip especializado (*Application-Specific Integrated Circuit* o *ASIC*) destinado a acelerar las operaciones de Multiplicación y Acumulación de tensores (un tensor es una estructura de datos multi-dimensional como puede ser un vector o matriz) que son claves en el procesamiento de las redes neuronales profundas (tanto para entrenamiento como para inferencia). Más aspectos sobre la *TPU* y el proceso de inferencia en las próximas secciones.

Como se explicará posteriormente, las cargas de trabajo del modelo *DLRM* se pueden descomponer en varias unidades computacionales de *DL* (a los que denominaremos componentes de DLRM), siendo el procesamiento de *MLPs* y un algoritmo denominado *EmbeddingBag* (ambas unidades serán explicadas en las próximas secciones) las que más porcentaje del tiempo de ejecución consumen: 5 % y 47 % respectivamente en CPU y 30 % y 25 % respectivamente en GPU (Naumov y cols., 2019) (ver Figura 8).

1.2 Definición

Dada la enorme repercusión que el modelo DLRM tiene en grandes empresas como Facebook para sus sistemas de recomendación, así cómo la necesidad de optimizar cada vez más su ejecución para ahorrar tiempo de cómputo y consumo de recursos (por ejemplo, memoria y energía) en los CPDs donde se ejecutan, al igual que hizo Google con su TPU para optimizar el procesamiento de *Deep Neural Networks, DNNs*, es necesario proponer nuevos sistemas de cómputo que sean capaces de acelerar eficientemente la ejecución del modelo de DLRM en su fase de inferencia.

Este proyecto supondrá un primer paso para este fin, de manera que se utilizará un simulador de arquitecturas aceleradoras para DNNs llamado *Simulation TOol of Neural Network Engines* (Martínez y cols., 2020), *STONNE*, sobre el que se integrará el modelo DLRM (desarrollado con el *Framework* de *PyTorch*) y al que configuraremos un acelerador híbrido especializado en acelerar cada una de las dos unidades básicas de cómputo del algoritmo DLRM (MLPs y EmbeddingBags). Mediante una caracterización detallada de la ejecución de DLRM en dicho acelerador simulado, se identificarán de manera precisa los cuellos de botella que surgen en la ejecución de DLRM. De esta manera, se podrá entender mejor cómo optimizar el procesamiento de DLRM en sistemas aceleradores y así poder crear un diseño del acelerador mejorado. Esto último, queda fuera de los objetivos alcanzables en este proyecto y será tenido en cuenta como trabajo futuro.

1.3 Objetivos propuestos

A continuación se van a exponer los objetivos del presente TFG.

1.3.1 Objetivos Generales

1. **Implementación y evaluación del modelo DLRM en arquitectura aceleradora híbrida simulada con *STONNE***
   Comprender los cuellos de botella que se producen en la ejecución del modelo DLRM para poder diseñar un acelerador más eficiente especializado en acelerar la ejecución de DLRM. Se usará la herramienta de simulación de aceleradores de DNNs llamada *STONNE*, se integrará DLRM en STONNE, se configurará un acelerador híbrido base en STONNE y, tras caracterizar la ejecución de DLRM sobre dicho acelerador, se identificarán los cuellos de botella que se tendrían que solucionar para acelerar su ejecución.

1.3.2 Objetivos Específicos

1. **Dar soporte en *STONNE* a la ejecución de todas las capas de *DLRM***
   Permitir que la herramienta de simulación *STONNE* sea capaz de ejecutar

nuevas capas como la EmbeddingBag a través de su implementación.

2. **Conectar *DLRM* con *STONNE***

   El modelo *DLRM* llamará a la API de *STONNE* para simular la ejecución de las diferentes capas que lo componen.

3. **Ejecutar *DLRM* en *STONNE***

   Una vez *STONNE* soporta la ejecución de nuevas capas requeridas por el modelo *DLRM* y su conexión se ha conseguido, el siguiente paso será ejecutar *DLRM* sobre *STONNE*.

4. **Caracterizar el rendimiento del modelo *DLRM* mediante un acelerador simulado en *STONNE***

   Haciendo uso de conjuntos de datos (*datasets*) sintéticos que son representativos de cargas de trabajo en entorno de despliegue real de DLRM, se realizará un análisis detallado de la ejecución de *DLRM* en una nueva propuesta de arquitectura aceleradora simulada con *STONNE*. Tras este análisis, se identificarán los principales cuellos de botella de ejecución que limitan el rendimiento del acelerador.

2 Estado del arte

2.1 Conceptos relevantes del dominio de aplicación

Hoy en día la *Inteligencia Artificial* ó *IA* es un campo próspero e innovador en varios temas de investigación y con objetivo de automatizar tareas humanas. La inteligencia artificial necesita adquirir su propio conocimiento a partir de datos sin procesar, máquinas sean capaces de auto-programarse, en otras palabras, queremos máquinas que aprendan de su propia experiencia. La disciplina del Aprendizaje Automático o *Machine Learning*, *ML*, se ocupa de este reto.

Dentro del aprendizaje automático existe lo que se denomina aprendizaje supervisado, pues requiere de la intervención de los humanos para indicar qué está bien y qué está mal. Es obvio que actualmente sería imposible para un humano clasificar todos los tipos de datos que estos sistemas de aprendizaje van a procesar, es por eso que el aprendizaje supervisado ha evolucionado hacia un aprendizaje no supervisado. También llamado aprendizaje automático.

En este paradigma los algoritmos son capaces de aprender sin intervención humana previa, sacando ellos mismos las conclusiones acerca de la semántica embebida en los datos. Aquí es donde aparece el término *Deep Learning DL*, que está muy de moda en la actualidad por su capacidad de acercarse cada vez más a la potencia perceptiva humana.

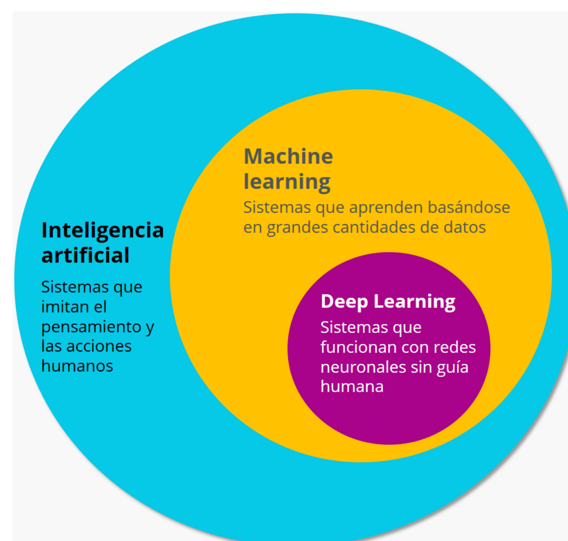

Figura 2: ML vs DL (IONOS, 2020).

El presente trabajo se centra en estructuras/algoritmos de *DL*, es por eso crucial entender su funcionamiento básico; usa estructuras lógicas que se asemejan en mayor medida a la organización del sistema nervioso (Arrabales, 2016). La Figura 3 representa el funcionamiento de nuestras neuronas y como estas se comunican; también, plasma como las neuronas “artificiales” que usan los modelos de *DL* se nutren de información; la salida de una neurona está conectada a las entradas de múltiples neuronas.

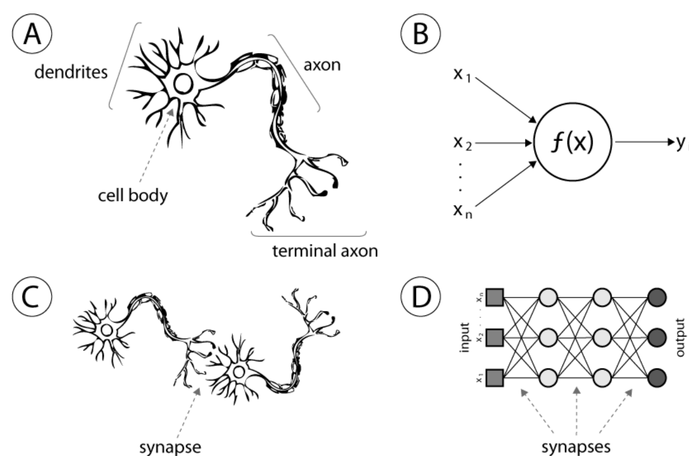

Figura 3: Neurona Biológica vs Artificiales (Meng, 2020).

Pero, ¿cuál es el funcionamiento de dichas neuronas “virtuales”?, el primer modelo matemático de una neurona fue la denominada neurona *McCulloch-Pitts*. Como se observa en la Figura 4 existen algunos símiles entre el modelo matemático y el real, una señal de entrada (dendrita), el procesamiento de los datos (soma o cuerpo celular) y la salida (axón), esta salida que sirve de entrada para posteriores neuronas.

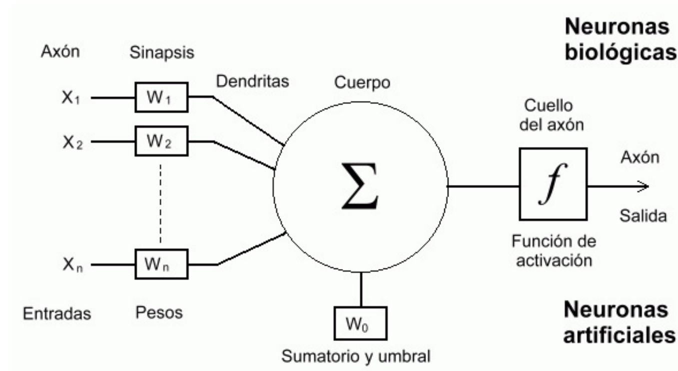

Figura 4: Modelo matemático de una neurona (Requena, 2021).

La primera red neuronal artificial se creó en 1943, con el modelo de neurona acuñado por *McCulloch*. Una red neuronal está compuesta por un conjunto de neuronas conectadas entre ellas, cada neurona tiene la capacidad de aprender y transmitir ese conocimiento a otras neuronas (Ceja, 2020).

Las redes neuronales usadas hoy en día en la mayoría de los modelos tienen numerosas capas, cada una de estas compuestas por diferentes neuronas. De entre todas las capas que conforman un modelo, hay una serie de capas muy bien identificadas:

1. **Capa de entrada**, también llamada *Input Layer*, esta primera capa se encarga de introducir los datos al modelo.
2. **Capa oculta**, conocida como *Hidden Layer*, en esta capa, los datos se procesan a través de funciones de transformación y/o activación.
3. **Capa de salida**, ó *Output Layer*, produce las salidas del programa.

Como se observa en la Figura 5 una red neuronal, *NN*, está compuesta por una capa oculta, mientras que las redes neuronales profundas, *Deep Neural Networks* ó *DNN*, están compuestas de *N* capas ocultas. La diferencia entre el número de capas ocultas es un factor determinante en los modelos de DL ya que cada una transmite un determinado valor a las consiguientes capas. Las técnicas de *DL* muy frecuentemente usan numerosas capas ocultas con el fin de que sea la propia red neuronal la que se entrene a sí misma.

La entrada de datos en los modelos de recomendación (los vídeos que el usuario ha visualizado, el género de las películas, tipo de anuncios, etc.) son tratados como variables categóricas, *sparse features*, término que explicaremos más adelante. El modelo de recomendación *DLRM* difiere significativamente de otros modelos de de *DL* debido a la necesidad de tratar variables categóricas, como por ejemplo las *CNNs*, *LSTMs* ó *RNNs*. En el artículo donde Facebook presenta *DLRM*, se hace especial atención a la “necesidad que existe de desarrollar modelos de recomendación que optimicen el cómputo de *sparse-data*” (Naumov y cols., 2019) ya que representa una parte significativa del cómputo total de los modelos actuales.

Generalmente, una DNN tiene dos fases fundamentales: una primera fase de *entrenamiento*, donde un conjunto de pesos son entrenados para que la DNN realice una determinada tarea, y una segunda fase de *inferencia*, donde la DNN se despliega para ser utilizada en un escenario para el cual ha sido entrenada previamente.

Normalmente, la fase de entrenamiento se lleva a cabo utilizando GPUs en un gran centro de datos, mientras que la fase de inferencia se suele realizar *in-situ*, en dispositivos con fuertes restricciones en cuanto a área y energía. Este hecho ha conllevado a la investigación y desarrollo de un gran número de arquitecturas aceleradoras que tratan de maximizar las demandas de eficiencia energética de estos escenarios.

Es por eso que en este proyecto nos centramos en la fase de inferencia para lograr proponer una mejora arquitectural al modelo de recomendación actual. Vamos a dejar al margen el entrenamiento del modelo.

A partir de este punto podemos comenzar a introducir el modelo de red neuronal *DLRM*, presentado por *Facebook*, junto con su arquitectura; este modelo de *DL* está concebido por la unión de dos perspectivas: Las diferentes técnicas de sistemas de recomendación (*content filtering* y *collaborative filtering*, ver Sección 2.3.3) y el análisis predictivo, técnicas estadísticas que analizan los datos para clasificar o predecir la probabilidad de que un evento suceda. Los modelos predictivos han evolucionado desde modelos simples (regresión lineal o logística) a

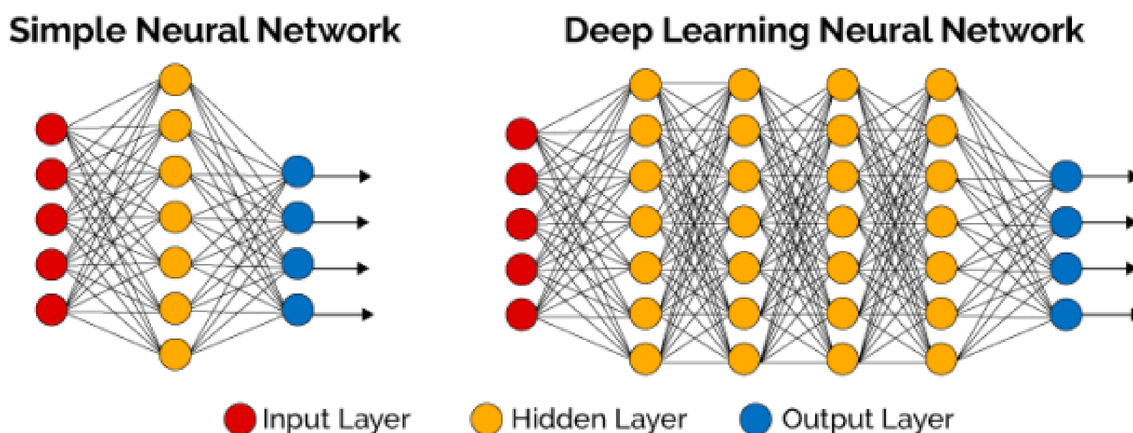

Figura 5: NN vs DNN (Ironhack, 2021).

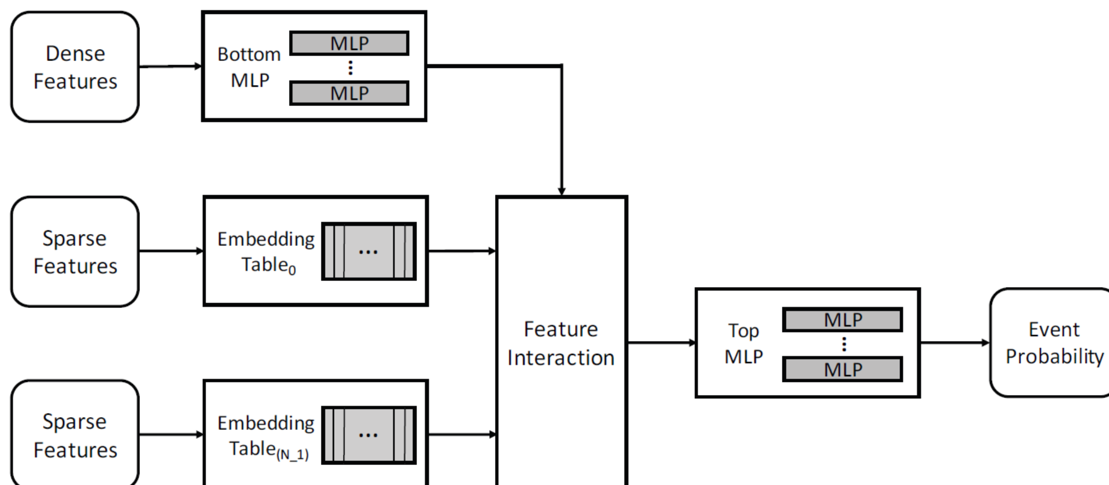

Figura 6: Arquitectura del modelo DLRM (Naumov y cols., 2019).

modelos que incorporan redes profundas (DL).

Como se observa en la Figura 6, el modelo recibe 2 entradas de datos diferentes; *dense features* y *sparse features* (más detalles abajo); el modelo usa *N EmbeddingTables*, nos referiremos a ellas como *EmbeddingBags*, para procesar las entradas *sparse*, (variables categóricas) y un *multilayer perceptron*, (*MLP*), para procesar las entradas *dense*. Posteriormente la salida de ambos algoritmos se combina en una fase denominada *Feature-Interaction* y se post-procesa en una *Top-MLP*, que muestra la probabilidad de que un evento suceda.

El modelo está compuesto por 4 componentes muy bien identificados: Una *Bottom-MLP*, **N** *EmbeddingBags*, una fase *Feature-Interaction* y finalmente, una *Top-MLP*.

En matemáticas *sparse* y *dense* son términos frecuentemente referidos al numero de ceros contenidos en un vector o matriz. Por ejemplo, una matriz *sparse* estará compuesta mayoritariamente de ceros, mientras que un vector dense, la mayoría de sus elementos serán no-nulos.

En *DLRM* las entradas *dense* (información a cerca del usuario, historial de estadísticas, predicciones de otros modelos (Haldar, 2019), etc) son procesadas por una *Bottom-MLP*, un *multilayer perceptron*, es un tipo de red neuronal compuesta por múltiples capas (una capa de entrada, varias capas intermedias *ocultas* y una capa de salida), a su vez cada una compuesta por diferentes neuronas. La

salida de cada capa es disjunta, es decir, la salida de una neurona es la entrada a varias. Un ejemplo de este tipo de redes se puede ver en la Figura 5.

Por otra parte las entradas *sparse* (categoría de los vídeos que el usuario ha visualizado, el género de las películas, tipo de anuncios, etc) serán procesadas por *EmbeddingBags*, un algoritmo que haciendo uso de dos vectores: *indices* y *offsets*; realizará búsquedas (*lookups* en adelante), en cada *EmbeddingBag* utilizando el formato *Compressed Sparse Row* (*CSR*); formato bien conocido para reducir considerablemente el tamaño de las matrices *sparse*. Con los índices seleccionados se realizará una determinada operación vectorial, como por ejemplo, el producto vectorial, *dot-product*. En la Figura 7 se encuentra un ejemplo simplificado de la conversión formato *CSR* a vector *sparse* (llamado multi-hot encoding en las futuras secciones).

A lo largo de este proyecto mencionamos las palabras *Embeddings* y *EmbeddingBags*, en realidad, ambas palabras son lo mismo. Ambas se refieren al algoritmo capaz de procesar datos *sparse*. Si nos fijamos en la documentación oficial de *PyTorch*, “El algoritmo *EmbeddingBag* computa la mediana, o la suma, de varias *Embeddings* simultáneamente sin instanciar *Embeddings* intermedias”. El concepto *EmbeddingBag* puede verse como una “bolsa” de *Embedings*. Si hemos optado por *EmbeddingBags* en lugar de *Embeddings* es debido a: “*however, EmbeddingBag is much more time and memory efficient than using a chain of Embeddings*” (PyTorch, 2021b).

Finalmente, en la fase *Feature-Interaction* la salida de las *EmbeddingBags* se concatena con la salida de la *Bottom-MLP* y se hace una factorización de matrices, técnica que explicaremos más adelante, y el resultado, una matriz dense, se envía a esa *Top-MLP* que devuelve finalmente la probabilidad de que un evento suceda.

Todas estas técnicas presentadas serán detallas a lo largo del trabajo (ver Sección 6.1.2).

Si nos detenemos a observar una caracterización de una carga de trabajo del modelo *DLRM* tanto sobre CPU como GPU , en la Figura 8 observamos cómo la fracción del tiempo destinada las *EmbeddingBags* es bastante significativo

(apréciense las categorías *EmbeddingBagBackward* y *embedding_bag**). De este modo, en *DLRM* no solamente las MLPs tendrán que ser aceleradas (como en otros modelos DNNs) sino que también las EmbeddingBags.

Lo que se quiere proponer y lograr con este trabajo es demostrar la necesidad de diseñar arquitecturas aceleradoras sobre partes específicas de gran carga de cómputo. Un ejemplo ya mencionado anteriormente pero sin entrar en detalle, es la arquitectura *TPU*, diseñada por Google. En el artículo en el que se presenta la TPU (Jouppi, 2017) se hace especial atención a la necesidad de desarrollar arquitecturas que sean específicas de dominio para lograr un gran avance en

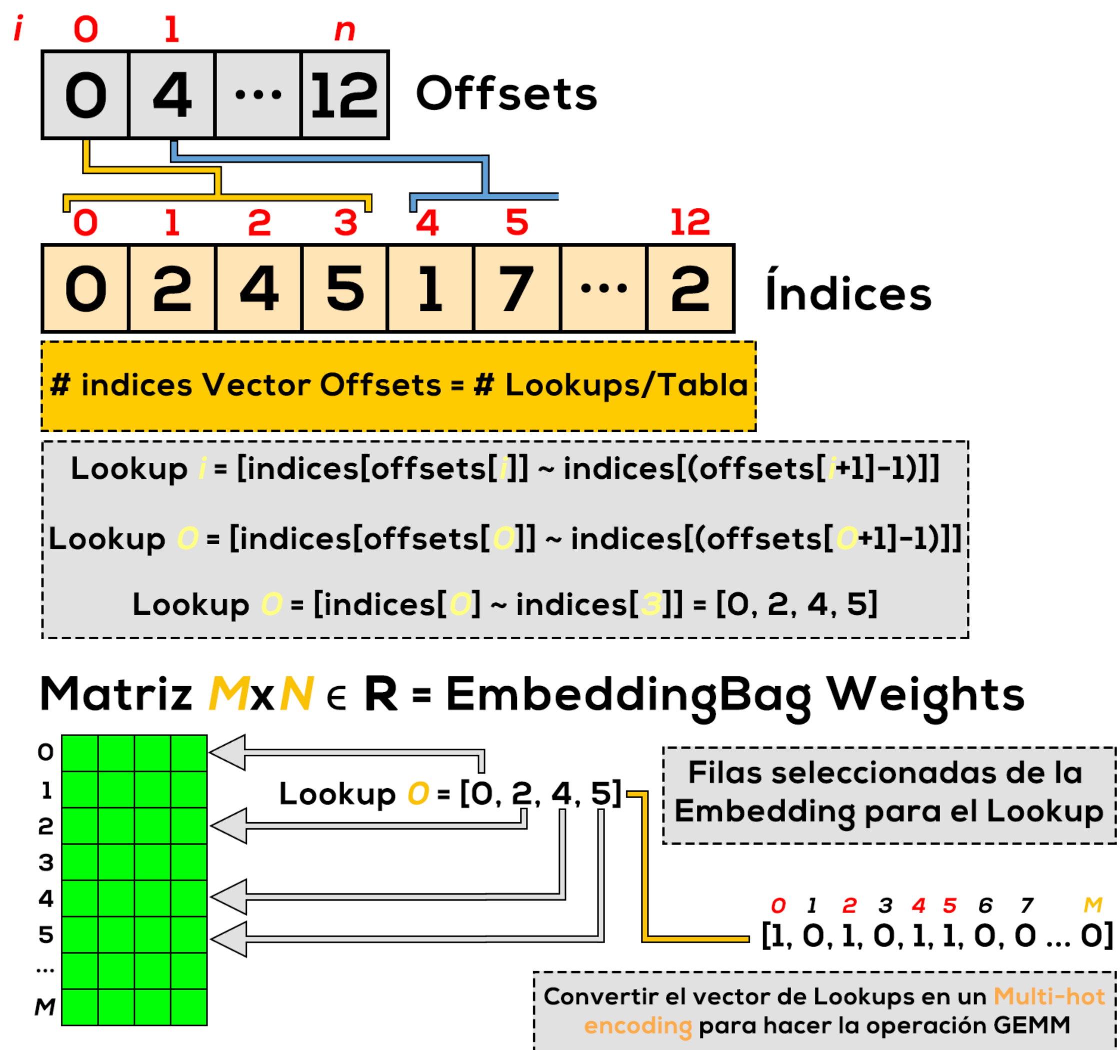

Figura 7: Formato CSR convertido a Multi-hot Encoding.

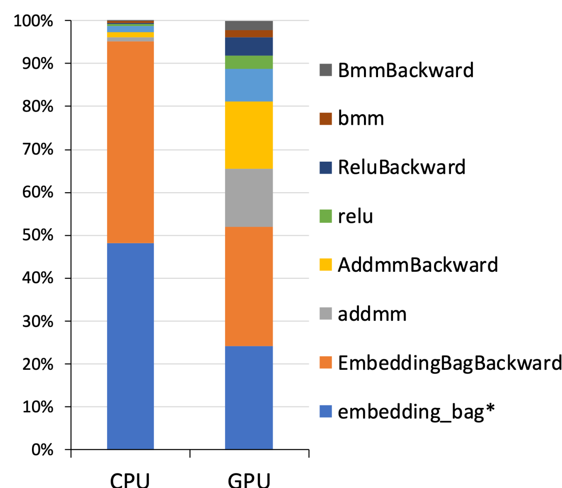

Figura 8: DLRM Workload of a sample (Naumov y cols., 2019).

coste-energía-eficiencia, el acelerador *TPU*, *Tensor Processing Unit* es un *ASIC* personalizado, *application-specific integrated circuit*, que se ha desarrollado con la intención de centrarse únicamente en acelerar la multiplicación de matrices haciendo uso de multiplicadores especializados, *Processing Elements PEs*. La mayoría de modelos de DL usados por Google como *MLPs*, *CNNs* o *LSTMs* representan el 95 % de la carga de sus centros de datos, la parte más intensiva en cómputo de estos modelos se encuentra en multiplicaciones de matrices.

En el caso del modelo *DLRM* ocurre de igual manera, el mayor tiempo de cómputo ocurre en las *MLPs y EmbeddingBags* por eso es necesario remarcar la necesidad de desarrollar una arquitectura aceleradora que reduzca tiempos e intensifique el cómputo.

Para lograrlo es necesario, al igual que en el caso de la *TPU*, desarrollar una arquitectura aceleradora; en el caso del modelo *DLRM*, hay dos tareas intensivas en cómputo, *MLPs* y *EmbeddingBags* con la operación denominada *Embedding Lookups*, es por eso que introducimos un diseño de acelerador híbrido compuesto por dos aceleradores específicos para inferencia de DNNs para acelerar el cómputo *dense* de las MLPs (mediante la arquitectura del acelerador *MAERI* (Kwon, Samajdar, y Krishna, 2018)), y acelerar el cómputo *sparse* presente en las *EmbeddingBags* (mediante la arquitectura del acelerador *SIGMA* (y otros, 2020)).

¿Como vamos a procesar el proceso de inferencia del modelo *DLRM* me-

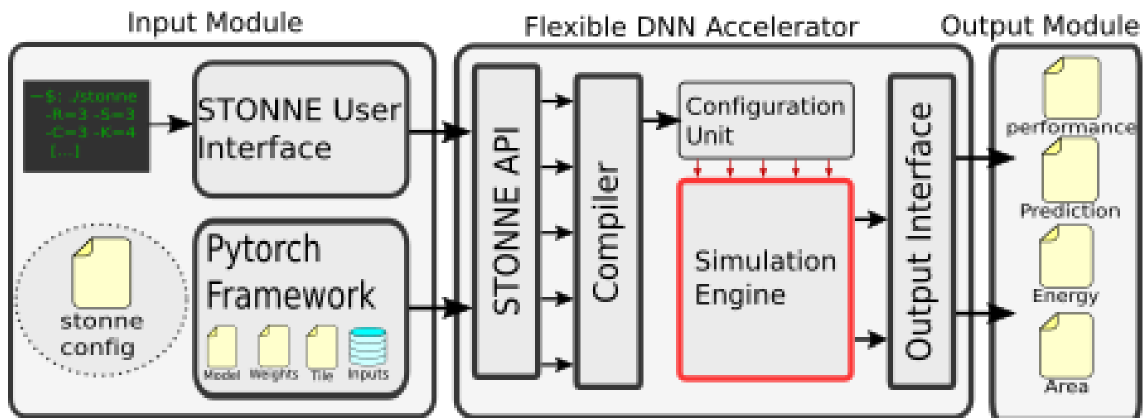

Figura 9: Arquitectura de *STONNE* (Martínez y cols., 2020).

diante ese acelerador híbrido? ¿Existe alguna herramienta que lo haga posible? ¿Están esas arquitecturas aceleradoras ya desarrolladas? En la Sección 1 introducimos brevemente *STONNE*, pero, como es un concepto muy importante de entender y de comprender, vamos a explicarlo.

STONNE es un simulador con precisión a nivel de ciclo y reconfigurable, que permite modelar distintos tipos de arquitecturas aceleradoras para *deep-learning*. Además, permite ser integrado dentro de un framework de *DL*, como *PyTorch*, donde se puede hacer un ejecución completa de redes neuronales profundas (también llamada ejecución *end-to-end*). *STONNE* ya cuenta con varias arquitecturas aceleradoras implementadas y desarrolladas como *MAERI*, *SIGMA* y la *TPU*.

En la Figura 9 se muestra la arquitectura de *STONNE*. Vamos a explicar muy brevemente el concepto de *STONNE* y de como el modelo *DLRM* se va a ejecutar sobre él, en la Sección 6 ampliaremos conceptos y lo explicaremos más detalladamente. *STONNE* va a ejecutar dos componentes *simulados* del modelo *DLRM*, las *MLPs* y las *EmbeddingBags*, estos componentes van a estar escritos en el *framework* de *PyTorch* que *STONNE* utiliza. Al mismo tiempo, existe un archivo de configuración, *stonne config*, en este archivo se especificará que arquitectura aceleradora queremos utilizar y la configuración de esta (número de multiplicadores, controlador de memoria, ancho de banda de los buses, etc), aquí es donde especificaremos las arquitecturas *MAERI* y *SIGMA*. Con todo esto, *STONNE* ejecutará el modelo con las arquitecturas aceleradoras (en la unidad

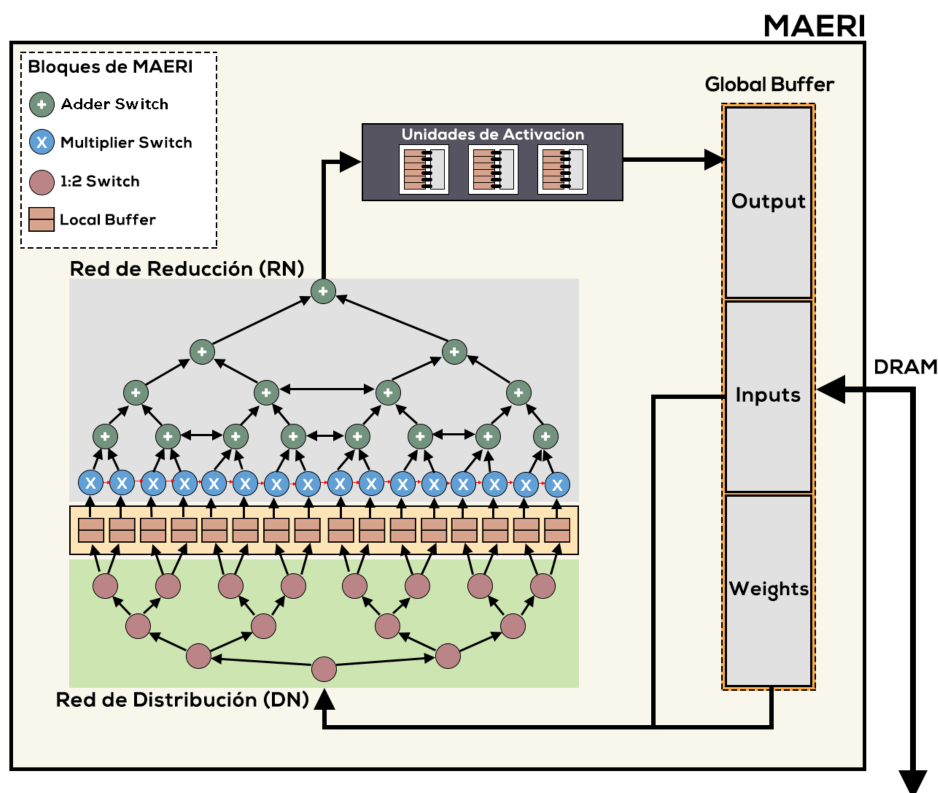

Figura 10: Arquitectura aceleradora *MAERI*.

Simulation Engine) y posteriormente devolverá como salida (unidad Output Interface) una serie de archivos. En estos archivos se pueden ver el número de ciclos que ha tomado la simulación, así como otras estadísticas interesantes como el número de escrituras y de lecturas a estructuras de memoria interna al acelerador, a memoria principal, el número de envíos a través de las redes de interconexión del acelerador, el porcentaje de utilización de las unidades de cómputo, el consumo energético, entre otras. Una vez comprendida la herramienta de simulación *STONNE*, vamos a introducir las dos arquitecturas aceleradoras que usaremos durante este trabajo.

MAERI es un acelerador modular para *DNNs*. Es común tener varias *DNNs* con diferentes filtros, tamaños de entrada, datos *sparse* o *dense*, especialmente hoy en día con modelos como *FC*, *CNNs*, etc. *MAERI* es una acelerador que consta de una red de distribución, *distribution network* ó *DN*, donde los pesos y las entradas se distribuyen por toda la red; unos *local buffers*, donde se almacenan y posteriormente pasan a la red de reducción, *reduction network*, *RN*, donde se comienza a realizar el cómputo *dense*. Dentro de nuestro modelo *DLRM*, *MAERI* se

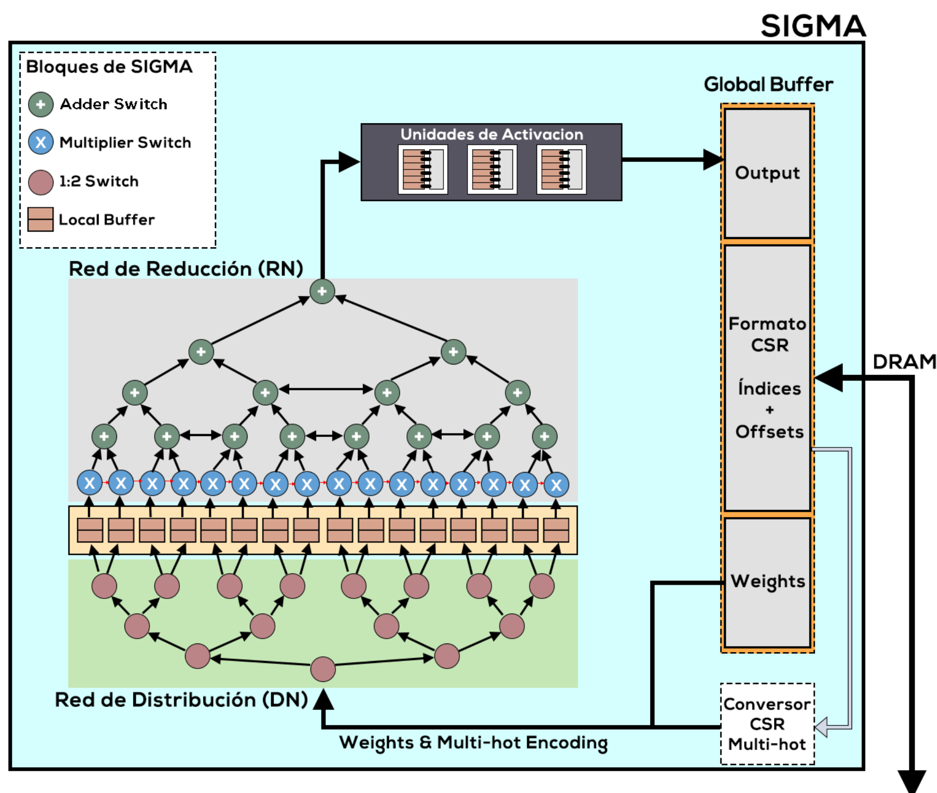

Figura 11: Arquitectura aceleradora *SIGMA*.

va a encargar de realizar el cómputo dense (*Bottom y Top MLP*). En la Figura 10 se muestra la arquitectura de *MAERI* de forma detallada.

Por otra parte *SIGMA* es una acelerador sparse para operaciones *GEMM sparse-dense*, ¿qué es una operación *GEMM*? Las operaciones *GEMM* son el corazón del *DL*, *GEneral Matrix to Matrix Multiplication*. Si nos fijamos en las operaciones que suceden en las *EmbeddingBags*, sin entrar en mucho detalle, se puede resumir en que, a partir del formato CSR (vectores *indices* y *offsets*) que se encuentran en el *Global Buffer*, se crea un vector *sparse* denominado *multi-hot encoding*, donde cada valor no-nulo hace referencia a un índice de la *EmbeddingBag* seleccionado, como se observa en la Figura 7. Una vez conformado el vector se realiza una multiplicación vector-matriz, vector multi-hot encoding (sparse) por la matriz de pesos de la *EmbeddingBag* (dense). Ya que el vector multi-hot confeccionado está compuesto en su mayoría de ceros, necesitamos un diseño que acelere esta operación *GEMM sparse-dense* (la Figura 24 muestra un ejemplo de operación *GEMM sparse-dense*), es aquí donde introducimos el segundo diseño, *SIGMA*. En la Figura 11 se muestra la arquitectura detallada de *SIGMA*.

En este trabajo no queremos estudiar las diferencias arquitecturales que existen entre *MAERI* y *SIGMA*, puesto que cada una puede tener diferentes maneras de implementar la red de reducción ó la red de distribución, etc., en su lugar, queremos remarcar la diferencia conceptual que existe en el *Global Buffer*, ¿Qué datos se almacenan?, ¿Cómo se extraen? y ¿Qué operaciones se realizan con dichos datos?; en el lugar de *SIGMA*, se realiza una conversión *CSR* previa a introducirlos a la red de distribución. Este conversor va cogiendo los *índices* y los *offsets* en formato *CSR* y los convierte en un *multi-hot encoding*, al mismo tiempo que los va enviando a la red de distribución junto con los pesos de la matriz *EmbeddingBag*, que se van distribuyendo por toda la red. Finalmente, en los multiplicadores de la Figura 11, se realiza el cómputo *sparse*.

Proponemos una arquitectura acelerada híbrida como la que se ve en la Figura 12, compuesta por los aceleradores, *MAERI* y *SIGMA*, para acelerar las partes intensivas en cómputo, ambas teniendo acceso a la *DRAM* y a un *Global Buffer* que suponemos ilimitado (en la Sección 6.3.2 daremos más detalle a cerca de la configuración escogida para los aceleradores y el tamaño del *Global Buffer*). Para la arquitectura propuesta nos hemos inspirado en la Figura 3 del artículo *HyGCN* (Yan y cols., 2020).

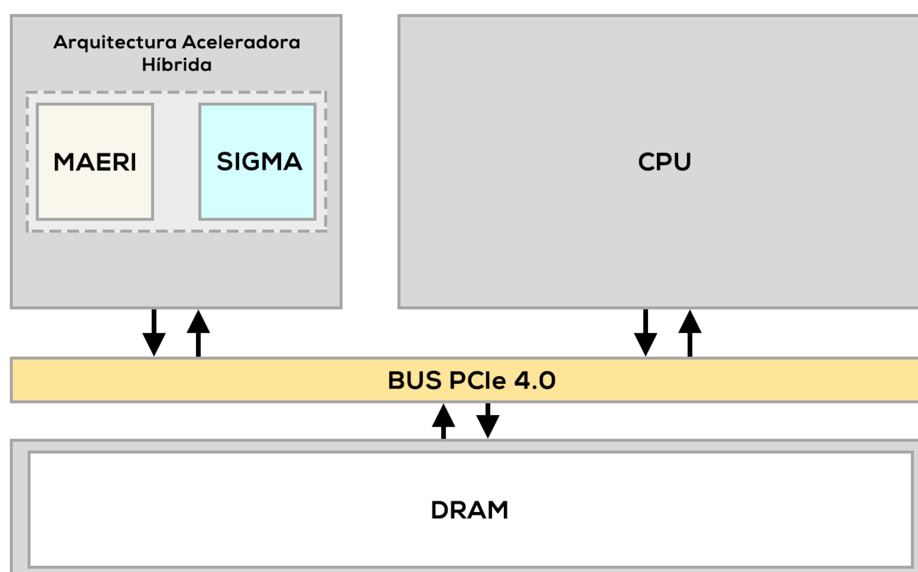

Figura 12: Arquitectura híbrida propuesta para DLRM.

2.2 Relación con proyectos con la misma funcionalidad

En la actualidad existen diferentes modelos de recomendación usados por diferentes compañías, entre ellas *Google*, *Netflix*, *HBO*, *Instagram*, *Facebook*, etc. empresas destinadas a ofrecer servicios a los usuarios. Entre todas ellas destaca le necesidad de gestionar tanto datos *dense* como *sparse*, aunque la arquitectura de este modelo de recomendación no sea la misma a la de los sistemas de recomendación usados por otras empresas, sí tienen aspectos similares.

Aparte del modelo de recomendación utilizado en el presente trabajo, en el mercado existen diferentes aproximaciones/sistemas de recomendación en auge que proponen alternativas o diferentes arquitecturas, muchos de ellos sobre mejoras en *training*, pero, puesto que vamos a presentar una arquitectura híbrida para inferencia, vamos a hablar sobre proyectos relacionados enfocados a la inferencia; en este punto vamos a debatir sus puntos fuertes y vamos a poner de manifiesto las diferencias que existen con lo que proponemos en este trabajo.

En el artículo *Centaur* (Hwang y cols., 2020), presentan un acelerador híbrido *multi-chiplet* (múltiples *chips* dentro de un mismo *chip*) (*CPU-Chiplet* y *FPGA-Chiplet*) para operaciones *sparse-dense*. En la primera parte del artículo muestran como de intensiva en término de memoria son las *embedding-layers* y como de intensivas en cómputo son las *MLPs*, problema que hasta ahora ya hemos mencionado y de ahí la necesidad del presente trabajo, desarrollar una arquitectura híbrida que apunte hacia estos problemas. La principal diferencia entre *Centaur* y el modelo que proponemos, aunque tenga el mismo objetivo, es la diferente manera en la que la arquitectura del *multi-chiplet* se presenta. *Centaur* propone una arquitectura compuesta, formada por 2 *chiplets CPU* y *FPGA*, en la Figura 13 se muestra la arquitectura propuesta. *Centaur* propone un diseño de comunicación entre *chiplets* mediante un bus de PCIe, donde los módulos encargados de realizar las operaciones *sparse-dense* han sido diseñados específicamente dentro de la *FPGA*. La arquitectura híbrida propuesta en el presente trabajo introduce dos diseños aceleradores de uso específico para tareas *sparse-dense*.

En la Sección 2.3.4 explicamos porque hemos descartado *Centaur*. En caso de querer saber más información acerca de la arquitectura propuesta y saber

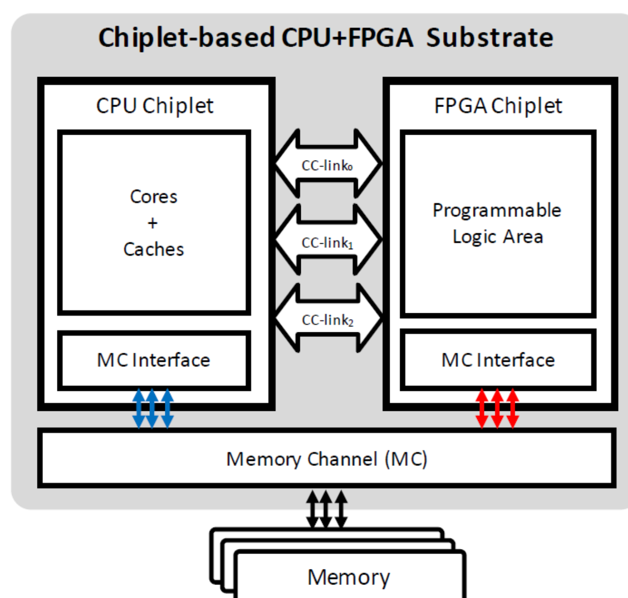

Figura 13: Arquitectura de *Centaur* (Hwang y cols., 2020).

los módulos incluidos dentro de la *FPGA*, me remito a la Imagen 9 del artículo *Centaur*.

Otro artículo relacionado es *DeepRecSys* (Gupta y cols., 2020), proponen una infraestructura de modelación *end-to-end* para diseñar modelos propios de recomendación. En esta arquitectura propuesta, incluyen un planificador que va junto con las peticiones de la *CPU*, este planificador también se encarga de medir la latencia que se acumula paulatinamente por las peticiones del modelo a la memoria. *DeepRecSys* va más enfocado a como diseñar un modelo de recomendación orientado al *training* y se aleja del objetivo principal de este trabajo.

Finalmente introducimos un tercer artículo más enfocado al estudio/implicación del sistema de recomendación propuesto por *Facebook Architectural Implications of Facebook's DNN* (Gupta y cols., 2019). En este artículo se pone de manifiesto la poca atención que han recibido las arquitecturas de los sistemas de recomendación, en especial la del modelo *DLRM*, a pesar de su importancia. En el artículo presentan diferentes cargas de trabajo (realizadas sobre *CPU* y *GPU*) con el objetivo de demostrar la ineficiencia de los diferentes módulos, dichas cargas de trabajo se asemejan a cargas reales que podrían estar siendo ejecutadas en centros de computación actuales. Como este trabajo supone una linea de referencia para nuestra investigación se han cogido algunas cargas de trabajo similares

a las que evalúan con el fin de probarlas sobre la arquitectura aceleradora que vamos a presentar.

2.3 Estudio de viabilidad

2.3.1 Alcance del proyecto

El proyecto está enfocado en integrar el modelo de recomendación de *Facebook* en *STONNE* para analizar su ejecución sobre un acelerador híbrido propuesto. Como ya hemos comentado *STONNE* es un simulador que permite ejecutar modelos de *DL* de última generación. Sin embargo, no soporta modelos de recomendación. Por lo que como primer paso crucial es necesario permitir que *STONNE* ejecute aquellas capas necesarias para la ejecución del modelo *DLRM*. Una vez que *STONNE* soporte las capas que los modelos de recomendación se usarán diseños de arquitecturas aceleradoras como *MAERI* y *SIGMA* para realizar una evaluación de la ejecución. Este proyecto se centra en la fase de inferencia.

Como trabajo futuro podríamos considerar incluir el *training* con el objetivo de acelerarlo mediante otros sistemas, diseños o arquitecturas. También, como propuesta a largo plazo sería ver el impacto de los diferentes diseños seleccionados, *MAERI* y *SIGMA*, variando su configuración, es decir, ajustando parámetros a más bajo nivel, entre los que encontraríamos ajustes como el *tiling*, *sparsity*, etc. también se podría realizar un estudio sobre la configuración a más bajo nivel en *STONNE*.

Gracias a la caracterización propuesta y realizada en este TFG, los resultados van a ser de gran aporte para la comunidad científica; principalmente dirigido a varios sectores entre lo que encontramos: los arquitectos de hardware, que quieren potenciar los modelos de inferencia mediante la construcción de arquitecturas específicas; a los expertos en *ML*, para que entiendan la importancia de mapear de manera eficiente el modelo *DLRM* y para que adquieran el conocimiento de como desarrollar modelos de recomendación más adecuados al HW que estén usando; a la comunidad de expertos en desarrollo de compiladores para modelos de recomendación, puesto que es fundamental conocer las limitaciones arquitecturales

actuales para mapear las capas de *DLRM* de manera eficiente, lograr hacerles entender las limitaciones actuales con el fin de que estas sean potenciadas.

2.3.2 Estudio de la situación actual

Partimos de *STONNE*, nos permite modelar arquitecturas aceleradoras tales como *MAERI*, *SIGMA* y la *TPU*. También, nos permite hacer ejecuciones *end-to-end* de modelos de *DL*, como *Alexnet*, *Squeezenet*, *BERT*, etc. Sin embargo, *STONNE* no es capaz de implementar un modelo de acelerador híbrido consistente en *MAERI* y *SIGMA*, el cual es necesario para acelerar las cargas de trabajo de *DLRM*.

Por otra parte, tenemos el modelo *DLRM*, desarrollado en *PyTorch*, que cuenta con 2 componentes cruciales, *MLPs* y *EmbeddingBags*. *STONNE* no da soporte para el modelo de recomendación *DLRM*, por lo que, en primer lugar tenemos que implementar las capas necesarias y realizar una evaluación/simulación *end-to-end* (un *benchmark*) para comprobar su funcionamiento. Posteriormente, combinar la ejecución de *MAERI* y *SIGMA* para poder dar soporte completo a la ejecución de *DLRM*.

Actualmente no existe ninguna arquitectura acelerada sobre la que *DLRM* pueda ejecutarse, se han presentado diversas alternativas en el Punto 2.2, pero ninguna es capaz de simular arquitecturas *full-model evaluation*, como *MAERI* y *SIGMA*, un aspecto que *STONNE* permite.

Se prevé que este trabajo ayudará a esclarecer aspectos clave en sistemas de recomendación para futuros modelos, gracias al uso de una arquitectura específica aceleradora para las operaciones intensivas en cómputo del proceso de inferencia en *DLRM*. También servirá para incentivar el desarrollo de nuevos modelos de recomendación con las arquitecturas presentadas en este trabajo.

2.3.3 Estudio y valoración de las alternativas de solución

En un principio se habían planteado diferentes alternativas, diferentes técnicas de modelos de recomendación, como *content-based filtering*, en los que si un usuario A ve dos vídeos de animales, el sistema le recomienda vídeos de ani-

males; y *collaborative filtering*, si un usuario A es similar a un usuario B (por los vídeos que ha visto) y el usuario A indica que le gusta un vídeo, entonces el vídeo se le mostrará al usuario B. Pero en los últimos años se ha comenzado a hacer uso de técnicas híbridas que combinan ambas aproximaciones.

Para permitir la ejecución de *DLRM* en *STONNE*, hay que darle la capacidad a *STONNE* de ejecutar las capas necesarias, el modelo *DLRM* se encuentra desarrollado en diferentes *frameworks*; *PyTorch* y *Caffe*. Por otra parte el simulador *STONNE*, permite la ejecución de código escrito en *PyTorch* y *Caffe*.

Caffe es es un framework de *DL* desarrollado para el procesamiento de imágenes, su mayor uso se encuentra en modelos convolucionales, *CNNs* ya que están optimizadas para ofrecer una mayor velocidad de procesamiento y aprendizaje. Sin embargo, el presenta trabajo se centra en cargas de trabajo sobre capas *MLPs* y *EmbeddingBags*.

Por otra parte, *TensorFlow* es una plataforma de código abierto para el aprendizaje automático desarrollada por *Google*. La arquitectura flexible de *TensorFlow* permite implementar modelos de DL en una o más *CPUs* ó *GPUs*. Algunos de sus usos más populares son: reconocimiento de imágenes o sonidos, aplicaciones de reconocimiento de texto, etc.

También existe *Keras*, biblioteca de redes neuronales escrita en *Python* capaz de ejecutarse sobre *TensorFlow*. Keras está diseñado para posibilitar la experimentación en poco tiempo con redes de DL, permite pasar de la idea al resultado. El uso de este framework se centra en redes neuronales convolucionales, *CNNs* y recurrentes, *RNNs*.

Finalmente, *PyTorch*, es una biblioteca de aprendizaje automático desarrollada por *Facebook* y de código abierto. Se utiliza para aplicaciones que implementan modelos relacionados con visión artificial, procesamiento de lenguajes naturales, etc. El modelo *DLRM* está escrito en *PyTorch* y por otra parte, la herramienta de simulación *STONNE* ofrece soporte para este lenguaje.

Otro punto a tener en cuenta serían las 3 diferentes arquitecturas aceleradoras que se encuentran ya implementadas en *STONNE*, hemos omitido su explicación en esta sección puesto que se encuentran definidas previamente en la

Sección 2.1. En la siguiente sección explicamos el motivo se haber elegido *MAERI* y *SIGMA* como arquitecturas aceleradoras.

2.3.4 Selección de la solución

De los proyectos relacionados que hemos explicado en la Sección 2.2 hemos descartado *Centaur*. ¿Por qué? La arquitectura *multi-chiplet* propuesta por *Centaur* se encuentra desarrollada sobre *FPGAs* (*Field-Programmable Gate Array*), mientras que los aceleradores *MAERI* y *SIGMA*, se encuentran desarrollados sobre *ASICs* (*Application-Specific Integrated Circuit*) programables; las *ASICs* son de propósito muy especifico, hacen muy bien una determinada tarea, aquella que se encuentre programada; una *FPGA* tiene un carácter más general, se puede programar cualquier hardware dentro de sus limitaciones. Puesto que este trabajo se centra en el proceso de inferencia y tenemos muy claro cual es el diseño final, nuestros *ASICs* van a ser mucho más óptimos en cuanto al área, consumo energético, rendimiento, etc. Por eso elegimos un diseño más optimizado basado en *ASIC*, las arquitecturas aceleradoras *MAERI* y *SIGMA* ya mencionadas.

Para implementar el modelo *DLRM*, es decir, dotar a *STONNE* de la capacidad para poder ejecutar las capas necesarias del modelo de recomendación, se ha hecho uso del *framework PyTorch*, puesto que: el modelo de recomendación está escrito en este framework y *STONNE* soporta *PyTorch* (ver Tabla 1).

En este proyecto no se usan redes convolucionales, y el simulador *STONNE*, cada vez opta más por dejar de usar *Caffe*, por lo que, mirando a largo plazo, modelar las capas necesarias en *Caffe* queda descartado. En un principio, para realizar el algoritmo *CSR*, de las *Embedding Lookups*, se optó por usar *TensorFlow*, pero más tarde fue descartado ya que el resto del modelo no podía ejecutarse

	CNNs	RNNs	MLPs	EmbeddingBags
Caffe	✓	✓	✓	✗
TensorFlow	✓	✓	✓	✗
Keras	✓	✓	✓	✗
PyTorch	✓	✓	✓	✓

Tabla 1: Características de los *frameworks* evaluados.

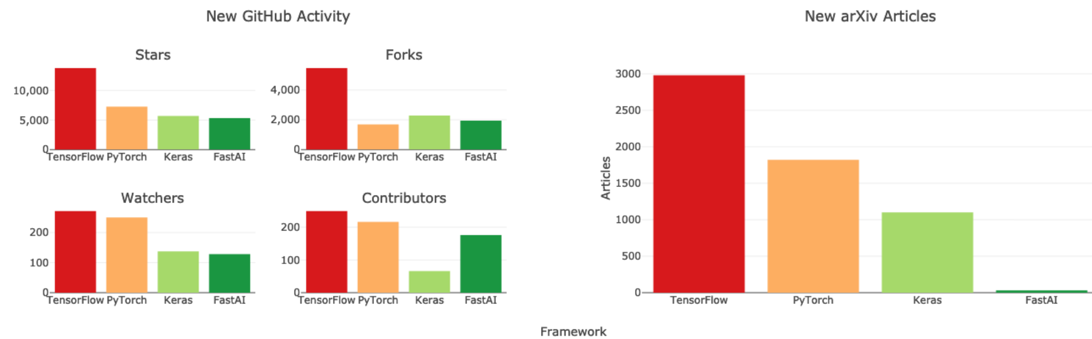

Figura 14: Actividad de los *frameworks* de *DL* (Hale, 2019).

correctamente imitando la ejecución nativa en *PyTorch* del modelo original.

Finalmente, ante las diferentes opciones de implementación se ha optado por realizar la implementación de *DLRM*, de las capas necesarias en *STONNE* haciendo uso de *PyTorch*, ya que es el lenguaje nativo en el que el modelo ha sido desarrollado y puesto que el simulador lo soporta. Extender el simulador *STONNE* con otro framework de *DL* que no sea *PyTorch* es un esfuerzo que queda fuera de la limitación del TFG.

En la Figura 14 se puede observar como existe una creciente popularidad del framework *PyTorch* en las últimas contribuciones dentro de la página de *GitHub* y como el número de artículos publicados en *arXiv* no difiere tanto entre *TensorFlow* y *PyTorch*.

Por otra parte, en *STONNE* se modela la *TPU*, *MAERI* y *SIGMA*. Entre los 3 aceleradores que se encuentran implementados en el simulador, optamos por un híbrido *MAERI-SIGMA*. Al final, la carga de trabajo de DLRM, es una carga de trabajo heterogénea, cómputo *dense* por un lado y cómputo *sparse* utilizando el formato *CSR*. Y usando solamente uno de estos tres modelos no sería eficiente.

También se podía haber optado por utilizar la arquitectura de la *TPU*, en vez de *MAERI*, ya que también soporta el cómputo *dense*, pero al ser *MAERI* reconfi-

	Dense	Sparse	CSR	Reconfigurable
MAERI	✓	✗	✗	✓
TPU	✓	✗	✗	✗
SIGMA	✗	✓	✓	✓

Tabla 2: Características de las arquitecturas aceleradoras.

gurable, como se cita en el artículo de *MAERI*: "*MAERI* es más eficiente a la hora de hacer cómputo dense con respecto a la TPU" (Kwon y cols., 2018). Por eso, se ha optado por usar esta arquitectura para acelerar el cómputo dense.

Sin embargo, puesto que la *TPU* no es capaz de soportar el cómputo *sparse* (no soporta el "*sparsity*", "This unit is designed for dense matrices. Sparse ar-chitectural support was omitted. Sparsity will have high prio-rity in future designs" (Jouppi, 2017)), se ha optado por la arquitectura aceleradora *SIGMA*. Además, esta arquitectura es mucho más eficiente (da soporte al formato *CSR*, clave para el procesamiento en DLRM) y es reconfigurable.

Finalmente, a la hora de seleccionar el tipo de arquitectura a proponer hemos optado por un diseño unificado; en el caso de optar por un diseño discreto, el tiempo de conexión y la latencia con la memoria principal/cpu sería algo que no podríamos pasar por alto. Es por eso, que el diseño propuesto está en consonancia con el modelo propuesto por *Centaur* en la Sección 2.2. En la Figura 15 proponemos la arquitectura de *DLRM* a más alto nivel inspirándonos en la arquitectura propuesta por *AMD*, con su diseño *APU*, tener aceleradores integrados dentro del mismo chip de la CPU, ya que la latencia y el trasiego de los datos es considerablemente inferior.

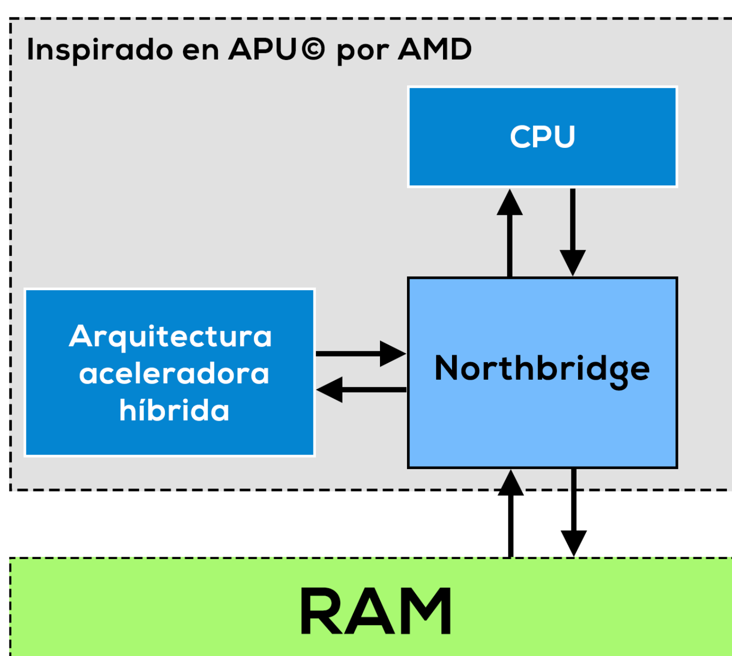

Figura 15: Arquitectura híbrida inspirada en *APU*.

3 Metodología de desarrollo del software

En el momento de desarrollar un proyecto, es necesario estructurar, planear y controlar el proceso de desarrollo, este conjunto de tareas constituyen una metodología (Peñalvo, Holgado, y Ingelmo, 2019). Antes tenemos que distinguir entre metodologías tradicionales y ágiles.

- Metodologías tradicionales. El método tradicional utiliza un enfoque lineal donde las etapas del proceso de desarrollo deben completarse en un orden secuencial. Esto significa que una etapa debe completarse antes de que comience la siguiente. Existieron diferentes metodologías como Rational Unified Process (RUP), OOram, las tarjetas CRC de Kent Beck, Métrica en España, etc.

- Metodologías ágiles. No existe como tal una metodología ágil que defina y enumere “exactamente” que hacer y cumplir para ser ágil (Garzás, 2014), en su defecto existen unos mínimos establecidos en el Manifiesto Ágil, 4 valores: individuos e interacciones sobre procesos y herramientas, software operativo sobre documentación extensiva, introducir al cliente en el proceso de desarrollo y la flexibilidad ante el cambio (Garzás, 2011).

	Ventajas	Desventajas
Tradicional	Modelo conocido y utilizado con frecuencia.	Muy difícil seguir una secuencia lineal.
	Orientado a Resultados.	Mucho tiempo para ver el producto terminado.
Ágil	Rápida respuesta a los cambios.	Falta de documentación.
	Entregas del producto a intervalos.	Soluciones erróneas en etapas largas.

Tabla 3: Metodología Tradicional vs Ágil.

Tras analizar ambas metodologías (Damorelos, 2019) se ha optado por una metodología ágil, debido al tiempo estimado para su implementación y el factor del cambio. Existen diferentes metodologías ágiles, vamos a introducirlas brevemente y escogeremos aquella que mejor se adapte.

- Programación eXtrema, *eXtreme Programming* ó XP. Consiste en llevar al límite el modelo de prototipos, haciendo entregas continuas con pequeños cambios en la funcionalidad. Una metodología ágil utilizada frecuentemente cuando los requisitos del producto final no están bien definidos y el entorno varía con cierta facilidad. Se basa en 4 valores centrales: Comunicación, Simplicidad, Feedback y Coraje. Además, se fundamenta al rededor de 12 prácticas, denominadas: "buenas prácticas", entre ellas *pair-programming*, integración continua, etc (Garzás, 2001).

- Kanban. Se traduce como "Tarjeta Visual", es una metodología que busca conseguir un proceso productivo, eficiente y organizado. Al ser un método visual permite que a golpe de vista se conozca el estado de los proyectos y asignar nuevas tareas de manera muy efectiva (APD, 2019). Kanban es una metodología flexible, podemos hacer variaciones al producto tan pronto como se detecte un error o un giro en el mercado y las tareas pueden tener una fecha límite, pero no es tan estricto el periodo determinado en un inicio para cada tarea.

- Scrum. Metodología ágil basada en: iteraciones en periodos de corta duración, llamados Sprints. Cuenta con un ciclo de vida iterativo e incremental, en el que se van realzando entregas parciales y regulares del producto final, evitando así el desarrollo de funcionalidades que sean extremadamente extensas y complejas de revisar (Agiles, 2020).

  Scrum es un proceso en el que se aplican de manera regular un conjunto de buenas prácticas para trabajar colaborativamente, en equipo, y obtener el mejor resultado posible, por esto y las ventajas que hemos comentado anteriormente, Scrum es la metodología ágil que mejor se adapta a este proyecto.

En el “annual review” de *The State of Agile* en el año 2019, se mostró como Scrum seguía liderando como la metodología ágil más usada. El 56 % de los equipos usan una Scrum, puesto que: “divide tareas complicadas en varias historias de usuario, además, les permite visualizarlas en un flujo de trabajo simplificado” (Petrova, 2019).

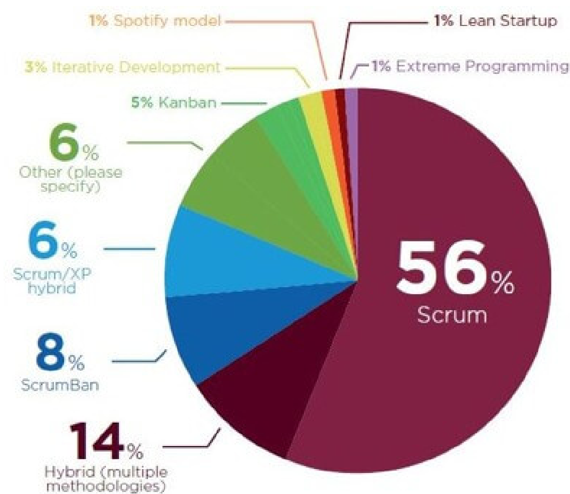

Figura 16: Most used agile metodologies *The State of Agile*, 2019 (Petrova, 2019).

Finalmente, a modo de conclusión, se ha optado por usar la metodología ágil Scrum, ya que, la programación extrema está orientada a grupos pequeños, haciendo especial atención al principio “pair-programming”, por otra parte, Kanban sería una buena elección, aunque la falta de reglas podría llegar a ser una gran desventaja. Puesto que Scrum cuenta con ventajas comunes a ambas metodologías y hace especial atención a “entorno donde se necesita obtener resultados pronto”, se ha decantado por su uso durante este proyecto.

3.1 Scrum

Según *Ken Schwaber* y *Jeff Sutherland*, creadores de “The Scrum Guide”, definen la metodología Scrum como “un marco de trabajo de *peso ligero* que ayuda a los equipos y organizaciones a generar valor a través de soluciones adaptadas a problemas complejos, permitiendo entregas de productos eficientes y con un alto valor creativo” (Schwaber y Sutherland, 2020).

3.1.1 Implementación de Scrum

Cada iteración se denomina Sprint, suele durar entre 2 y 3 semanas, como resultado del Sprint se obtiene un producto con mayor valor que el inicial, posteriormente, cada Sprint se ajustará a la funcionalidad. El framework Scrum tiene unos componentes muy bien identificados: roles, eventos y herramientas.

3.1.2 Roles en el equipo Scrum

Con la metodología Scrum, el equipo tiene como foco entregar valor y ofrecer resultados de calidad que permitan cumplir los objetivos de negocio del cliente. Para ello, los equipos de Scrum son auto-organizados y multifuncionales. En Scrum existen 3 roles muy importantes:

- Product Owner ó Propietario del Producto. Es el máximo responsable, ha de garantizar el trabajo del equipo de desarrollo, es el único perfil que habla constantemente con el cliente.
- Scrum Master. Es el responsable de que las técnicas Scrum sean comprendidas y aplicadas en la organización.
- Equipo de desarrollo. Encargados de realizar las tareas priorizadas por el Product Owner. Son los únicos que estiman las tareas del Product Backlog.

3.1.3 Eventos de cada sprint

El desarrollo iterativo se realiza en un Sprint (2 a 3 semanas), dentro de este periodo limitado de tiempo existen una serie de eventos para regular y minimizar las reuniones:

- Sprint. Es el órgano principal de Scrum, es el contenedor de los demás de hitos que se planifican durante el Sprint.
- Sprint Planning. Primer hito dentro de un Sprint, es una reunión en la que todo el equipo Scrum define qué tareas (Sprint Backlog) se van a abordar y el principal objetivo (Sprint Goal) a cumplir durante el sprint. El Sprint Planning

se realizará una vez el anterior sprint haya finalizado y se vaya a comenzar uno nuevo.

- Daily Meeting o Daily Scrum. Reuniones diarias dentro del Sprint de como máximo 15 minutos. En ella participan el equipo de desarrollo y el Scrum Master. El objetivo de esta reunión es inspeccionar el trabajo y adaptarse al cambio en caso de ser necesario, principalmente se pregunta sobre qué se hizo el día anterior.
- Sprint Review. Reunión que se realiza al final de un Sprint, en esta reunión el cliente puede asistir y en ella, el Product Owner presentará lo desarrollado al cliente, mientras que, el equipo de desarrollo se encargará de mostrar su funcionamiento.
- Sprint Retrospective. Es el último evento dentro de una iteración Sprint, es la reunión de equipo en la que se hace una evaluación de como se ha implementado la metodología Scrum en el último Sprint.

3.1.4 Herramientas de Scrum

Las herramientas que se utilizan en Scrum están definidas para maximizar la transparencia dentro del equipo.

- Product Backlog. Es el listado de tareas que engloba todo un proyecto, cualquier cosa que debamos hacer debe estar en el Product Backlog y con un tiempo estimado por el equipo de desarrollo.

  La responsabilidad de ordenar el Product backlog es del Product Owner, puesto que se encuentra en constante comunicación con el cliente para asegurarse que las prioridades están bien establecidas, por lo que las tareas que están más arriba deben de ser las de mayor prioridad.

- Sprint Backlog. Es el grupo de tareas del Product Backlog que el equipo de desarrollo elige en el Sprint Planning junto con el plan para poder desarrollarlas.

4 Tecnologías y herramientas utilizadas

4.1 Infraestructura software

- Python. Es un lenguaje de programación interpretado y multiparadigma, ya que soporta parcialmente la orientación a objetos y la programación funcional. En los últimos años el lenguaje ha incrementado notablemente su popularidad gracias al uso del mismo en campos como *Big Data*, *Data Science* o *Machine Learning*. Con respecto a *Machine Learning*, Python es el lenguaje favorito para los desarrolladores gracias a aspectos como la legibilidad de su código, facilidad de incluir nuevas librerías, la comunidad de desarrolladores y programadores, etc. Existen dos versiones utilizadas acutalmente, Python 2 y Python 3; debido a la complejidad del modelo de recomendación, puesto que usa algoritmos más modernos, para el presente trabajo se ha usado la versión de Python 3.8.0 instalada en el servidor, del cual hablaremos más adelante (Python, 2021).

- PyTorch. Es una biblioteca de aprendizaje automático, *machine learning*, de código abierto basada en la biblioteca de *Torch*, utilizado para aplicaciones que implementan modelos basados en visión aritficial, procesamiento de lenguajes naturales, imgágenes, etc. desarrollado por el laboratorio de investigacion de *IA* de *Facebook*. PyTorch tiene dos interfaces, una basada en Python y otra en C++, aunque no está tan pulida (PyTorch, 2021a). En la Sección 2.3.4 explicamos porque hemos elegido este *framework*.

- Anaconda. Es una distribucción de Python de código abierto que funciona como un gestor de entorno, un gestor de paquetes y que posee una colección de más de 720 paquetes de código abierto (Toro, 2020). Principalmente se ha usado para instalar las dependencias necesarias para ejecutar la herramienta de simulación *STONNE* en el servidor.

- STONNE. Es un simulador con precisión a nivel de ciclo y reconfigurable, que permite modelar distintos tipos de arquitecturas aceleradoras para *deep-learning*. Además, permite ser integrado dentro de un framework de

DL, como *PyTorch*, donde se puede hacer un ejecución completa *end-to-end*, de redes neuronales profundas (Martínez y cols., 2020).

- GitHub. Es una forja (plataforma de diseño colaborativo) para alojar proyectos utilizando el sistema de control de versiones Git. Para llevar un control del modelo de recomendación y su implementación en STONNE se ha utilizado GitHub con el fin de controlar los posibles errores que se puedan generar (NextU, 2020).

- MobaXterm. Es una terminal mejorada para Windows destinada a realizar tareas de administración en ordenadores y servidores remotos. MobaXterm permite sesiones como SSH, Telnet, FTP, SFTP, Shell, entre muchas otras. Se ha usado durante el presente trabajo para conectarse de manera remota al servidor de la Universidad, donde se encuentra alojado el proyecto (RZone, 2020).

- Slack. Es una plataforma de mensajes basada en canales. Los usuarios pueden crear múltiples servidores e invitar a diferentes usuarios. Slack se ha usado durante el presente trabajo como herramienta de comunicación entre los integrantes del equipo de investigación, para poder así planear reuniones, comentar problemas que aparezcan o dejar plasmadas ideas futuras (Slack, 2021).

- Wrike. Es un software de gestión de proyectos versátil y eficaz para metodologías ágiles, permite crear tableros para planificar las historias de usuario y es capaz de generar listas/diagramas de gantt a partir de las tareas creadas (Wrike, 2021).

- GNU Screen. Es un programa informático de Multiplexación de terminales, permite a los usuarios acceder a múltiples sesiones separadas dentro de una sola ventana de terminal o en una sesión de terminal remota. Puesto que las simulaciones que hemos realizado en el presente trabajo se asemejan con cargas reales, el tiempo de cómputo puede tomar múltiples horas fácilmente, es necesario el uso de Screen para que se sigan ejecutando

aún cuando no estemos (GNU, 2016).

- Overleaf. El presente documento se ha escrito usando LaTeX mediante la plataforma software Overleaf, que permite compilar el código *.tex* a formato PDF (*pdfTeX*). Es un software colaborativo mediante el cual varios colaboradores pueden revisar/editar un mismo documento simultáneamente.

4.2 Infraestructura hardware

Todo el desarrollo del presente trabajo se ha realizado a través de un servidor remoto de la Universidad Católica de San Antonio de Murcia, tanto para el desarrollo, la implementación y la caracterización del modelo *DLRM* sobre la herramienta de simulación *STONNE*.

- Nombre SSH: *boston.ucam.edu*
- CPU: Intel Xeon E5-2640 v4
- Memoria RAM: 125Gb
- Memoria Swap: 3,7 Gb

5 Estimación de recursos y planificación

Planificar es construir una secuencia de tareas con la lógica necesaria para alcanzar el objetivo del proyecto. Como vimos en la asignatura gestión de proyectos informáticos, “El trabajo se expande hasta agotar el tiempo disponible”. Por eso es necesario realizar una estimación de los recursos necesarios y planificar que vamos a hacer durante los meses de trabajo. “La planificación es el todo y la programación sería el núcleo del proyecto”.

5.1 Preparación preliminar

Antes de comenzar con el desarrollo del proyecto, es decir, con los *Sprints*, es necesario planear que vamos a hacer, esta fase es conocida como “Sprint 0”. En verdad, este primer sprint no es un artefacto de *Scrum*, el objetivo de este *Sprint* inicial es construir una parte de la arquitectura ágil, para que los futuros *Sprints* puedan añadir valor de forma eficiente al producto (Menzinsky, 2015).

Dentro de este *Sprint* inicial, se realiza una reunión con las personas que conforman el proyecto para formar el *Scrum Team* y asignar los roles (los roles del equipo *Scrum* se vieron en la Sección 3.1.2). Como se trata de un TFG, el equipo estará compuesto por el tutor y por el alumno que lo presenta. Los roles del equipo *Scrum* son los siguientes:

- Product Owner: D. José Luis Abellán Miguel.
- Scrum Master: D. Nicolás Meseguer Iborra.
- Equipo de desarrollo: D. Nicolás Meseguer Iborra.

En el *Product Backlog* (Sección 3.1.4) se encuentra el listado de tareas que engloba todo el proyecto. Por eso es necesario una lista ordenada de los requisitos del producto y fijar un formato determinado. En este proyecto usaremos historias de usuario; una explicación general e informal de una función de software escrita desde la perspectiva del usuario final (Rehkopf, 2021). Cada historia de usuario puede verse como una tarea general que se desglosa en una lista de pequeñas

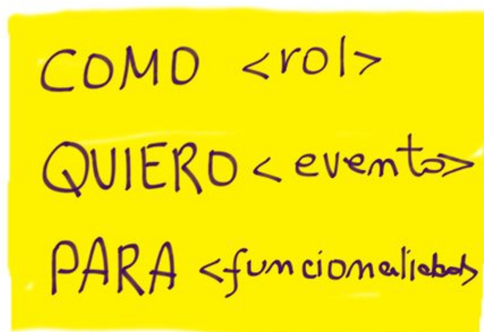

Figura 17: Historias de usuario (Martin, 2018).

tareas más específicas. En la Figura 17 se muestra la estructura típica de una historia de usuario.

Una vez elaborada la lista de historias de usuario, se pueden realizar estimaciones para medir el esfuerzo o el tamaño de los objetivos a implementar. Usaremos una técnica conocida como *Planning Poker*, técnica colaborativa que permite la participación de todos los miembros del equipo *Scrum* para calcular el esfuerzo necesario. Asociando a cada historia de usuario una valoración numérica. Estas puntuaciones estimadas pueden verse como puntos de historia, una medida que indica la complejidad relativa.

En la Figura 18 se muestra una baraja de cartas, cada valor puede verse como la dificultad de una tarea, teniendo las tareas más complejas puntuaciones más altas y las más fáciles obteniendo valores numéricos inferiores. Sigue parcialmente la sucesión de *Fibonacci* y las tres últimas cartas hacen referencia a: “esfuerzo incalculable”, “desconocimiento”, “taza de café”, respectivamente. Usaremos este *deck* para estimar los puntos de historia.

Partimos de una lista de historias de usuario iniciales y definidas para comen-

Figura 18: Estimación de póquer (Manager, 2021).

zar el desarrollo del proyecto. En la Tabla 4 se muestra las historias de usuario iniciales, junto con sus puntos de historia (P.H.). En un primer momento, los elementos del *Sprint Backlog* serán todas las historias de usuario que aparecen, ya que todas son igual de importantes y siguiendo el mismo orden en el se presentan.

ID	Historia de usuario	ID	Tarea	P.H.
01	Como: desarrollador Quiero: estudiar el campo del *DL* Para: desarrollar el proyecto	01.01	Leer Artículo *DLRM*	6
		01.02	Entender funcionamiento del algoritmo *EmbeddingBag*	15
		01.03	Entender funcionamiento de las *MLPs*	10
		01.04	Entender el funcionamiento de las *Interact Features*	10
		01.05	Leer artículo *STONNE*	9
		01.06	Estudio de aceleradores de inferencia para *deep-learning*	10
02	Como: desarrollador Quiero: instalar *framework* de desarrollo Para: codificar el proyecto	02.01	Instalar localmente Python	3
		02.02	Instalar localmente Anaconda + *PyTorch*	2
		02.03	Ejecutar y comprobar el modelo DLRM nativo	5
		02.04	Instalar + Compilar las librerias de STONNE	5
		02.05	Simular ejecución test en STONNE	4
03	Como: desarrollador Quiero: codificar el *benchmark* de simulación Para: conocer el funcionamiento	03.01	Preparar los datos de entrada	4
		03.02	Codificar algoritmo CSR	2
		03.03	Simular ejecución de EmbeddingBag	10
		03.04	Evaluar el paso *Interact Features*	3
		03.05	Validación del *benchmark* vs simulación nativa PyTorch	4
04	Como: desarrollador Quiero: integrar *DLRM* en *STONNE* Para: proponer Arq. híbrida	04.01	Implementar algoritmo *EmbeddingBag* en *STONNE*	10
		04.02	Preparar los datos de entrada para *MLPs* y *EmbeddingBags*	5
		04.03	Conectar *DLRM* con *STONNE* mediante su *API*	10
		04.04	Configurar aceleradores de inferencia	5
		04.05	Comprobar resultados de ejecución	6
05	Como: desarrollador Quiero: preparar casos de prueba Para: caracterizar el trabajo	05.01	Investigar como realizar cargas de trabajo personalizadas	7
		05.02	Investigar cargas de trabajo reales/sintéticas	5
		05.03	Realizar ejecución de pruebas en *STONNE*	15
		05.04	Valorar y analizar el resultado de las pruebas	10

Tabla 4: Product Backlog: Historias de usuario (Wrike, 2021).

El número total de P.H. estimados ha sido de 175. Para la gestión del proyecto se ha usado la herramienta software *Wrike* (ver Sección 4.1). A continuación (Figura 19), se muestra el tablero inicial con las historias de usuario planificadas para su desarrollo.

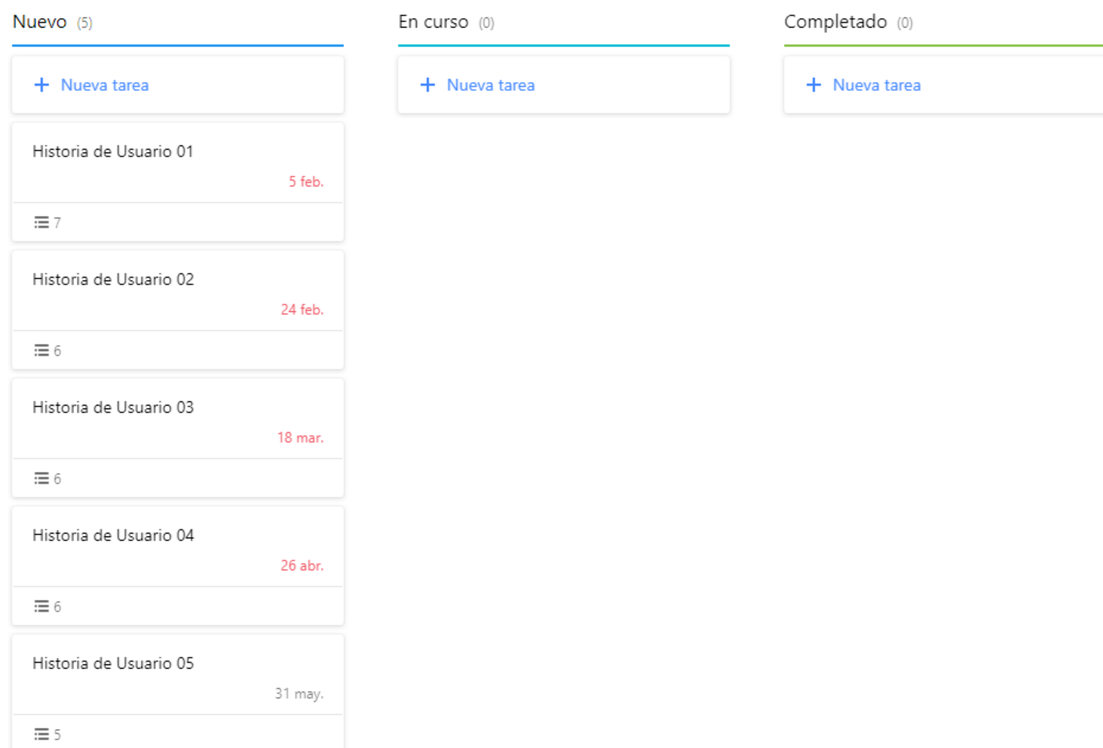

Figura 19: Wrike: Product Backlog.

5.2 Estimación por puntos de historia

Inicialmente, cuando se realizó la primera reunión con todos los miembros del *Scrum Team*, a mediados de octubre, se definió que la duración del proyecto sería de 7 meses, desde noviembre hasta mayo, ambos inclusive. La primera parte del proyecto, es decir, los meses correspondientes al primer cuatrimestre estarían dedicados a la investigación sobre el campo del *DL*, ya que la carga de estudios era significativamente importante, mientras que, a partir de febrero se comenzaría con la implementación y desarrollo íntegro del proyecto.

Estableciendo dos dedicaciones diferentes al proyecto, una correspondiente al primer cuatrimestre, ecuación 1, y una segunda correspondiente al segundo cuatrimestre, ecuación 2, se pueden calcular las horas estimadas para la realización del proyecto. En ambas estimaciones, se asume que cada mes está compuesto por cuatro semanas (aproximadamente) y se sigue un estilo de trabajo a media jornada (lunes-viernes). Para la primera estimación se asume una dedica-

ción de 1/2 de media jornada (2 horas) y para la segunda una jornada media (4 horas).

$$Duracion(h) = \left(3_{meses} x \frac{4_{semanas}}{1_{mes}} x \frac{5_{dias}}{1_{semana}} x \frac{2_{horas}}{1_{dia}}\right) = 120_{horas} \quad (1)$$

$$Duracion(h) = \left(4_{meses} x \frac{4_{semanas}}{1_{mes}} x \frac{5_{dias}}{1_{semana}} x \frac{4_{horas}}{1_{dia}}\right) = 320_{horas} \quad (2)$$

Entre ambas estimaciones se calcula una duración estimada de 450 horas de trabajo. Una vez hemos obtenido la duración estimada del proyecto y los P.H. (puntos de historia), podemos calcular la velocidad de trabajo estimada en P.H./h (puntos de historia por hora). Desafortunadamente, esta métrica no puede ser considerada como medida final, ya que genera cierta incertidumbre con el paso del tiempo (no es lineal). Por ello, se debe considerar un cierto margen, optimista y pesimista. La ecuación 3 muestra la velocidad de trabajo estimada del proyecto, siendo (*a*) la versión optimista (*+20 % velocidad*), (*b*) la neutral y (*c*) la versión pesimista (*-20 % velocidad*).

$$Velocidad(PH/h) = \begin{cases} (a) \equiv (b) + ((b)\ x\ 0,2) \approx 0,47_{PH/h} \\ (b) \equiv \frac{175_{PH}}{450_{h}} \approx 0,39_{PH/h} \\ (c) \equiv (b) - ((b)\ x\ 0,2) \approx 0,31_{PH/h} \end{cases} \quad (3)$$

Además, con la duración estimada del proyecto (ecuaciones 1 y 2) se puede calcular la duración estimada del proyecto desde tres puntos de vista, uno optimista, neutral y pesimista. La ecuación 4 muestra la duración estimada a partir

de las velocidades calculadas.

$$Duracion(h) = \begin{cases} (a) \equiv \frac{175_{PH}}{0{,}47_{PH/h}} \approx 372_h \\ (b) \equiv 450_h \\ (c) \equiv \frac{175_{PH}}{0{,}31_{PH/h}} \approx 565_h \end{cases} \tag{4}$$

Suponiendo que la duración media de un Sprint es de un mes y medio, es decir, 4 semanas más 2 semanas, y a partir de las duraciones estimadas del proyecto, es posible determinar el número de Sprints necesarios para desarrollar el proyecto. En la ecuación 5 se calculan el número de Sprints estimados, de acuerdo a tres visiones, optimista, neutral y pesimista. Atención, para estimar las horas que se van a trabajar diariamente, puesto que son diferentes en cada cuatrimestre, se ha realizado la media entre ambas, dando un resultado de 3 horas diarias.

$$Sprints = \begin{cases} (a) \equiv \frac{372_h}{1_{mes} x \frac{6_{semanas}}{3/2_{mes}} x \frac{5_{dias}}{1_{semana}} x \frac{3_h}{1_{dia}}} \approx 4_{sprints} \\ (b) \equiv \frac{450_h}{1_{mes} x \frac{6_{semanas}}{3/2_{mes}} x \frac{5_{dias}}{1_{semana}} x \frac{3_h}{1_{dia}}} \approx 5_{sprints} \\ (c) \equiv \frac{565_h}{1_{mes} x \frac{6_{semanas}}{3/2_{mes}} x \frac{5_{dias}}{1_{semana}} x \frac{3_h}{1_{dia}}} \approx 6_{sprints} \end{cases} \tag{5}$$

Con el número de *Sprints* estimados, podemos estimar los P.H. por Sprint y la duración de cada Sprint en h, al igual que hasta ahora, realizaremos tres estimaciones, una optimista, una neutral y otra pesimista. Hacemos referencia a las ecuaciones 6 y 7.

$$PH/sprint = \begin{cases} (a) \equiv \frac{PH_{totales}}{Sprints_a} = \frac{175_{PH}}{4_{sprints}} \approx 44_{PH/sprint} \\ (b) \equiv \frac{PH_{totales}}{Sprints_b} = \frac{175_{PH}}{5_{sprints}} \approx 35_{PH/sprint} \\ (c) \equiv \frac{PH_{totales}}{Sprints_c} = \frac{175_{PH}}{6_{sprints}} \approx 29_{PH/sprint} \end{cases} \tag{6}$$

$$h/sprint = \begin{cases} (a) \equiv \frac{Duracion_a}{Sprints_a} & = \frac{372_h}{4_{sprints}} \approx 93_{h/sprint} \\ (b) \equiv \frac{Duracion_b}{Sprints_b} & = \frac{450_h}{5_{sprints}} \approx 90_{h/sprint} \\ (c) \equiv \frac{Duracion_c}{Sprints_c} & = \frac{565_h}{6_{sprints}} \approx 94_{h/sprint} \end{cases} \quad (7)$$

También es posible calcular la duración estimada por cada punto de historia (P.H.), como se observa en la ecuación 8.

$$h/ph = \begin{cases} (a) \equiv \frac{93_{h/sprint}}{44_{PH/sprint}} \approx 2,11_{h/PH} \\ (b) \equiv \frac{90_{h/sprint}}{35_{PH/sprint}} \approx 2,57_{h/PH} \\ (c) \equiv \frac{94_{h/sprint}}{29_{PH/sprint}} \approx 3,24_{h/PH} \end{cases} \quad (8)$$

Finalmente, en la Tabla 5 se muestran todas las estimaciones que se han realizado para el proyecto.

Punto de vista	Duración (h)	Velocidad (PH/h)	Sprints	Por sprint		Por P.H.
				PH	Duración (h)	Duración (h)
Optimista (+20 %)	372	0,47	4	44	93	2,11
Neutral	450	0,39	5	35	90	2,57
Pesimista (-20 %)	565	0,31	6	29	94	3,24

Tabla 5: Estimaciones realizadas para el proyecto.

5.3 Planificación temporal del proyecto

Planificar es construir una secuencia de tareas con la lógica necesaria para alcanzar el objetivo del proyecto en plazo óptimo, coste y calidad. La finalidad de los diagramas que se presentan a continuación es identificar aquellas acciones

que puedan hacerse de forma simultánea y cuales requieren que otra haya sido realizada con anterioridad.

Entre los objetivos de la planificación existen algunos aspectos clave como: establecimiento y cumplimiento de metas, reducción de la incertidumbre, detección de problemas potenciales, etc.

A partir de la ecuación 8 podemos calcular las horas estimadas para cada tarea de la Tabla 4 historias de usuario. Seguiremos el mismo método que hasta ahora, realizaremos tres estimaciones, una optimista, neutral y pesimista. En la Tabla 6 se muestra la duración para cada punto de vista de las historias de usuario.

Para la planificación temporal de este proyecto usaremos el diagrama de Gantt, considerando las horas estimadas desde un punto de vista neutral. El diagrama de Gantt es un diagrama de barras que muestra las actividades (filas) y su duración (barras) en una escala de tiempo (columnas). Muestra fechas de inicio y fin de actividades. Entre las actividades se pueden añadir dependencias, es decir, una actividad ha de finalizar para que otra se pueda iniciar.

Antes de introducir el diagrama de Gantt es necesario que hagamos una planificación temporal de las actividades que se van a llegar a cabo durante el proyecto (Figura 20. Se recuerda que durante el primer cuatrimestre se ha estimado un trabajo diario de 2 horas al día, mientras que para el segundo cuatrimestre, unas 4 horas al día (media jornada).

Como se observa en la planificación temporal de la Figura 20, durante los primeros meses del proyecto se llevó un ritmo constante de aproximadamente 2 horas al día, puesto que se estaba desarrollando a la par que cursando el primer cuatrimestre. Cuando llegamos a la treceava semana se observa un incremento notable en las horas de trabajo, ya que esta semana corresponde con el comienzo del mes de febrero, cuando ya habíamos terminado los exámenes y es por eso que a partir de este momento estimamos un trabajo diario aproximado de 4 horas. También, si observamos en las últimas semanas, la tarea 05.04 se alarga, ya que se trata de razonar el resultado de las pruebas y hasta el último día se van realizando pequeños ajustes/cambios al documento. Estas últimas semanas

ID	P.H.	Duración (h)		
		Optimista	Neutral	Pesimista
01.01	6	12,7	15,4	19,4
01.02	15	31,7	38,6	48,6
01.03	10	21,1	25,7	32,4
01.04	10	21,1	25,7	32,4
01.05	9	19	23,13	29,2
01.06	10	21,1	25,7	32,4
02.01	3	6,3	7,7	9,7
02.02	2	4,2	5,1	6,5
02.03	5	10,6	12,9	16,2
02.04	5	10,6	12,9	16,2
02.05	4	8,44	10,3	13
03.01	4	8,44	10,3	13
03.02	2	4,2	5,1	6,5
03.03	10	21,1	25,7	32,4
03.04	3	6,3	7,7	9,7
03.05	4	8,44	10,3	13
04.01	10	21,1	25,7	32,4
04.02	5	10,6	12,9	16,2
04.03	10	21,1	25,7	32,4
04.04	5	10,6	12,9	16,2
04.05	6	12,7	15,4	19,4
05.01	7	14,8	18	22,7
05.02	5	10,6	12,9	16,2
05.03	15	31,7	38,6	48,6
05.04	10	21,1	25,7	32,4
Total	175	369.3	449,8	567

Tabla 6: Estimaciones realizadas para el proyecto.

sirven para perfeccionar el trabajo.

Una vez realizada la planificación temporal, en la Figura 21 mostramos el diagrama de Gantt resultante.

5.4 Valoración de la dedicación y coste económico

Estimamos que el salario anual de un desarrollador de software en España es de 19.000,00 € anuales (Carrasco, 2020). Vamos a realizar la valoración del coste económico considerando un único programador dentro del equipo de desarrollo.

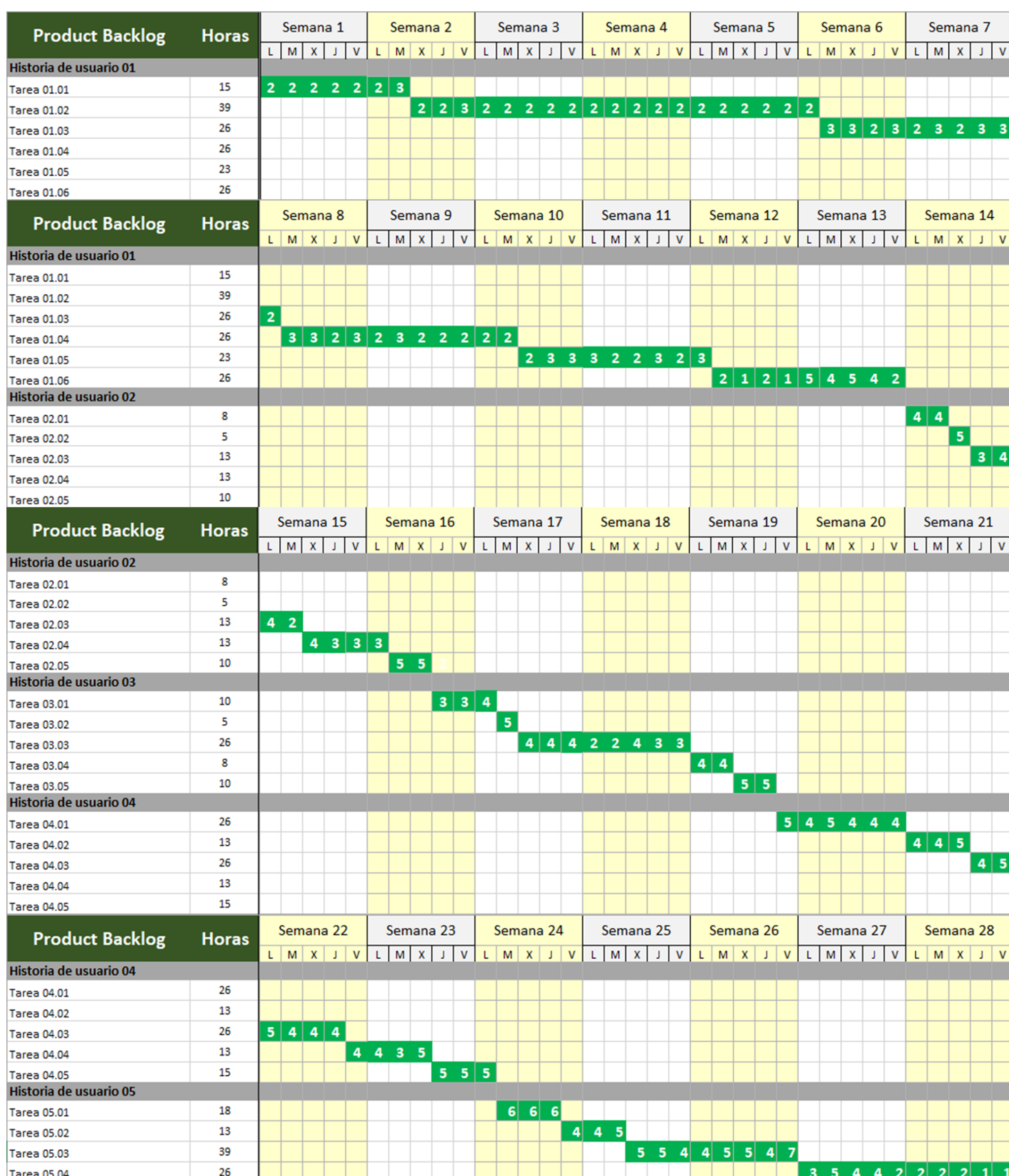

Figura 20: Planificación temporal.

Si nos fijamos en el diagrama de Gantt y en las ecuaciones del punto anterior, establecemos un total de 5 Sprints (punto de vista neutral) para el desarrollo del proyecto a media jornada, ya que el desarrollo del proyecto ha de compaginarse con las asignaturas. Por lo que reducimos el salario establecido a la mitad. A continuación, se muestra el salario de 7 meses a media jornada, asumiendo 1 paga extra, donde *S*, es el salario; *JC*, jornada completa y *MJ*, media jornada:

$$S(€) = (7_{meses} + 1_{paga\ extra})\, x \left(\frac{19{,}000€}{14_{pagas}} x \frac{1_{JC}}{2_{MJ}} \right) \approx 5{,}428,57€$$

Además de la remuneración salarial del equipo de desarrollo, se debe tener

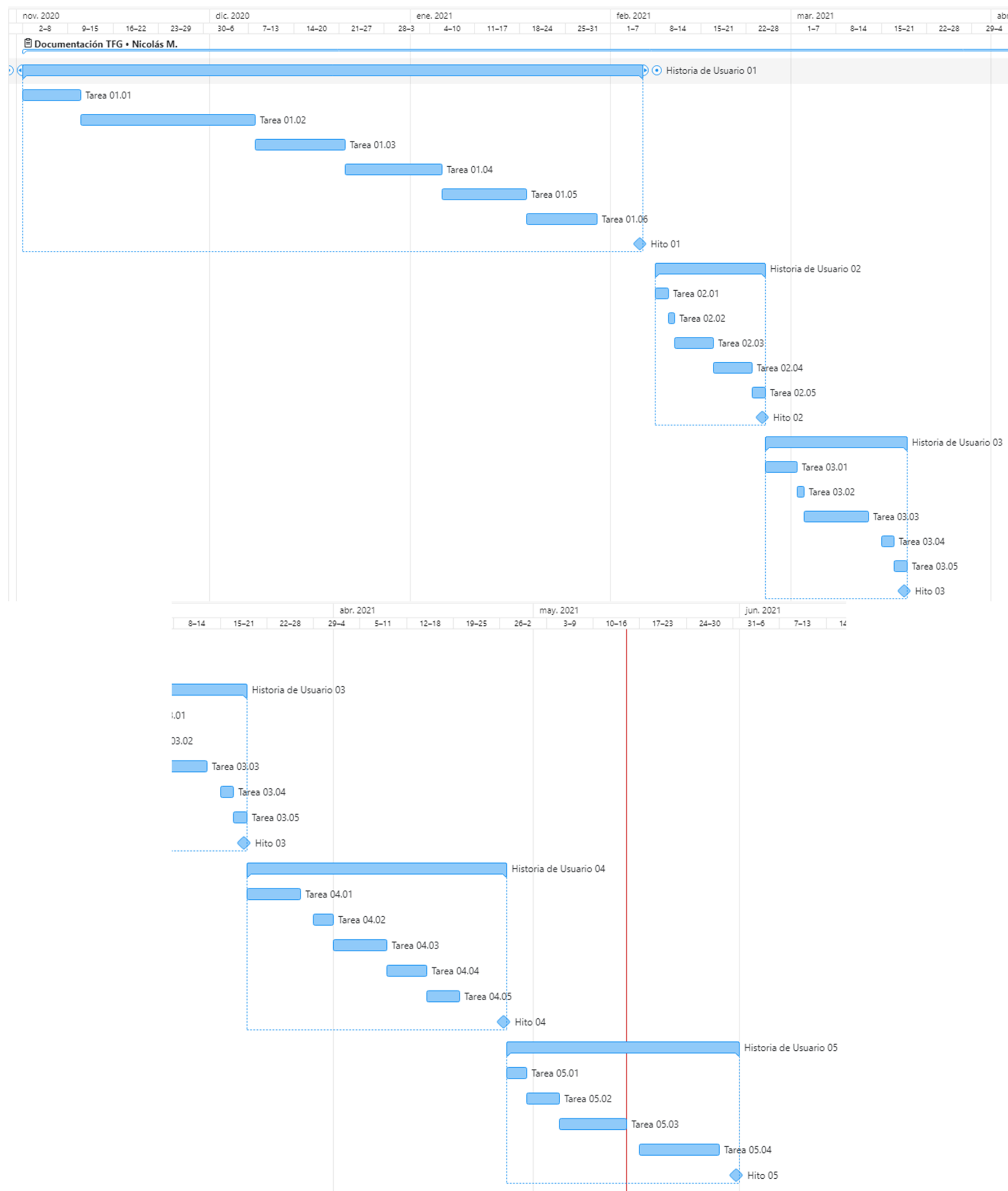

Figura 21: Diagrama de Gantt.

en cuenta la del Scrum Master, viendo un informe de *indeed* sobre el salario, en 2021 se establece en 36.090 € anuales para una jornada completa (Indeed, 2021). Asumiendo una jornada media, al igual que con el equipo de desarrollo, y 7 meses de desarrollo + 1 paga extra, aplicamos nuevamente la fórmula para obtener el salario del Scrum Master *SSM*:

$$S(€) = (7_{meses} + 1_{paga\ extra})\ x \left(\frac{36{,}090€}{14_{pagas}} x \frac{1_{JC}}{2_{MJ}} \right) \approx 10{,}311,42€$$

Finalmente, en la Tabla 7 se puede observar en relación al desarrollo del pro-

yecto el coste total de los salarios. Además, hay que tener en cuenta la necesidad de contratar los servicios básicos, entre los que figuran un proveedor de internet (ISP) que calcularemos unos 44,99 €/mes; Gastos de luz y agua, entre ambos sumando 80€/mes y un portátil que usará el desarrollador para realizar la integración. La duración del contrato de los servicios será de 7 meses, el plazo estimado para completar los Sprints.

Recurso	**Coste (€)**
Salario Desarrollador	5,428.57
Salario Scrum Master	10,311.42
ISP	49,99x7
Luz y Agua	80x7
Portátil	1.000
Subtotal	17.649,99
I.V.A. 21 %	3.706,50
Total	21.356,49

Tabla 7: Estimación económica del proyecto.

Del coste estimado, una parte serán los beneficios y otra los costes del proyecto. A continuación introducimos el ROI, de sus siglas en inglés *Retorno sobre la inversión*, métrica usada para saber por cada euro, cuanto beneficio generó la empresa (Custódio, 2018).

Aplicando la fórmula del ROI, donde los beneficios aparecen como *B* y los costes como *C*, obtenemos el siguiente resultado:

$$ROI = \frac{B - C}{C}$$

$$= \frac{(5{,}428,57 + 10{,}311,42)_{salarios} - (349,96_{ISP} + 560_{LuzyAgua} + 1{,}000_{Porttil})}{(349,96_{ISP} + 560_{LuzyAgua} + 1{,}000_{Porttil})} \approx 7,27$$

Los salarios forman parte de los beneficios y los costes asociados como parte de los costes *C*, dentro de la ecuación del ROI. El valor del ROI, como se observa, es un numero positivo (7,27), confirmando así la viabilidad del proyecto.

6 Desarrollo del contenido del proyecto

Para el desarrollo del proyecto se ha seguido la metodología Scrum, como ya se ha indicado en la Sección 3. Según las estimaciones realizadas en la Sección 5 hemos dividido las Historias de Usuario en 4 Sprints diferentes. En cada uno de estos Sprints se desarrollaran los eventos propios de la metodología Scrum ya vistos en la Sección 3.1.3. Por simplicidad hemos descartado el incluir los *Daily* Scrums ya que, a pesar de haberlos realizado, eran reuniones cortas e informativas, no van a quedar representados en este apartado de la documentación.

6.1 Sprint 1

En este primer Sprint se realizará una toma de contacto con el mundo de la investigación. Se analizarán artículos clave dentro del campo de los sistemas de recomendación y el *deep-learning*, como el modelo *DLRM*, el simulador *STONNE*, etc. Entender el funcionamiento de los algoritmos del modelo de recomendación es un aspecto crucial para su posterior desarrollo e implementación en STONNE.

6.1.1 Sprint planning

En la Figura 22 se muestra la Historia de Usuario seleccionada, junto con las sub-tareas, que se desarrollarán durante este primer Sprint.

El Sprint Goal de esta primera fase consiste en adquirir conocimiento relativo al campo en el que se va a realizar el proyecto (modelos DL, sistemas de recomendación, técnicas relacionadas, etc.). Entender el funcionamiento de los algoritmos que usan los sistemas de recomendación y comprender *STONNE*.

Hay que entender perfectamente como el sistema de recomendación propuesto por *Facebook* funciona. Conocer su arquitectura para así identificar donde se encuentra el cómputo del modelo debería de ser uno de los objetivos principales dentro de este primer Sprint. Una vez identificada su arquitectura, hay que saber diferenciar los componentes que la conforman.

Como tareas derivadas de esta 01.01 se encuentran las tareas 01.02, 01.03 y 01.04. Una vez se tienen identificados los módulos que forman la arquitectura

Historia de Usuario 01

Documentación TFG +

Nuevo | + Añadir asignado | #690830457 por Nicolás M. a las 15:17

2 nov. 2020 – 5 feb. 2021 (70d) | 7 subtareas | Adjuntar archivos | Añadir dependencia | 2

Tarea	Fecha	Estado
Hito 01	5 feb.	Nuevo
Tarea 01.06 | Estudio sobre aceleradores de inferencia	29 ene.	Nuevo
Tarea 01.05 | Leer artículo STONNE	18 ene.	Nuevo
Tarea 01.04 | Entender funcionamiento "Interact Features"	5 ene.	Nuevo
Tarea 01.03 | Entender funcionamiento MLPs	21 dic. 2020	Nuevo
Tarea 01.02 | Entender funcionamiento Embedding	7 dic. 2020	Nuevo
Tarea 01.01 | Leer artículo DLRM	10 nov. 2020	Nuevo

+ Nueva tarea

Como: Desarrollador
Quiero: Estudiar el campo del *deep-learning*
Para: Desarrollar el proyecto

Figura 22: Historias de Usuario para el Sprint 1.

DLRM es necesario entender como el trasiego de datos fluye por estos componentes. ¿Como entran los datos? ¿Qué les ocurre en cada fase? ¿La salida de una fase se interconecta con la entrada de otra?

Una vez el modelo *DLRM* se ha entendido, es necesario estudiar el campo de los simuladores para el *deep-learning*, necesitamos un simulador que permita hacer ejecuciones completas (*end-to-end*) y ofrezca precisión a nivel de ciclo. Este estudio no es relativo a entender el funcionamiento de *STONNE* al completo, sino más bien, entender porque es necesario un simulador y que función cumple dentro del proyecto.

Finalmente, una vez leído el artículo del simulador *STONNE*, damos por hecho que hemos adquirido conocimiento a cerca de este y comprendemos que *STONNE* permite simular arquitecturas aceleradoras, entre las que se encuentran: *MAERI*, *SIGMA* y la *TPU*. En la tarea 01.05 tendremos que realizar un último estudio a cerca de las arquitecturas aceleradoras para elegir las que acelerarán

el cómputo del modelo *DLRM*.

En la Figura 23 se muestra el tablero de *Wrike* al comienzo del sprint.

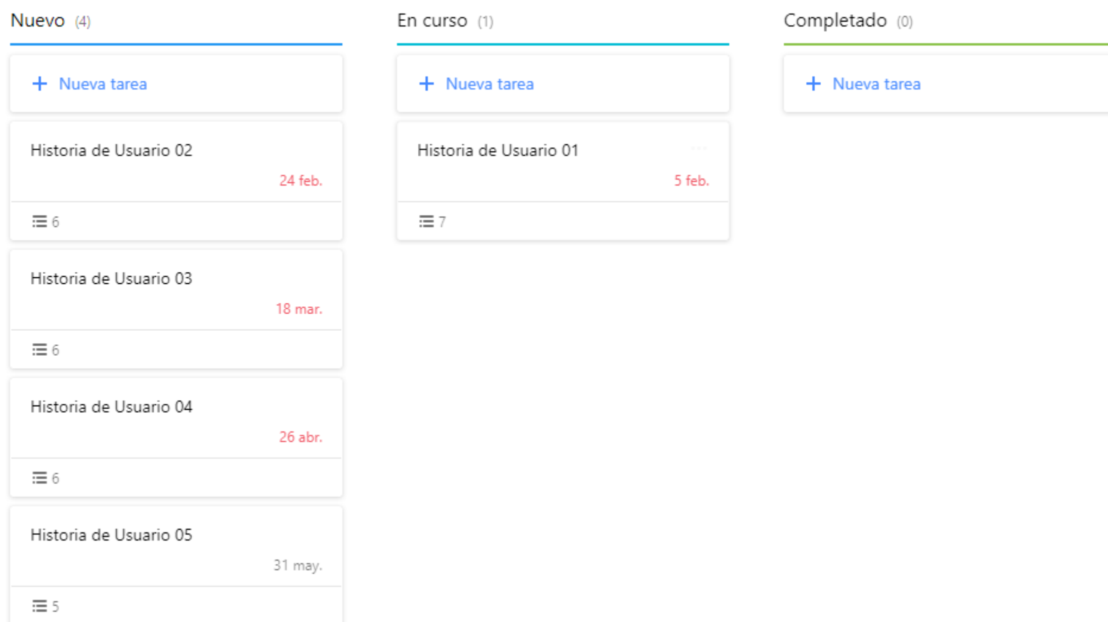

Figura 23: Tablero para el Sprint 1.

6.1.2 Sprint review

Esta primera Historia de Usuario, como se observa, tiene un carácter meramente investigador y con la finalidad de adquirir conocimiento sobre el mundo de los sistemas de recomendación dentro del *deep-learning*. En la primera tarea, 01.01 el objetivo es leer el artículo del sistema de recomendación *DLRM* propuesto por *Facebook* para entender como los sistemas de recomendación funcionan, que técnicas usan y como se sitúan hoy en día en el mercado. Lo que al final queremos lograr con este punto es entender el modelo de recomendación e identificar los componentes que lo forman, esta explicación ya se encuentra detallada en la Sección 2.1. De igual manera, las tareas 01.02 y 01.03, son una gran parte del estado del arte del problema, cuyo estudio, de igual manera que la introducción al modelo *DLRM*, ya ha sido explicado en la Sección 2.1.

A pesar de esto, existen algunos detalles que se han de tener muy en cuenta para comprender de manera correcta el resto del trabajo que presentamos. Una parte es entender muy bien como funciona el algoritmo *EmbeddingBag*, aunque

ya se encuentra explicado en la Sección 2.1, vamos a indagar un poco más en las operaciones que se realizan y presentaremos una imagen para facilitar su comprensión.

El computo *sparse* sucede en las *EmbeddingBags*, cada *EmbeddingBag* tiene una matriz de pesos asociada (supondremos que los pesos ya se encuetran entrenados, nos encontramos en la fase de inferencia). Por cada *EmbeddingBag* se haran X*lookups*, estas *lookups* consisten en seleccionar Y filas de la matriz de pesos (el número de *lookups* X y los índices seleccionados Y varían en función de dos variables que veremos más adelante). Este proceso de *lookup* selecciona filas de la matriz de pesos usando el formato CSR. Aquellos valores no-nulos son las filas seleccionadas de la matriz de pesos. En la Figura 7 se observa como se produce la conversión CSR-multi-hot encoding.

Una vez el vector multi-hot encoding se ha confeccionado, realizaremos una multiplicación matricial (*GEneral Matrix to Matrix Multiplication*, *GEMM*), conocida como *GEMM sparse-dense*, vector multi-hot encoding (sparse) por la matriz de pesos (dense) de la *EmbeddingBag*. En esta parte se observa un déficit en el rendimiento del modelo, es por eso necesario una arquitectura que acelere esta parte del cómputo, esto ha servido como motivación para investigar arquitecturas aceleradoras.

Los valores no-nulos de este vector multi-hot tienen el valor 1, esto significa que las operaciones que realizamos al final son la suma de aquellas columnas cuyos índices han sido seleccionados por el vector sparse. Como esto es difícil de comprender a priori, en la Figura 24 adjuntamos un ejemplo de esta operación que hemos comentado.

Cada *EmbeddingBag* devolverá un vector resultante (multiplicación vector-matriz, una única fila y J columnas). Una vez todas las *EmbeddingBags* han realizado sus *lookups*, la salida de todas se concatena formando una matriz de tamaño MxJ, esta matriz es de tipo *dense*. Donde M es el número de *EmbeddingBags* (este concepto no es importante tenerlo pero es un detalle a tener en cuenta para entender la salida de las *EmbeddingBags*).

Como se observa en la Figura 24, el vector multi-hot encoding (sparse) se

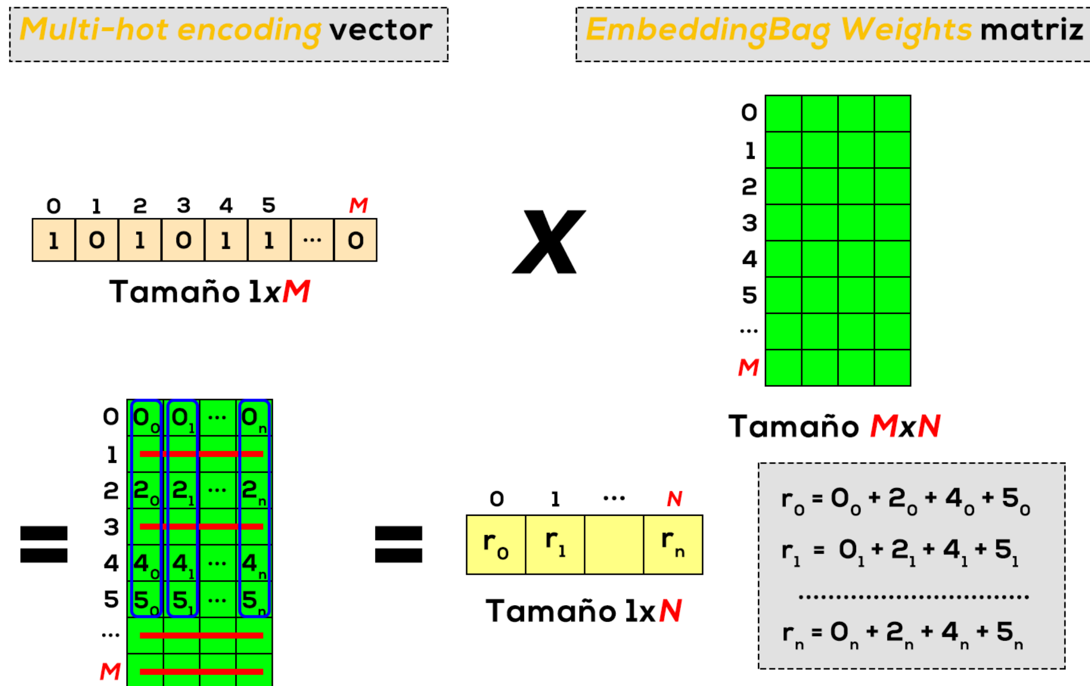

Figura 24: Ejemplo de una operación *GEMM Sparse*.

encuentra compuesto de 0's mayoritariamente. Aquellos índices cuyo valor no es 0, en su lugar 1, indica las filas seleccionadas de la matriz de pesos. En este caso, el vector sparse ha seleccionado las filas 0, 2, 4 y 5. Como se observa, el resultado de esa operación sparse-dense es la suma vectorial de los índices seleccionados (se recomienda encarecidamente realizar una operación a papel si no se entiende esta parte).

El cómputo dense que ocurre dentro de las *MLPs* (*Bottom* y *Top*) es una operación de matrices GEMM dense únicamente. La salida de este componente es un vector dense del mismo tamaño (mismo numero de columnas, J) que la matriz de salida de las *EmbeddingBags*.

En la fase denominada *Feature-Interaction*, se producen dos operaciones muy características. Por una parte la salida de la *Bottom-MLP*, un vector *dense*, se concatena con la salida de las *EmbeddingBags*, una matriz dense; es muy importante remarcar un aspecto, el vector dense ha de tener la misma dimensión (índices) que columnas tenga la matriz resultante con el fin de concatenarse (esto implicará un mayor o menor cómputo en la *Bottom-MLP*), tamaño J, siguiendo

la nomenclatura. A continuación, esta matriz se multiplica por su traspuesta. Finalmente, se concatena una vez más la salida de la *Bottom-MLP* con la matriz resultante de la operación anterior. Esta es una técnica denominada *Matrix Factorization*, haciendo referencia a la operación de matrices que se realiza.

Existe una peculiaridad que es muy importante destacar de cara a la evaluación que realizaremos en el sprint 4. En la *Top-MLP* además de las capas que especificaremos cuando confeccionemos los experimentos, el modelo DLRM realiza un cálculo interno; en base al número de *EmbeddingBags* y a la última capa de la *Bottom-MLP*, el modelo añadirá una capa extra con un determinado número de neuronas a la *Top-MLP*. Esta capa extra se ejecutará previamente al resto de las capas que nosotros le indiquemos. Más detalles sobre este aspecto se detallarán en las conclusiones.

Respecto a las últimas tareas dentro de este primer Sprint, 01.04 y 01.05, donde el objetivo principal, una vez más, es entender porque usamos *STONNE*, que función cumple, porque necesitamos este simulador de arquitecturas aceleradoras, etc. La explicación, al igual que con las tareas anteriores se encuentra detallada en la Sección 2, donde a parte de explicar STONNE, ponemos de manifiesto las diferencias que existen entre las diferentes arquitecturas aceleradoras (Sección 2.3.4).

En Sprints posteriores (ver Sección 6.3.2) explicaremos como se han configurado las arquitecturas aceleradores para inferencia que hemos usado (*MAERI* y *SIGMA*) así como alguno de sus parámetros, configuración, etc.

Una vez se ha realizado este estudio preliminar al trabajo y todas las tareas se han completado, la Historia de Usuario 01 pasa a estado "*Done*".

6.1.3 Sprint Retrospective

Este proyecto está centrado en temas de investigación actuales y novedosos (sistema de recomendación que usa actualmente *Facebook*). Esta primera Historia de Usuario ha sido la que más trabajo ha llevado de todas las demás. Ha sido necesario ponerse en contacto con el campo del *deep-learning* y aprender muchas técnicas usadas actualmente. Además de la necesidad de entender los

artículos, es necesario un *background* sobre estos temas que no se adquiere en pocas semanas. Además, esta primera Historia de Usuario se ha desarrollado sobre el primer cuatrimestre, ha sido necesario compaginarla junto con las asignaturas de la universidad. Estas cinco subtareas han representado la mayor carga de trabajo, ya que han sido vitales para desarrollar el proyecto y como se observa, han sido explicadas en la Sección 2. Afortunadamente se ha adquirido un conocimiento increíble a cerca del mundo del *deep-learning* y de muchas de las técnicas usadas hoy en día. Además, se ha adquirido una noción del funcionamiento de los sistemas de recomendación, tanto los que usan datos *dense*, como los que usan *dense* y *sparse* y hacen uso de técnicas del análisis predictivo y *deep-learning*, como el modelo *DLRM*.

La tarea 01.02 es también donde se ha invertido una gran parte del tiempo, este componente/algoritmo denominado *EmbeddingBag* es un componente esencial en cualquier sistema de recomendación que trate con datos *sparse*. El hecho de “ponerse en movimiento”, con algo tan novedoso y desconocido ha supuesto una gran parte del tiempo dentro de este primer Sprint para su investigación y entendimiento. Para futuras investigaciones y/o implementaciones se recomienda tratar de investigar el código y observar como los datos se comportan, todo esto, acompasado de un estudio intenso del artículo, cosa que se ha hecho.

Destacar también la tarea 01.05, en la que ha sido necesario investigar cada una de estas arquitecturas aceleradoras por separado (*MAERI*, *SIGMA* y *Google TPU*). A pesar de que comparten algunos componentes funcionales (redes de distribución, redes de reducción, multiplicadores, unidades lógicas, etc.), el hecho de tener que compararlas ha supuesto otra gran parte del estudio.

En la Tabla 8 se muestra la duración final del Sprint a partir de las horas estimadas inicialmente, una estimación ideal y de las horas reales. Afortunadamente, este Sprint se ha logrado desarrollar dentro de los márgenes estimados, finalizando 1 semana antes, de acuerdo con el diagrama de Gantt estimado (Figura 21).

Finalmente, en la Figura 25 se muestra el diagrama *burn down* realizado a partir de la estimación inicial (Sección 5.3). Estimamos la duración de este primer Sprint durante el primer cuatrimestre del curso y con una jornada de 2 horas diaria.

ID	Tarea	P.H.	Duración (h)	
			Estimada	Real
01.01	Leer artículo *DLRM*	6	15	32
01.02	Entender funcionamiento *EmbeddingBag*	15	39	82
01.03	Entender funcionamiento *MLPs*	10	26	15
01.04	Entender funcionamiento *Interact Features*	10	26	10
01.05	Lee artículo *STONNE*	9	23	20
01.06	Estudio sobre aceleradores de inferencia	10	26	50
Total		60	155	209

Tabla 8: Duración final Sprint 1.

Si nos fijamos en el gráfico *burn down* y en la Tabla 8, al comienzo, aunque la tarea 01.01 nos tomó mas tiempo, logramos hacerla por debajo de la estimación ideal. Sin embargo, la tarea 01.02, como se observa en la tabla y en el gráfico, nos tomó mas tiempo, esto es debido a la dificultad que ya hemos comentado anteriormente, no solo la dificultad, sino la importancia de este componente. El resto de tareas se desarrollaron dentro del plazo estimado de tiempo y finalmente, en la tarea 01.05, se hizo una estimación de horas por debajo de lo real, a pesar de esto, aunque en el gráfico se observa una desviación, conseguimos finalizar el proyecto una semana laboral antes del plazo estimado.

Lo ideal sería representar los P.H. totales y los que se han ido completando día a día, en lugar de las horas que se han trabajo diariamente, ya que esta es la función del gráfico de *burn-down*, sin embargo, para mostrar una visión más realista y de acuerdo con la planificación temporal realizada en la Sección 5.3 (estimación realizada en horas), se muestra en el eje Y las horas de trabajo.

6.2 Sprint 2

Una vez adquirido el conocimiento relativo al campo del *deep-learning* y al modelo *DLRM*, comenzamos con el desarrollo de un *benchmark*, una simulación del modelo para comprobar así que se entiende el funcionamiento de los diferentes componentes. Previo al desarrollo, se instalará el entorno software necesario en el servidor SSH de la Universidad.

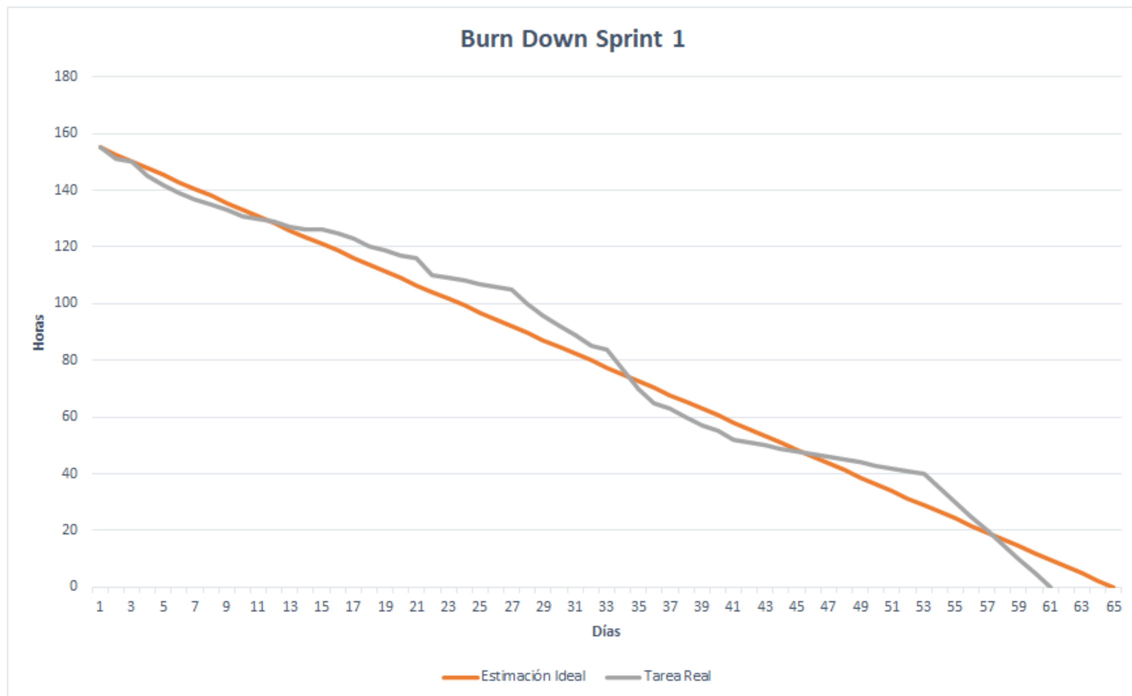

Figura 25: Gráfico de *burn down* para el Sprint 1.

6.2.1 Sprint planning

En la Figura 26 se muestran las Historias de Usuario seleccionadas, junto con las sub-tareas, que se desarrollarán durante este segundo Sprint.

Como se ha mencionado brevemente al comienzo de este Sprint, el Sprint Goal es:

- Instalar el entorno de desarrollo (Historia de Usuario 02). Para realizar este proyecto es necesario instalar *Python*, *Anaconda*, *PyTorch* para ejecutar el modelo *DLRM*. Y por otra parte, compilar las librerías que *STONNE* necesita mediante el gestor de librerías *Anaconda* previamente instalado.

- Codificar *benchmark* del modelo *DLRM* (Historia de Usuario 03). Una vez se han identificado muy bien los componentes y sabemos como funcionan, replicar el funcionamiento de estos en *CPU* usando *PyTorch*. Esto nos permitirá realizar la integración del modelo *DLRM* en *STONNE* de manera más fácil.

Como primer paso fundamental es necesario tener preparada una infraestructura para el desarrollo del software. Al estar trabajando en un servidor donde no

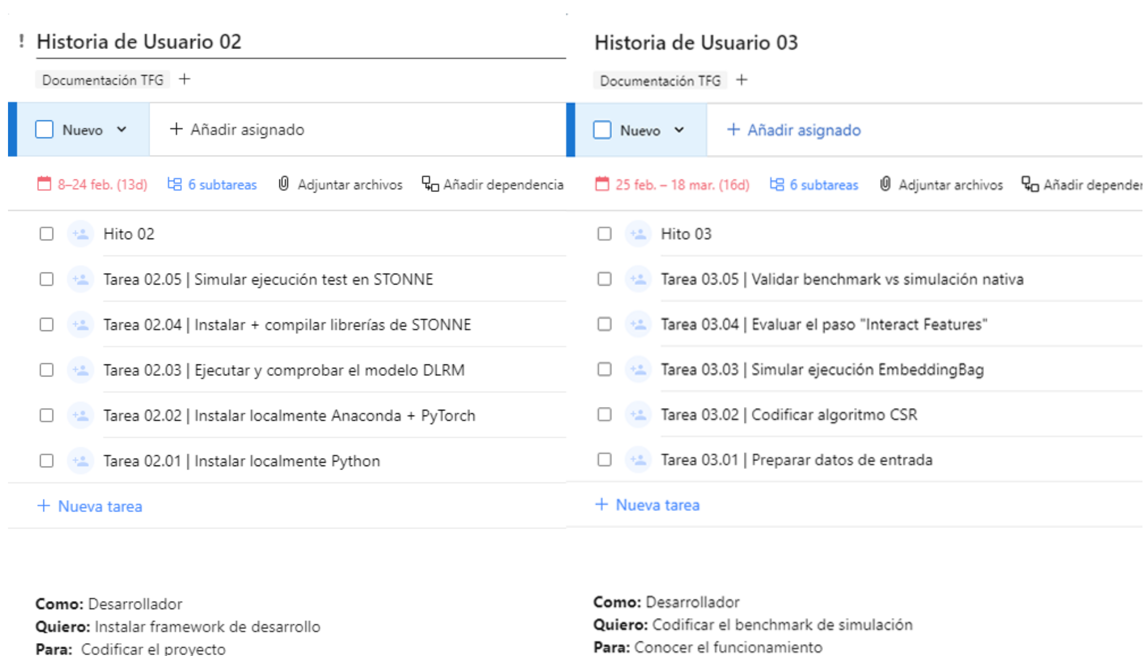

Figura 26: Historias de Usuario para el Sprint 2.

soy *super-user su* y además, existen otros usuarios realizando ejecuciones sobre *Python*, no puedo realizar cambios al compilador, instalar nuevas versiones o cambiar paquetes/librerías, es por eso necesario realizar una instalación del entorno de desarrollo de manera local.

Una vez logrado este primer paso, se instalarán las librerías necesarias para el modelo *DLRM*, *PyTorch*. Se podría optar por *Caffe*, pero como ya se ha explicado en la Sección 2.3.4 lo descartamos. Una vez instaladas se realizará una ejecución de prueba con el modelo *DLRM*, como la que aparece en el repositorio (Facebook, 2021), para comprobar su funcionalidad.

De igual manera, se instalaran de manera local los paquetes y librerías necesarias para *STONNE* y se realizará una ejecución de prueba de acuerdo con las indicaciones en su GitHub oficial (Martínez, Abellán, Acacio, y Krishna, 2021).

Una vez instalado el entorno de desarrollo y comprobada su funcionalidad, en la Historia de Usuario 03 se realizará un *benchmark* simulando la ejecución del modelo *DLRM*, la idea de esta simulación es comprobar que la ejecución aislada de las *EmbeddingBags*, *MLPs* y *Feature-Interaction* es la misma que la del modelo, asumiendo los mismos datos de entrada. Esta Historia de Usuario se divide en subtareas para facilitar el desarrollo.

En la Figura 27 se muestra el tablero de *Wrike* al comienzo del sprint.

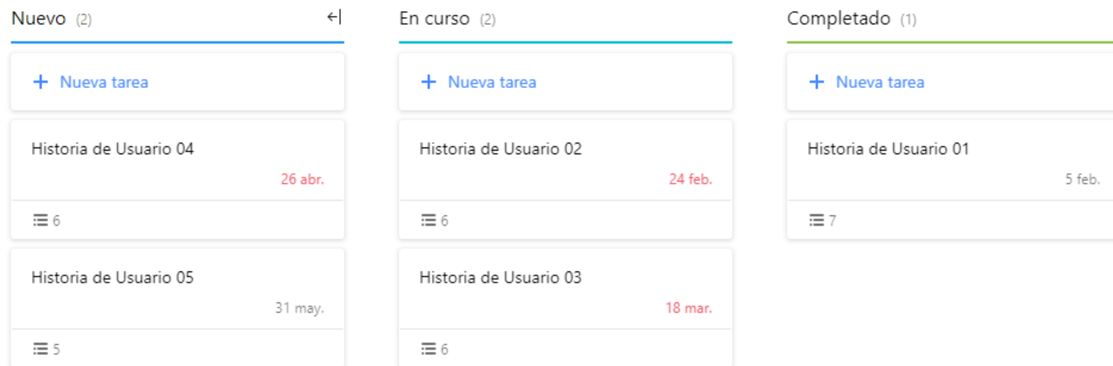

Figura 27: Tablero para el Sprint 2.

6.2.2 Sprint review

En la primera tarea, 02.01 se ha instalado localmente *Python* y también ha servido como estudio para los *frameworks* de *deep-learning* que existen hoy en día, Figura 28. La justificación para realizar el proyecto usando *PyTorch* queda reflejado en la Sección 2.3.4. También ha servido para profundizar en los conocimientos del lenguaje de programación *Python* y el peso de algunas librerías dentro del campo de la ciencia de datos, como *NumPy*, *SciPy*, *Pandas*, *Matplotlib* y muchas más.

En la tarea 02.02 y 02.04 se ha seguido un enfoque igual a la tarea 02.01, se ha instalado *Anaconda* localmente y a partir de ahí se han instalado las librerías ne-

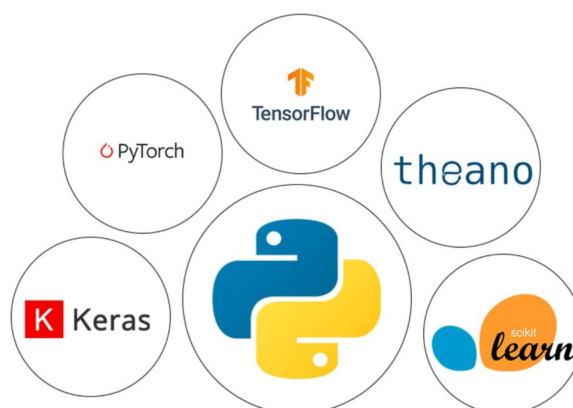

Figura 28: *Frameworks* para *deep-learning* escritos en *Python* (Costa, 2020).

cesarias. Cuando hacemos referencia a “instaladas localmente”, queremos decir que solo pueden ser usadas a través del *home* del usuario que las haya instalado, de tal manera que no afectamos a las variables globales, a las variables del entorno del sistema, por lo que no perjudicamos a otros usuarios que estén haciendo uso de la versión instalada de *Python* en el servidor.

Para las tareas 02.03 y 02.05, una vez instaladas todas las librerías necesarias y compiladas se ha comprobado correctamente el funcionamiento, además, se ha verificado la salida del modelo *DLRM* con la salida en modo *debug* que aparece en su GitHub oficial (Facebook, 2021). Para *instalar* el modelo *DLRM* se ha realizado un *git clone* del repositorio.

Antes de hablar sobre la Historia de Usuario 03 hay algunos aspectos que debemos reflejar y dejar claros. Toda la instalación del entorno de desarrollo, tanto para el modelo *DLRM*, como para *STONNE* se ha hecho en CPU, es decir, no se ha instalado ninguna librería relacionada con GPU (*CUDA*, *cuDNN*, *keras-gpu*, etc.), ya que estamos centrados en el proceso de inferencia, toda la ejecución en *GPU* no es motivo de este trabajo y queda descartada. Para instalar *STONNE*, de igual manera que *DLRM* se ha realizado un *git clone* al repositorio privado denominado *stonne-dev*. La herramienta se encuentra bajo desarrollo pero se puede encontrar una versión demo (Martínez y cols., 2021).

La Tabla 9 muestra las versiones instaladas de los *frameworks* y librerías (las más relevantes) en el entorno local del servidor.

Python	**PyTorch**	**Anaconda**	**NumPy**	**pip**
3.8.0	1.7.0	4.9.2	1.19.4	20.2.4

Tabla 9: Versiones de *frameworks* y librerías.

Una vez llegados a este punto los entornos de desarrollo para el proyecto (*Python* + librerías) quedan instalados correctamente (tareas 02.01, 02.02 y 02.04). Destacar que para la instalación de *STONNE*, el uso del gestor de paquetes *Anaconda* ha facilitado el proceso de creación y de instalación. Finalmente, se ha comprobado el correcto funcionamiento de los frameworks instalados y del modelo *DLRM* + el simulador *STONNE*. Tras algunas ejecuciones fallidas del

simulador *STONNE* por falta de información se ha conseguido obtener una ejecución exitosa, de tal manera que las tareas 02.03 y 02.05 se han completado con éxito, dando lugar a finalizar la Historia de Usuario 02.

Respecto a la segunda Historia de Usuario 03, se han abordado diferentes tareas, las cuales introduciremos en su debido orden. Para la tarea 03.01 se han preparado los datos de entrada de acuerdo a como el modelo *DLRM* los acepta. Para esto hemos creado diferentes cargas de trabajo simuladas, no reales, con el fin de modelar estos datos para introducirlos como datos de entrada al modelo *DLRM*, tras algunas simulaciones incorrectas por parte de los datos *sparse* se ha logrado modelar los datos para que sean aceptados y procesados por el modelo *DLRM*. Se han creado dos tipos de datos, un vector *dense*, generado aleatoriamente que será procesado por la *Bottom-MLP* y una “matriz” sparse en formato CSR (compuesta por dos vectores, uno denominado *offsets* y otro *indices*). Para la ejecución nativa y simulada del modelo *DLRM* se usarán las mismas cargas de trabajo, con el fin de demostrar que la salida de ambos modelos es la misma y por tanto se entiende como funcionan los componentes.

Una de las tareas mas desafiantes dentro de esta Historia de Usuario 03, son los datos *sparse* y como estos se procesan, es muy importante enviarlos de manera correcta al simulador cuando hagamos la implementación en *STONNE*. Como estamos realizando una simulación del modelo en *CPU*, una vez generados los índices de *offsets* e *indices* que representan el formato CSR es necesario convertirlos a un vector denominado *multi-hot encoding* que en su mayoría está compuesto de ceros. Es por eso necesario, crear un algoritmo que simule el conversor CSR-multi-hot encoding que se encuentra dentro de la arquitectura de *SIGMA*, Figura 11 que posteriormente usaremos. Por mantener la documentación limpia, se ha creado un algoritmo que se encarga de la conversión dentro de este benchmark de complejidad:

$$O(n^2)$$

Este algoritmo se encarga de generar vectores *multi-hot-encoding* (tantos como búsquedas hayan por *EmbeddingBag*) (Figura 7) y enviarlos a CPU donde se ralizará la operación. Han habido varias complicaciones en este punto que serán

explicadas en el *sprint retrospective*. Aún así, podemos concluir con que la tarea 03.02 se ha completado con éxito.

Una vez tenemos los datos generados y se han procesado por la unidad *CSR* generando un *multi-hot-encoding* tenemos que realizar una operación *sparse* como la de la Figura 24. Esta operación puede verse como una suma de aquellas columnas cuyos índices han sido seleccionados con "*unos*" por el vector *multi-hot-encoding*. Esta operación *GEMM Sparse* se ha realizado de dos acercamientos diferentes, uno mediante la librería *NumPy*, haciendo un *dot-product* del vector *multi-hot* por la matriz de pesos de la *EmbeddingBags* y el resultado de cada *EmbeddingBag* agrupándose en una matriz (el funcionamiento de la *EmbeddingBag* queda explicado en la Sección 6.1.2). Y otro haciendo uso de la librería *Sparse* dentro del *framework* de *PyTorch*, que es como el modelo *DLRM* lo realiza de manera nativa. Esto es algo muy importante, porque nos permite comprobar si ambos resultados son iguales y por ende si se ha entendido correctamente el funcionamiento del algoritmo *EmbeddingBag*, un algoritmo crucial en los sistemas de recomendación. Como resultado final de esta tarea 03.03 el valor de ambas ejecuciones es exactamente el mismo, Figura 29, dando como resultado un entendimiento favorable del funcionamiento de este algoritmo, este aspecto favorecerá la implementación del modelo en el simulador *STONNE*.

```
nico@boston:~/GitHubNico/dlrm$ python dlrm_s_pytorch.py
Using CPU...
time/loss/accuracy (if enabled):
[[0.42708 0.15925]]
[[-0.18108  0.26448]]
[[-0.44902 -0.45898]]
```

```
Llega al punto 2
Executed with exit
[[0.42708 0.15925]]
[[-0.18108  0.26448]]
[[-0.44902 -0.45898]]
```

Figura 29: Salida *Embedding DLRM vs benchmark*.

En la tarea 03.04 se ha evaluado el paso *Feature-Interaction* y tras comprobar el impacto en el cómputo que tiene en CPU (Figura 8 (<1 % bmm, batch matrix multiplication) se ha decidido no implementarlo dentro del simulador ya que no aportaría ningún beneficio, sino en todo caso, podría enlentecer su ejecución.

Este paso se realiza en *CPU*, como ya hemos explicado anteriormente (ver Sección 6.1.2), en este paso se realiza una factorización de matrices y posteriormente se le concatena la salida de las *MLPs*, este cómputo se puede realizar en *CPU* de manera más rápida y eficiente que llevar el trasiego de datos mediante un bus como *PCIe* al simulador, realizar el cómputo y devolver el resultado.

Un aspecto que no hemos detallado en esta sección son las *MLPs*, al contar el simulador con la ejecución de *MLPs* nativa no es necesario replicar su funcionamiento en el *benchmark*, en su lugar se usa la librería *Linear* de *PyTorch* para realizar las *MLPs*, esta es la misma libreria que usa el modelo DLRM de manera nativa en CPU.

Como última tarea dentro de esta tercera Historia de Usuario, 03.05 se pide validar el *benchmark* creado *versus* la simulación nativa del modelo *DLRM*. Como se observa en la Figura 30 la salida de ambos modelos es exactamente la misma usando los mismos datos de entrada. Esto como resultado pone de manifiesto que la Historia de Usuario 03 se ha completado con éxito.

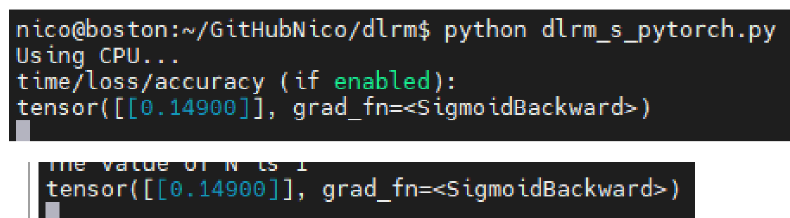

Figura 30: Salida *DLRM* nativo *vs benchmark*.

6.2.3 Sprint retrospective

Este segundo sprint está centrado en desarrollar el entorno de desarrollo para el proyecto y de realizar la implementación del modelo *DLRM* a nivel de simulación para comprobar si se ha entendido bien el funcionamiento de las diferentes partes.

En la tarea 02.01 se han encontrado dificultades, el hecho de instalar un paquete, *Python* de manera local en el sistema, para que solamente un usuario pueda usarlo y posteriormente las librerías, todo esto sin afectar al resto de usuario ha supuesto un reto. Afortunadamente se ha aprendido a como instalar paquetes

de manera única en el *home* de un usuario para así aislarlos del resto de usuarios y evitar problemas de incompatibilidad a nivel general en el servidor.

La tarea 02.04 también ha supuesto otro gran reto, el simulador *STONNE* hace uso de varias librerías internas y debido a su arquitectura y forma de estar programado, lo que tiene es una *API* que conecta *Python* con *C++* para su ejecución. Las librerías de *deep-learning* que usa *STONNE* se encuentran escritas en *PyTorch* y estas tienen que ser compiladas por una versión de *PyTorch* muy específica. Se puede encontrar un manual de instalación en el Anexo 10.1.

Respecto a la Historia de Usuario 03, aparecieron algunos errores de formato en el algoritmo *CSR*. El algoritmo debía procesar los vectores *indices* y *offsets*, que se encontraban en formato *tensor* y enviarlos en formato *matriz* al componente *EmbeddingBag*. Resulta que *Python* añade comillas a la salida de cada tensor, como la unidad *CSR* genera varios tensores, al final hay que juntarlos todos en la misma matriz. Se optó por transformar los tensores en vectores haciendo uso de *NumPy*, juntarlos todos en una matriz y posteriormente volver a convertirlos a tensores, de esta manera se logró solucionar el problema y la unidad *EmbeddingBag* ya podía procesar los datos.

Como recomendación futura se aconseja crear entornos virtuales aislados donde se encuentren todos los paquetes instalados para poder ejecutar *DLRM* y *STONNE*, de esta manera no habrán problemas y se podrá portar el entorno de un servidor a otro.

En la Tabla 10 se muestra la duración final del Sprint a partir de las horas estimadas inicialmente, una estimación ideal y de las horas reales. Afortunadamente, este Sprint se ha logrado desarrollar dentro de los márgenes estimados, de acuerdo con el diagrama de Gantt (Figura 21).

Finalmente, en la Figura 31 se muestra el diagrama *burn down* realizado a partir de la estimación inicial (Sección 5.3). Estimamos la duración de este segundo Sprint al comienzo del segundo cuatrimestre del curso y con una jornada media, es decir, 4 horas de trabajo al día durante 29 días laborales. Como se observa en la gráfica, las tareas 02.01 y 02.04 se estimó el trabajo de horas por debajo de lo real, por lo que se observan dos incrementos de trabajo durante

ID	**Tarea**	**P.H.**	**Duración (h)**	
			Estimada	**Real**
02.01	Instalar localmente *Python*	3	8	12
02.02	Instalar localmente *Anaconda* y Python	2	5	8
02.03	Ejecutar y comprobar el modelo *DLRM*	5	13	5
02.04	Instalar y compilar las librerías de *STONNE*	5	13	27
02.05	Simular ejecución en *STONNE*	4	10	4
03.01	Preparar datos de entrada	4	10	16
03.02	Codificar algoritmo *CSR*	2	5	15
03.03	Simular ejecución *EmbeddingBag*	10	26	34
03.04	Evaluar el paso *Interact Features*	3	8	3
03.05	Validar *benchmark vs* simulación nativa	4	10	4
Total		42	108	128

Tabla 10: Duración final Sprint 2.

las dos primeras semanas de este Sprint. Afortunadamente, las estimaciones de algunas tareas estaban por debajo de las horas de trabajo reales, por lo que al final se ha conseguido que las horas estimadas estén conforme a las reales de trabajo, a pesar de existir diferencias.

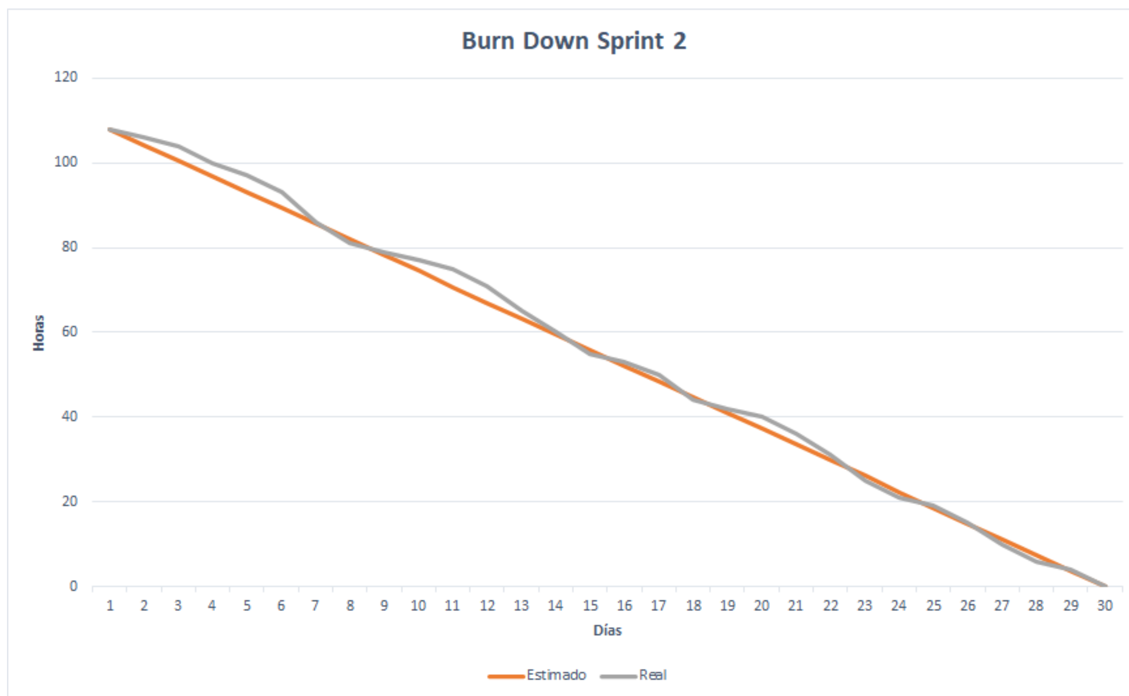

Figura 31: Gráfico de *burn down* para el Sprint 2.

6.3 Sprint 3

En este Sprint se desarrollará la tarea principal del proyecto, integrar el modelo *DLRM* dentro del simulador *STONNE*. Es muy importante asegurarse del correcto funcionamiento (realizar una evaluación) y posteriormente configurar los aceleradores que computan las cargas de trabajo del modelo.

6.3.1 Sprint planning

En la Figura 32 se muestran las Historias de Usuario seleccionadas, junto con las sub-tareas, que se desarrollarán durante este tercer Sprint.

El Sprint Goal se puede dividir en dos objetivos muy bien identificados:

- Integrar el modelo *DLRM* en *STONNE*. Dar capacidad a *STONNE* para ejecutar las nuevas capas del modelo *DLRM* y posteriormente realizar la conexión *DLRM-STONNE* mediante su *API*.

- Posteriormente, configurar los aceleradores de inferencia. Una vez la integración se ha realizado con éxito se deben configurar los aceleradores de

Historia de Usuario 04

Documentación TFG +

Nuevo + Añadir asignado #690830478 por Nicolás M. a las 15:17

19 mar. – 26 abr. (27d) 6 subtareas Adjuntar archivos Añadir dependencia Compartido con 1 grupo y 1 persona

Tarea	Fecha	Estado
Hito 04	26 abr.	Nuevo
Tarea 04.05 | Comprobar resultados de ejecución	19 abr.	Nuevo
Tarea 04.04 | Configurar aceleradores de inferencia	14 abr.	Nuevo
Tarea 04.03 | Conectar DLRM con STONNE mediante su API	8 abr.	Nuevo
Tarea 04.02 | Preparar los datos de entrada para MLPs y Embeddings	31 mar.	Nuevo
Tarea 04.01 | Implementar algoritmo Embedding en STONNE	26 mar.	Nuevo

+ Nueva tarea

Como: Desarrollador
Quiero: Integrar DLRM en STONNE
Para: Proponer una arquitectura híbrida

Figura 32: Historias de Usuario para el Sprint 3.

inferencia para que realicen el cómputo de las operaciones *dense* y *sparse*. A partir del estudio realizado en la tarea 01.06.

En el anterior Sprint se ha logrado desarrollar ese *benchmark* que simulas las diferentes partes del modelo de recomendación *DLRM*. A continuación se va a seguir el mismo procedimiento, se integrarán los diferentes componentes del modelo de recomendación (comprendido entre las tareas 04.01 hasta 04.02). Una vez *STONNE* sea capaz de computar el componente *EmbeddingBag*, entonces se realizará la conexión DLRM-API de STONNE.

Una vez el modelo se haya conectado y se haga una evaluación de los datos de salida, entonces pasaremos a las tareas 04.04 y 04.05, configurar los aceleradores de inferencia y posteriormente volver a comprobar los resultados con el modelo DLRM nativo.

En la Figura 33 se muestra el tablero de *Wrike* al comienzo del sprint.

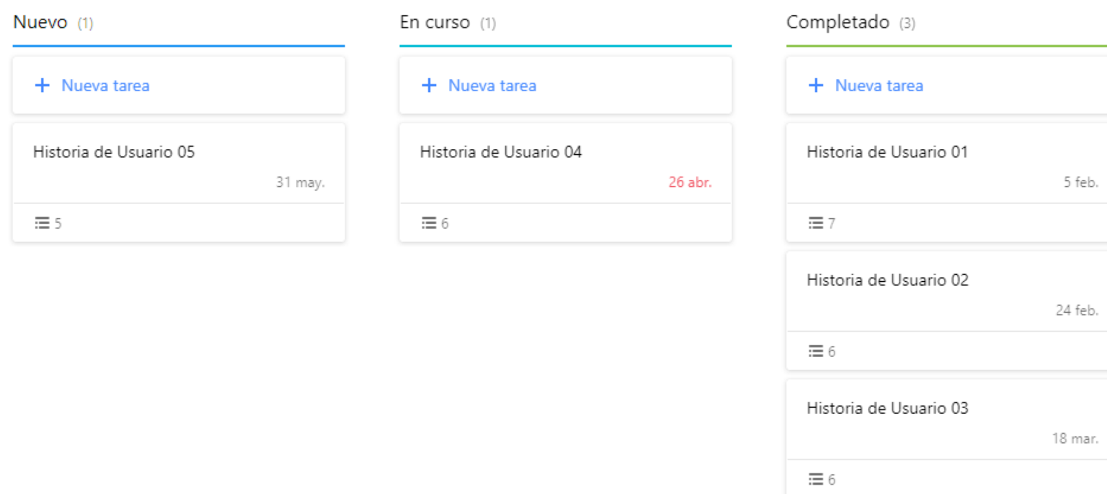

Figura 33: Tablero para el Sprint 3.

6.3.2 Sprint review

Partimos de saber perfectamente como funciona el algoritmo *EmbeddingBag*, en su esencia que hace y como funciona. Para dotar a *STONNE* de la capacidad para ejecutar esta nueva capa, primero, vamos a realizar otro pequeño *benchmark*, pero este se va a ejecutar sobre *STONNE* (algoritmo EmbeddingBag y MLPs) y seguiremos un procedimiento similar al que hicimos en el anterior sprint.

Se ha cogido un dataset sintético, igual al que se ejecuta sobre *DLRM* nativo (mismos datos de entrada). Tras estudiar en profundidad el artículo de *STONNE* se ha seguido la siguiente metodología para implementar el algoritmo *EmbeddingBag* en STONNE. *STONNE* se ejecuta sobre código *C++*, pero los modelos de *deep-learning* soportados se encuentran escritos en *PyTorch*, por eso es necesario una *API* que convierta ese código de *PyTorch* a *C++* (unidades STONNE API + Compiler, ver Figura 9) para que se ejecute dentro del simulador.

El primer paso dentro de esta implementación ha sido duplicar la librería *Sparse* (con el nombre *SimulatedSparse*), librería encargada de las *EmbeddingBags*. Con el conocimiento obtenido en el sprint 1, sabemos que existe un método denominado “*forwarding*”, dentro de la clase *EmbeddingBag*, que se encarga de procesar los datos de las redes neuronales. Es decir, a partir de unas entradas, comenzar a distribuir los pesos por la red y posteriormente computarlos. Esta

librería *Sparse* se encuentra dentro del paquete nn de *PyTorch*.

Para conectar el componente *EmbeddingBag* a *STONNE*, necesitamos que el método *forwarding* llame a *STONNE*. De tal manera que la operación de las *EmbeddingBags*, que gracias al estudio y el benchmark realizado anteriormente sabemos que es una operación *GEMM Sparse*, se realice en el simulador *STONNE*.

Por otra parte, dentro del simulador existe una función denominada *sparse_mat_mult*, que en esencia es una *GEMM Sparse*. Esta función se encuentra en la API de STONNE y permite ser llamada desde el frontend de *PyTorch* (nuestra libreria SimulatedSparse). Esta función recibe como entrada diferentes tipos de parámetros; formato CSR (indices y offsets), matriz dense, archivo de configuración para la arquitectura a simular, valores de *sparsity*, etc.

De tal manera que, para lograr que la operación GEMM Sparse se ejecute en STONNE (sin arquitectura simulada, lo veremos más adelante), modificaremos la función *forwarding*. Importando la libreria de STONNE (el frontend), torch_stonne, lograremos que dentro de este *forwarding* se realize la llamada a la operación *sparse_mat_mult* dentro del simulador; los parámetros serán abordados a continuación. A parte de modificar la función *forwarding* se han añadido algunas propiedades extra a la clase *EmbeddingBag* para que esta pueda ser ejecutada en STONNE, como esto no es relevante no lo mencionaremos.

A partir de ahora, ya no importaremos la librería *Sparse*, que se ejecuta en *CPU* nativa, en su lugar importaremos la librería *SimulatedSparse* que el *forwarding* conecta con la *API* de *STONNE*.

Puesto que *STONNE* permite transformar código *Python* a *C++* mediante su *API* y la operación de multiplicación matrices ya se encuentra implementada en el simulador, la conexión de la *EmbeddingBag* con *STONNE* ya se ha realizado, concluimos así la primera tarea de este tercer sprint.

Para aclarar lo que llevamos hasta ahora, la operación *EmbeddingBag* ya se puede computar en *STONNE*, pero aún falta codificar los datos *sparse* para enviarlos al simulador, esta es una tarea que se abordará en los próximos párrafos.

Para la tarea 04.02 partimos de un caso peculiar, los datos *dense* se pueden

generar sin problema y computar, pero los datos *sparse* a parte de generarlos, hay que enviarlos mediante la *API* de *STONNE* para que se genere ese vector *multi-hot encoding* y posteriormente se ejecuten en el simulador. En este punto se alargó más el trabajo debido a varias complicaciones que surgieron, estos problemas serán comentados en el sprint retrospective.

Como ya hemos mencionado en varios puntos anteriores, el simulador *STONNE* ofrece soporte a las capas *MLPs*, por lo que no es necesario su implementación. La forma de llamarlas se produce de igual manera que con las *EmbeddingBags*. En el *forwarding* de la librería *SimulatedLinear* se realiza la llamada a la *API*. El cómputo de las *MLPs* difiere del modelo nativo, lo comentaremos en el sprint retrospective. La entrada de datos a las *MLPs* se conserva tal y como existe en el modelo DLRM, mediante un vector dense que se envía junto la matriz de pesos de la MLP a *STONNE* para computar.

Los datos de entrada que genera el modelo *DLRM* varían dependiendo del tipo del tamaño de ejecución que se realice (número *EmbeddingBags*, tamaño de las *MLPs*, número de *lookups/EmbeddingBag*, etc.). En el próximo sprint propondremos diferentes casos de estudio con el fin de analizar como el modelo se comporta. Por ejemplo, el mayor o menor cómputo de las *MLPs* viene determinado por el número de *lookups/EmbeddingBag* que realicemos, además de esta co-relación existen muchas más en todo el modelo que serán expuestas en los resultados (Sección 6.4.2). Esta etapa se ha alargado más debido a la necesidad de saber como se generan los datos y como afectan al modelo.

Además, es necesario saber como gestionar los datos. Como ya se ha comentado, los datos dense no es necesario modificarlos puesto que esta unidad ya se encuentra implementada en *STONNE*. Sin embargo, para comprobar que el componente *EmbeddingBag* se ha implementado correctamente en STONNE, necesitamos realizar una ejecución con una carga de trabajo; para ello, necesitamos emular esa unidad “conversor *CSR*-multi-hot encoding” que se encuentra en la arquitectura de SIGMA (ver Figura 11), de tal manera que podamos comprobar si el forwarding de nuestro *SimulatedSparse* se ha implementado correctamente.

Tras varios intentos fallidos se ha logrado que el *benchmark* sea capaz de

enviar esos datos *sparse*, convertidos a multi-hot encoding, al simulador y por ende se pueda ejecutar una *EmbeddingBag* en *STONNE*. En la Figura 34 se puede observara la ejecución de las *EmbeddingBags* en el modelo *DLRM* nativo y la ejecución de estas en el simulador *STONNE*.

```
nico@boston:~/GitHubNico/dlrm$ python dlrm_s_pytorch.py --arch-embedd
rch-mlp-bot=12-6-4 --arch-mlp-top=32-16-1 --data-generation=random --
ta-size=2
Using CPU...
time/loss/accuracy (if enabled):
[[-0.3273   0.46337 -0.3113  -0.19785]
 [ 0.14789 -0.22271 -0.37634 -0.52509]]
[[ 0.30154 -0.10239  0.95653  0.39881]
 [ 0.7972  -0.07071  0.1847   0.03301]]
[[-0.1124  -0.52261 -0.69579 -0.01871]
 [-0.55666  0.47513 -0.92619 -0.37318]]
```

```
Time ds: 0
Llega al punto 2
Executed with exit
[[-0.3273   0.46337 -0.3113  -0.19785]
 [ 0.14789 -0.22271 -0.37634 -0.52509]]
[[ 0.30154 -0.10239  0.95653  0.39881]
 [ 0.7972  -0.07071  0.1847   0.03301]]
[[-0.1124  -0.52261 -0.69579 -0.01871]
 [-0.55666  0.47513 -0.92619 -0.37318]]
```

Figura 34: EmbeddingBags en *DLRM vs STONNE*.

En este punto, se ha implementado el algoritmo *EmbeddingBag* en *STONNE* y sabemos como hay que tratar los datos *sparse* y *dense* para que puedan ser computados. El *benchmark* ha sido un éxito y el siguiente paso es ahora conectar el modelo *DLRM* nativo a *STONNE*.

El primer paso dentro de esta tarea 04.03 es, llamar a las librerías simuladas que se ejecutan en *STONNE*, *SimulatedLinear* para las *MLPs* y *SimulatedSparse* para las *EmbeddingBags*. Una vez realizado, se ha de gestionar el conversor *CSR-multi-hot* para que el tipo de dato que le enviamos al simulador (formato CSR) a través de la *API* sea el correcto. Llegados a este punto, ambas *MLPs*, *Bottom* y *Top* y las *EmbeddingBags* se ejecutarán en el simulador. Una vez se ha completado, podemos decir que la implementación del modelo *DLRM* en *STONNE* ha sido un éxito. Concluimos así con las 3 primeras tareas de este tercer sprint.

Gracias a la investigación realizada en la tarea 01.06 sabemos como funcionan los aceleradores de inferencia y que características ofrecen cada uno, esto nos sirve para ahora escoger que tipo de arquitectura queremos simular en

	Número *PEs*	***Bandwidth* RD**	***Bandwidth* RR**	**CM**
MAERI	256	128	128	DENSE
SIGMA	128	128	128	SPARSE

Tabla 11: Archivo de configuración para las arquitecturas.
RR - Red de reducción. RD - Red de distribución. CM - Controlador de memoria

STONNE.

Una vez el modelo *DLRM* se encuentra integrado dentro de *STONNE*, podemos configurar las llamadas a la *API* de *STONNE* para que mediante un archivo de configuración (*stonne config*), ver Figura 9, *STONNE* simule una arquitectura u otra (unidad *Simulation Engine*). En el caso de los datos *dense*, usamos la arquitectura *MAERI* con su archivo de configuración denominado: *maeri_256mses*, este archivo de configuración establece el tipo de red de reducción, número de multiplicadores, el tipo de controlador, ancho de banda, etc. La arquitectura de *MAERI* puede verse en la Figura 10. Toda esta investigación a cerca de configuraciones arquitecturales; proponer un mejor diseño arquitectural de los aceleradores para DNN podría considerarse trabajo futuro puesto que se sale de las horas estimadas para el presente TFG.

De igual manera sucede para el cómputo *sparse*, usamos la arquitectura *SIGMA* con su archivo de configuración denominado: *sigma_128mses*, un controlador que usa la arquitectura de *SIGMA*. Mediante este archivo de configuración podemos modificar el ancho de banda de la arquitectura, número de multiplicadores, etc. Se puede encontrar la arquitectura de *SIGMA* en la Figura 11.

En la Tabla 11 se muestran los parámetros más relevantes que hemos configurado de los fichero de configuración para las arquitecturas *MAERI* y *SIGMA*.

Hemos escogido un valor de parámetros similares a los considerados en los artículos de *MAERI* y *SIGMA*, un tamaño intermedio en comparación con otras propuestas (por ejemplo, la TPU). La diferencia principal entre ambas configuraciones se observa en el número de *Processing Elements*, *PEs*, en el caso de *SIGMA* se usan la mitad, puesto que “al explotar el *sparsity* se obtiene una mejor eficiencia” (y otros, 2020).

Como en este trabajo nos centramos en desarrollar una arquitectura acelera-

dora, para evitar entrar en jerarquías de memoria y los posibles problemas que podrían ocurrir, vamos a asumir un tamaño de *Global Buffer* ilimitado.

Una vez los archivos de configuración se han configurado de acuerdo al estudio realizado sobre los aceleradores de inferencia y se conectan con la llamada a la *API* de *STONNE*, podemos concluir en que esta tarea 04.04 se ha completado correctamente.

En la Figura 35 se observa la ejecución del modelo *DLRM*, usando las librerías *SimulatedLinear* y *SimualtedSparse* (se ha obviado la palabra *Simulated* para que la imagen se al estilo de página).

Primero se ejecuta la *Bottom-MLP* (cómputo dense), a través de la librería *nn.Linear STONNE* simula la arquitectura aceleradora *MAERI*. Posteriormente, se ejecutan las *EmbeddingBags* (cómputo sparse), llamando a la librería *nn.Sparse* STONNE simula la arquitectura *SIGMA*. A continuación, el paso *Feature-Interaction* se realiza en *CPU*, los motivos han sido expuesto en Sprints anteriores (ver Sección 6.1.2). Finalmente, la *Top-MLP*, al igual que la *Bottom-MLP*, envía los datos mediante el forwarding al simulador para que simule la arquitectura *MAERI* y realice el cómputo dense final.

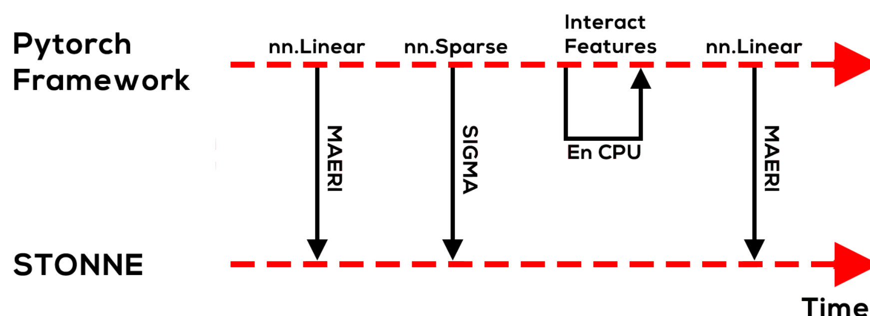

Figura 35: Ejecución del modelo *DLRM* en *STONNE* con *MAERI* y *SIGMA*.

Finalmente se realiza una ejecución con un *dataset* sintético solamente con el objetivo de comprobar que la salida en ambos modelos es la misma, como se observa en la Figura 36 la salida de ambas ejecuciones es la misma, por lo que podemos concluir en que la integración del modelo *DLRM* usando las arquitecturas aceleradoras *MAERI* y *SIGMA* para el cómputo *dense-sparse* se ha configurado con éxito dentro del simulador *STONNE*.

```
nico@boston:~/GitHubNico/dlrm$ python dlrm_s_pytorch.py
rch-mlp-bot=16-8-4 --arch-mlp-top=16-1 --data-generatio
size=1
Using CPU...
time/loss/accuracy (if enabled):
tensor([[0.57362]], grad_fn=<SigmoidBackward>)
```

```
Printing stats
Number of cycles running: 23
Time mem: 0
Time lt: 0
Time as: 0
Time ms: 0
Time ds: 0
The value of M is 1
The value of N is 1
tensor([[0.57362]], grad_fn=<SigmoidBackward>)
```

Figura 36: Modelo *DLRM* nativo *vs STONNE*.

6.3.3 Sprint retrospective

Este Sprint ha sido clave para el desarrollo del proyecto, se ha logrado integrar el modelo de recomendación *DLRM* en la herramienta de simulación *STONNE*, solo queda realizar pruebas para ver como la arquitectura que proponemos se comporta ante diferentes cargas de trabajo.

La tarea 04.01 generó cierto retraso, ya que apareció un problema dentro del simulador *STONNE*, las librerías de *PyTorch* que usa el simulador no habían sido compiladas correctamente y cualquier cambio realizado en los ficheros *SimulatedSparse* o *SimulatedLinear* generaba una excepción del simulador en tiempo de ejecución. Esto se debe a un problema del servidor de la Universidad, ya que tiene una versión de compilador *GNU* muy antigua y algunas instrucciones escritas en *C* no las reconoce. Para poder resolver este problema se modificó el código de algunos ficheros escritos en *C* y *C++*. Finalmente, se compiló nuevamente el *frontend* de *STONNE* (*PyTorch*) dentro del entorno *conda* y exitosamente se logró compilar sin ningún problema. En este punto ya estábamos listos para comenzar con la tarea 04.01. El error que apareció en el Sprint 2 y en este han sido solucionados y aparecen detallados en el Anexo 10.1.

La tarea 04.02, tomó mas tiempo de lo normal. No debido a como se generan los datos *sparse* y *dense*, sino más bien a como hay que tratarlos para enviarlos

al simulador *STONNE*. La función escrita en la *API* encargada de llamar al código *C++* acepta solo un tipo de vector en formato *sparse*, se probó a intentar crear este vector sparse de diferentes formas, como un vector escrito en *NumPy*, como un vector *sparse* escrito con la librería de *TensorFlow* y finalmente, se consiguió crear un *sparse-vector* a través de *PyTorch* con el uso de su documentación, aunque no se desvió mucho de la estimación de horas de trabajo, supone un pequeño plus. Afortunadamente se ha aprendido a como generar el vector *sparse* en diferentes lenguajes.

Respecto al funcionamiento de las *MLPs*, el modelo nativo de *DLRM* computa todas las capas de las *MLPs* continuamente, es decir, la salida de una capa, alimenta a la siguiente capa, esto se consigue agrupando las capas de las *MLPs* mediante un *Module List*, elemento de *PyTorch*. Sin embargo, necesitamos que las capas se computen de manera secuencial, de una en una, para eso, en vez de usar un *ModuleList* que agrupe todas las capas, mantenemos las capas dentro de un vector y por cada capa se realiza una llamada a *STONNE*. La salida de cada capa alimenta a la siguiente, es más complejo que usar la aproximación del componente *Module List*, pero no supone ningún impacto en la eficiencia del modelo.

En la Tabla 12 se muestra la duración final del Sprint a partir de las horas estimadas inicialmente, una estimación ideal y de las horas reales. Aunque han surgido dificultades en este tercer sprint, se ha logrado alcanzar el objetivo dentro de las estimaciones realizadas.

Finalmente, en la Figura 37 se muestra el diagrama *burn down* realizado a partir de la estimación inicial (Sección 5.3). Al igual que el Sprint anterior, consideramos una jornada de 4 horas de trabajo diarias y una duración de 22 días laborales. Como se observa, la tarea 04.02, como hemos dicho, el hecho de haber realizado la conversión de tres maneras distintas del vector *sparse* ha consumido algo de tiempo del que habíamos estimado. El resto del desarrollo del sprint se lleva bajo control como se había estimado en un principio.

ID	Tarea	P.H.	Duración (h)	
			Estimada	**Real**
04.01	Implementar algoritmo *EmbeddingBag* en *STONNE*	10	26	35
04.02	Preparar los datos de entrada para *MLPs* y *EmbeddingBags*	5	13	20
04.03	Conectar *DLRM* con *STONNE* mediante su *API*	10	26	20
04.04	Configurar aceleradores de inferencia	5	13	20
04.05	Comprobar resultados de ejecución	6	15	10
Total		36	93	105

Tabla 12: Duración final Sprint 3.

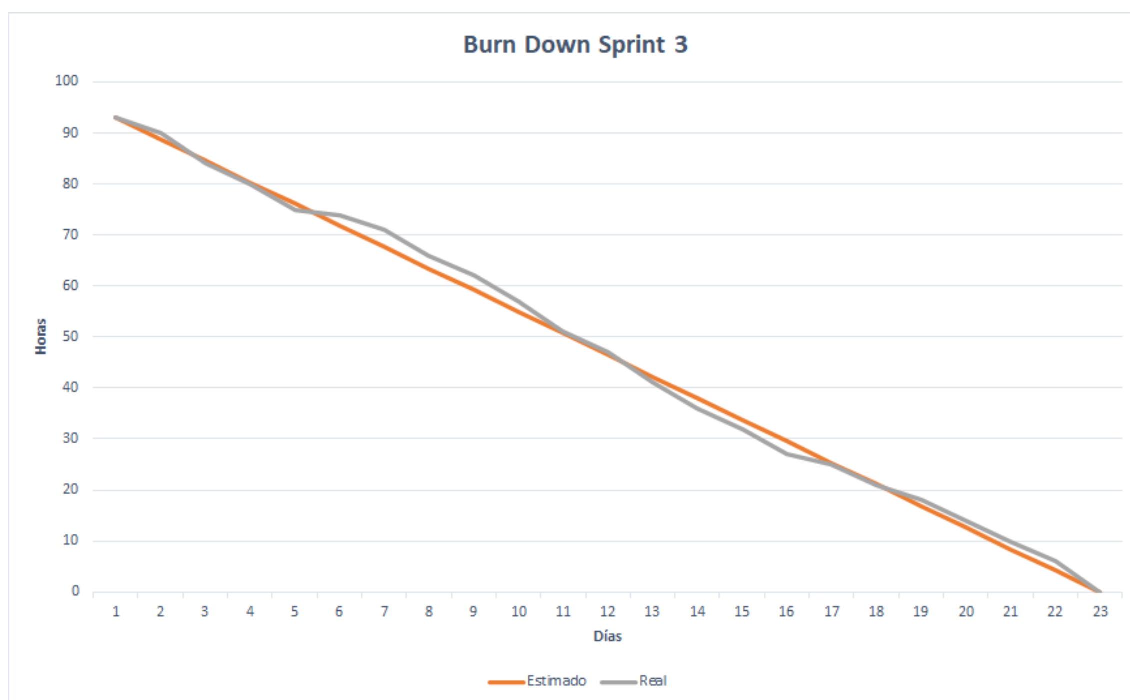

Figura 37: Gráfico de *burn down* para el Sprint 3.

6.4 Sprint 4

En este ultimo sprint del proyecto se realizarán diferentes experimentos sobre el modelo ya implementado en *STONNE*, junto con las arquitecturas aceleradoras.

6.4.1 Sprint planning

En la Figura 38 se muestran las Historias de Usuario seleccionadas, junto con las sub-tareas, que se desarrollarán durante este cuarto Sprint.

! Historia de Usuario 05

Documentación TFG +

Nuevo + Añadir asignado #690830483 por Nicolás M. a las 15:17

27 abr. – 31 may. (25d) 5 subtareas Adjuntar archivos Añadir dependencia Compartido con 1 grupo y 1 persona

Tarea	Fecha	Estado
Hito 05	31 may.	Nuevo
Tarea 05.04 | Valorar y analizar el resultado de las pruebas	28 may.	Nuevo
Tarea 05.03 | Realizar ejecución de pruebas en STONNE	14 may.	Nuevo
Tarea 05.02 | Investigar cargas de trabajo reales/sintéticas	4 may.	Nuevo
Tarea 05.01 | Investigar como realizar cargas de trabajo personalizadas	29 abr.	Nuevo

+ Nueva tarea

Como: Desarrollador
Quiero: Preparar casos de prueba
Para: Caracterizar el trabajo

Figura 38: Historias de Usuario para el Sprint 4.

El sprint goal se puede resumir en, investigar como hacer cargas de trabajo personalizadas en *DLRM* y analizarlas. Estas cargas de trabajo han de ser *datasets* reales o sintéticos. La finalidad de simular cargas de trabajo reales es analizar como el modelo se comporta y obtener datos que nos sirvan de cara a defender la arquitectura que presentamos.

Como primera tarea, 05.01, se trata de investigar como el modelo *DLRM* nativo permite realizar cargas de trabajo personalizadas, es decir, como podemos modificar los parámetros de las *EmbeddingBags*, *MLPs*, tamaños, número de *lookups*, etc.

A continuación, la tarea 05.02. Algunos artículos, como los ya mencionados en la Sección 2.3.3, presentan cargas de trabajo sobre el modelo *DLRM* (*Centaur* y *Facebook Architectural Implications*), se trata de investigar/estudiar que cargas

realizan, el peso que le dan a cada componente, número de ejecuciones que realizan, etc.

Una vez investigadas las cargas de trabajo reales, en la tarea 05.03, vamos a ejecutarlas sobre *STONNE*. Una vez la ejecución ha finalizado, por cada carga de trabajo realizada se confeccionará un gráfico a modo de comparativa (tiempo de ejecución, ciclos, lecturas, escrituras, etc.).

Finalmente, en la tarea 05.04, se valorarán y analizarán los resultados obtenidos con el fin de dar validez al proyecto y de responder a la Sección 7.

En la Figura 39 se muestra el tablero de *Wrike* al comienzo del sprint.

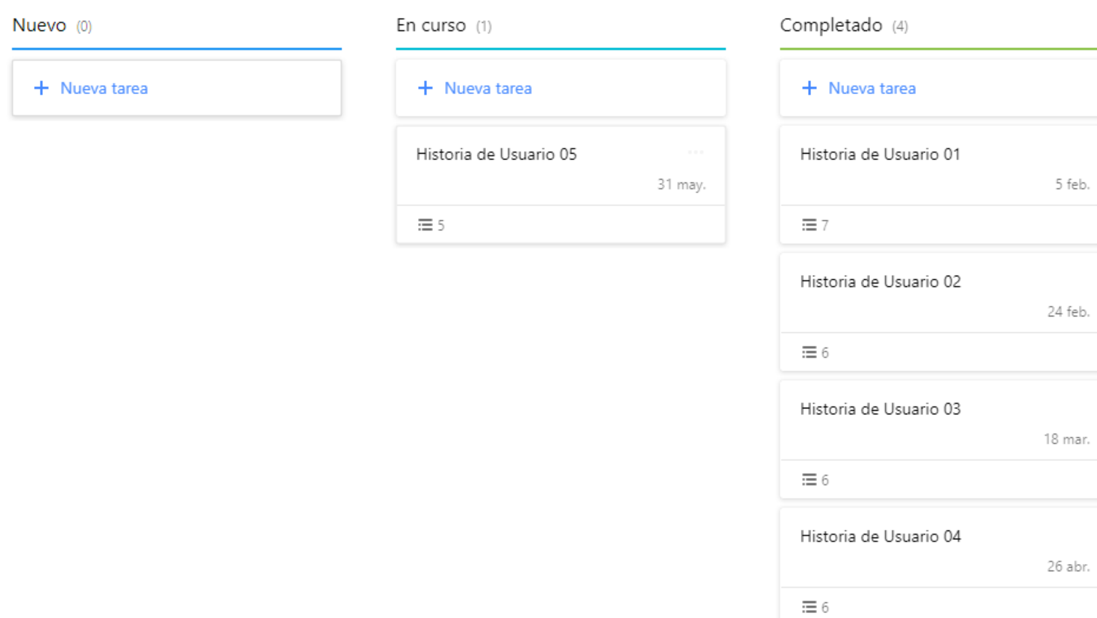

Figura 39: Tablero para el Sprint 4.

6.4.2 Sprint review

Se ha comenzado identificando qué componentes se pueden modificar dentro de cada ejecución de *DLRM*, como sabemos, tenemos las *EmbeddingBags* y las dos *MLPs*, *Bottom-MLP* y *Top-MLP*. Cuando lanzamos la ejecución estándar del modelo, existen algunos parámetros, como los que veremos más adelante, que por defecto tienen un valor predefinido. El modelo cuenta con más de 100 variables que se pueden modificar en cada ejecución. De entre todas estas variables se encuentran algunas relacionadas con el *training* y *GPU*, variables que

no entran dentro del campo de estudio del proyecto.

Las tareas 05.01 y 05.02 van conjuntas, si nos fijamos en el artículo de *DLRM*, al final muestran un ejemplo de ejecución que ellos usan para dar validez al modelo y a los resultados propuestos. Puesto que aparece un ejemplo de ejecución personalizada y con un cierto peso relativo a cargas de trabajo reales, se va a seguir la ejecución que ellos proponen variando algunos parámetros. En los próximos párrafos los detallaremos.

Como hemos comentado anteriormente, en cada ejecución nos interesa dar un peso distinto a los diferentes componentes con el fin de poner de manifiesto la intensidad de cómputo y de memoria que existe en la *MLPs* y *EmbeddingBags* respectivamente. En la Tabla 13 se muestran las variables que vamos a modificar durante los experimentos.

embedding-size	***sparse-feature size***	***mlp-bot***	***mlp-top***	***num-indices lookup***	***data-size***
100-100-100	4	16-8-4	32-16-8-1	100	5

Tabla 13: Variables modificables.

Si nos fijamos en el parámetro *embedding-size*, este, por ejemplo, nos indica que existen 3 *EmbeddingBags* y cada una de estas tiene 100 índices. El parámetro *mlp-bot*, nos indica que la *Bottom-MLP* tiene 3 capas y en cada una de ellas 16, 8 y 4 neuronas respectivamente. Para algunos parámetros vamos a omitir su explicación, pero de esta manera es como vamos a confeccionar todos los casos de estudio que ejecutaremos en *STONNE*. En el Anexo 10.2 se encuentra un manual de uso para poder realizar cargas personalizadas al modelo *DLRM*, también se especifican todos los parámetros y como se puede cambiar/modificar la arquitectura aceleradora para el cómputo *dense* y *sparse* (se recomienda verlo para entender las cargas de trabajo y como afectan unas con otras).

Las principales cargas de trabajo investigadas han sido extraídas de los artículos: *DLRM Architectural Implications* y *Centaur* (ver Sección 2.3.3). Han habido algunos problemas, ya que al estar centrados en el proceso de inferencia, hemos tenido que descartar algunos casos de estudio que se centraban en el *training*. Algo a destacar es la capa final de la *Bottom-MLP* y el valor del parámetro *sparse-*

feature-size, ambos van a tener el valor de 32, ya que en los experimentos y en el artículo de *Centaur* lo resaltan. Presentan la operación *EmbeddingBag* como una operación GEMM sparse compuesta por una matriz de 32-columnas. De igual manera, puesto que en el artículo de DLRM presentan una carga de trabajo con parámetros predefinidos y en el resto de experimentos propuestos no los detallan, hemos usado el valor predefinido para los primeros experimentos, detallaremos más información según nos acerquemos a los últimos casos de estudio.

En total se han obtenido 13 diferentes casos de estudio que se ejecutaran sobre el modelo *DLRM*. Entre estos se pueden diferenciar 4 grandes grupos. Para que experimentos sean lo más variados posibles, se han realizado combinaciones. Las *EmbeddingBags pueden llegar a ocupar GBs de memoria* (Naumov y cols., 2019), es un aspecto que se ha tenido en cuenta en los últimos casos de estudio, el tamaño de la *EmbeddingBags* incrementa desde *MBs GBs*.

El primer grupo de experimentos ha servido como validación para comprobar si el modelo estaba bien implementado y si los resultados obtenidos eran los esperados, nos referiremos a este grupo como grupo de validación. Posteriormente se han usado tres grupos para clasificar los experimentos, el primero de un peso ligero, donde se usan 10, 25 y 50 *EmbeddingBags* respectivamente, manteniendo el parámetro *data-size* con un valor de 20. El segundo grupo representa una carga de peso medio. El número de *EmbeddingBags* sigue siendo 10, 25 y 50 respectivamente, pero el parámetro *data-size* aumenta a 80. Este segundo grupo tiene la finalidad de estudiar como el parámetro *data-size* afecta a las lecturas y escrituras que se producen en el *GlobalBuffer*. Finalmente, el tercer grupo representa una carga de trabajo pesada, donde, el número de *EmbeddingBags* es 50 en todos los casos, aumentamos el *num-indices-lookup* por *EmbeddingBag* considerablemente y alteramos el valor *data-size*. Finalmente, en los dos últimos experimentos de este cuarto grupo modificamos el parámetro *num-indices-lookup* para estudiar su comportamiento sobre el modelo.

Una vez se han obtenido los diferentes casos de estudio estamos listos para abordar la tarea 05.03, ejecutarlos en *STONNE*, en los primeros casos, donde el peso de las *EmbeddingBags* y el cómputo no es muy significativo se pueden

ejecutar de inicio a fin, sin embargo, según nos acercamos a los últimos experimentos el tiempo de ejecución sobrepasa las horas.

Para obtener los parámetros como los ciclos de ejecución, número de lecturas o escrituras se ha utilizado un *script* escrito en *Command Line Interface*, *CLI*. Por cada ejecución *STONNE* devuelve un *JSON*, en este archivo se pueden ver los ciclos, tamaño del *buffer*, escrituras y lecturas del *Global Buffer* y consumo de energía, etc. En caso de ser necesario, los diferentes *scripts* necesarios para leer la salida de los datos se encuentran en el Anexo 10.3.

Comenzamos así las tareas 05.03 y 05.04 de manera conjunta, puesto que hay que obtener la salida de un caso de estudio, estudiarla, analizarla y posteriormente ejecutar la siguiente carga de trabajo.

Los parámetros que no sean mencionados en cada grupo de experimentos asumiremos como valor el que aparece por defecto en el Anexo 10.2, se recomienda encarecidamente revisar este Anexo.

A continuación presentamos el grupo de validación, con los tres primeros casos de estudio (validación), estos han sido extraídos del artículo *Centaur* (Sección 2.3.3). En la Tabla 14 se observan los parámetros para cada ejecución.

Con estos tres primeros casos de estudio queremos ver como se comporta el simulador *STONNE* para ver si los resultados son los esperados y si se corresponden con los presentados en Centaur.

Num Embeddings	Indices/Embedding	data-size	mlp-bot	mlp-top
5	1.000.000	20	128-64-64-32	128-64-1
5	1.000.000	80	128-64-64-32	128-64-1
5	1.000.000	2	512-256-32	256-256-128 128-64-64-1

Tabla 14: Grupo de prueba para evaluar el modelo. *Embedding = EmbeddingBag

En los dos primeros experimentos, el peso de cada *EmbeddingBag* es de 128 *MBs* y el de cada una de las *MLPs* de 57 *KBs*. Mientras que, en el último experimento las *EmbeddingBags* siguen pesando lo mismo, las *MLPs* pasan a ocupar 557 *MBs*. Con este grupo de validación se quiere demostrar como el simulador ejecuta cargas de trabajo *dense* y *sparse* por igual, queremos corroborar

su funcionalidad y efectividad (casos de estudio 1 y 2). Finalmente, se presenta un último caso de estudio (*MLP* de 557 *MBs*) para ver como el modelo propuesto y la arquitectura se comporta.

En la Figura 40 se muestra el número de ciclos consumido por ambos componentes, teniendo en cuenta el término *MLPs* como la suma de ciclos para la *Bottom-MLP* y la *Top-MLP*.

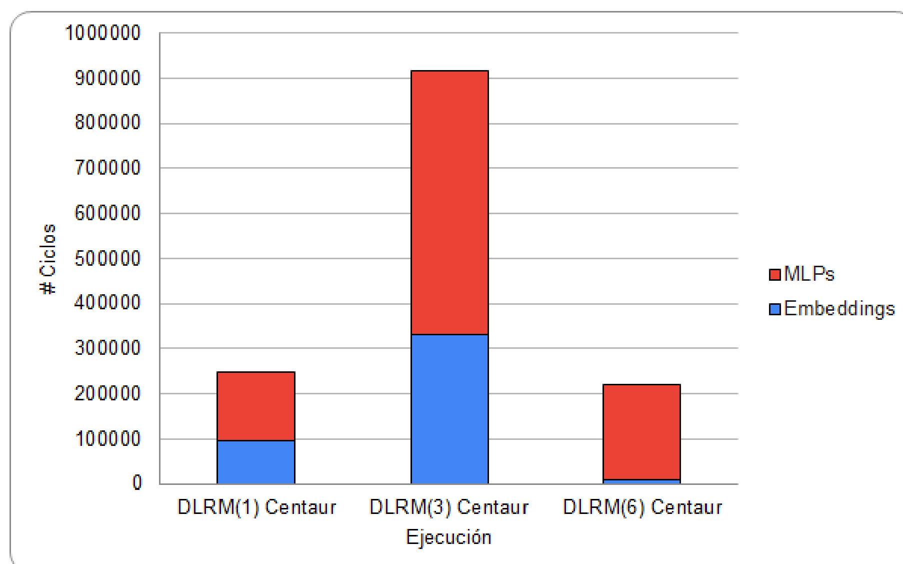

Figura 40: Ciclos de ejecución, grupo evaluación.

A primera vista, podemos observar como las *MLPs* (*Bottom* y *Top*) son intensivas en cómputo (ocupan la mayor parte de los ciclos). Si comparamos estos con los ciclos de ejecución de las *EmbeddingBags* (refiriéndonos al primer y segundo caso de estudio), observamos como toman aproximadamente un 35-40 % del cómputo total, estos números se corresponden con los mostrados en el artículo de *Centaur*. De estos primeros casos de estudio hay un detalle muy importante a destacar, el número de ciclos de la *Top-MLP* va en función del tamaño de las *EmbeddingBags* (número de *EmbeddingBags*, *data-size*, *índices*, etc.). En el primer y segundo caso se puede ver claramente, mientras que en el tercero, al ser la *EmbeddingBag* tan poco intensiva en memoria, el cómputo de la *Top-MLP* es significativamente inferior que en el resto de los casos.

Vamos a introducir dos gráficas, para entender como afecta una determinada carga de trabajo a las lecturas y/o escrituras en cada caso. En la Figura 41 se

pueden ver las lecturas que ha tomado cada una de las partes y en la Figura 42 las escrituras.

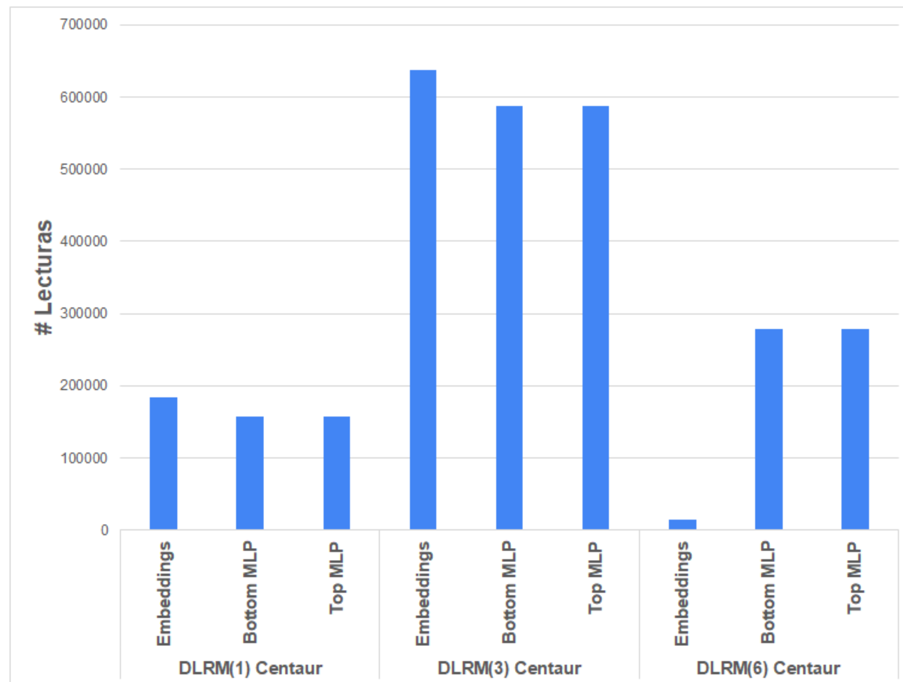

Figura 41: Número de lecturas, grupo evaluación.

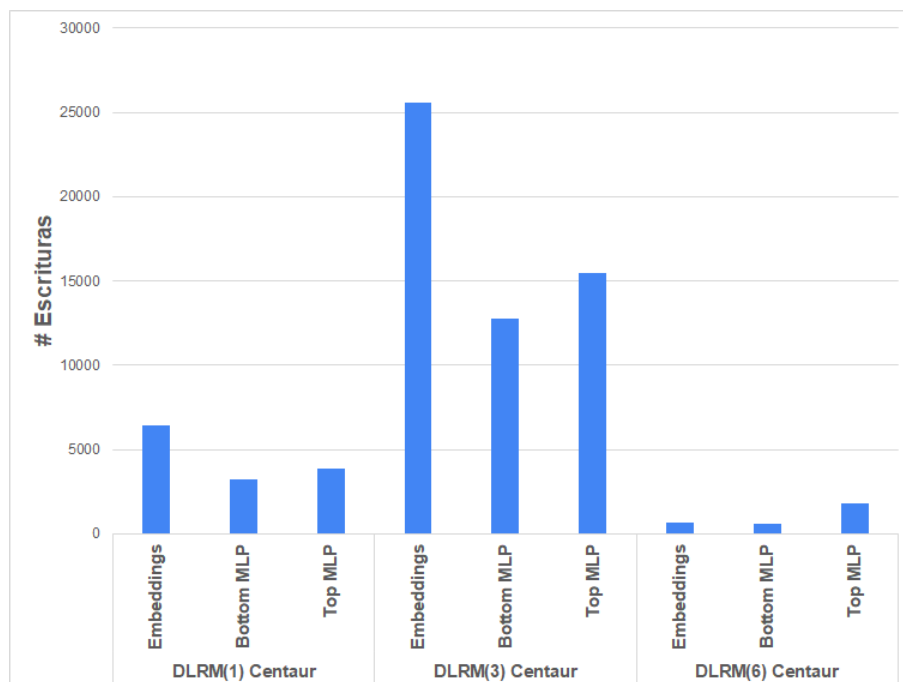

Figura 42: Número de escrituras, grupo evaluación.

Para entender estas gráficas, vamos primero a explicar los dos primeros casos de estudio referentes a las lecturas (Figura 41). Si nos fijamos en la Tabla 14,

tanto la *Bottom-MLP* como la *Top-MLP* tienen el mismo tamaño. Entonces, ¿por qué no se obtienen las mismas lecturas en ambos? La *Top-MLP* sabemos que post-procesa la matriz resultante de la fase de *Feature-Interaction*, por lo que un cambio de tamaño en la dimensión de las *EmbeddingBags* resultaría en un incremento del número de lecturas. Además, recordemos que el modelo DLRM añade por debajo una capa extra a esta *Top-MLP*, por lo que al aumentar o disminuir el número de *EmbeddingBags*, aunque mantengamos el resto de parámetros por defecto entre experimentos, las lecturas aumentarán.

Para justificar el caso de las lecturas en la *Bottom-MLP* es un poco más complejo, ya que para poder concatenar la salida de la *Bottom-MLP* con la salida de las *EmbeddingBags* el vector resultante ha de tener el mismo tamaño que la matriz. Por ende, si se realizan 20 *búsquedas/embedding* (parámetro *data-size*) dará como resultado un vector de menor tamaño (menos lecturas) que si se realizan 80.

En ambos casos de estudio se realizan un número muy similar de lecturas en la *Bottom-MLP* y *Top-MLP* y esto ha sido mera casualidad. Debido al poco peso/tamaño de las *EmbeddingBags*, el número de lecturas que se realizan en la *Top-MLP* es prácticamente similar al de lecturas en la *Bottom-MLP*. Al haberse usado el mismo número de *EmbeddingBags* en los dos casos de estudio, y solamente aumentado las *lookups/embeddingBag* las lecturas de la *Bottom* y *Top MLP* son similares. A lo largo de esta caracterización presentaremos diferentes gráficas de lecturas donde se podrá observar la verdadera diferencia entre ambas *MLPs*.

Las lecturas del componente *EmbeddingBags* dependen de varios factores, número de *EmbeddingBags*, número de índices, *num-indices-lookup* y *data-size*. Puesto que el parámetro *data-size* difiere en los tres experimentos, el número de lecturas es diferente en cada caso.

Para concluir con la gráfica de las lecturas vamos a introducir el tercer experimento, *DLRM(6) Centaur*. El número de lecturas de las*EmbeddingBags* es significativamente inferior al de las *MLPs*. Si nos fijamos en casos de estudio anteriores (menor tamaño de *MLPs*), el cómputo de las *MLPs* estaba guiado por el

número de lecturas de las *EmbebddingBags*. Sin embargo, este tercer caso está orientado a ver como el modelo y simulador se comportan ante grandes cargas de cómputo. A pesar de obtener una salida prácticamente insignificante para el tamaño de las *MLPs*, el número de lecturas de ambas *MLPs* es algo a tener en cuenta.

De esta manera la gráfica de las lecturas queda entendida y explicada, para los experimentos que hagamos posteriormente es vital haber entendido como el componente *EmbeddingBags* afecta a las *MLPs*.

Para la gráfica de las escrituras se puede seguir el mismo punto de vista que se ha realizado con las lecturas. Para el primer caso de estudio, tenemos un tamaño de *EmbeddingBag* de *128 MBs*, el mismo que para el segundo caso. Entonces, ¿por qué hay tanta diferencia de escrituras? en el primer caso se realizan 20 búsquedas, parámetro *data-size*, mientras que en el segundo se realizan 80. De esta gráfica podemos aprender como este parámetro es un valor significativo de cara a las escrituras y a tener en cuenta para el cómputo *sparse*. En experimentos posteriores alteraremos este valor para observar como el modelo se comporta.

Relacionado con los dos primeros casos de estudio, si ahora nos fijamos en las escrituras de la *Bottom-MLP* y *Top-MLP*, siempre las escrituras de la *Bottom-MLP* van a ser inferiores, esto se debe a que la *Top-MLP* recibe la matriz resultante de concatenar *Bottom-MLP* y *EmbeddingBags*, por lo que el número de escrituras siempre va a ser mayor. Además, como ya mencionamos en el Anexo 10.2 y mencionaremos a lo largo de este Sprint, el modelo DLRM añade una capa extra a la *Top-MLP*, es por eso, que a pesar de tener el mismo tamaño inicial, en tiempo de ejecución la *Top-MLP* tendrá un mayor impacto que la *Bottom-MLP*. Como apunte final, destacar que entre los dos primeros experimentos las escrituras de las *MLPs* son diferentes, ya que el tamaño de las *EmbeddingBags* es diferente y esto condiciona a la *Bottom-MLP* y *Top-MLP*.

El último caso de estudio (tamaño de MLPs muy superior a las *EmbeddingBags*) sigue lo esperado hasta ahora, puesto que el tamaño de las *EmbeddingBags* es tan pequeño, el número de escrituras a memoria que se realizan es prácticamente insignificante. Mientras que, las escrituras que se realizan en la

Top-MLP presentan una mayor carga de cómputo.

Este último experimento, a parte de servir para comprobar la funcionalidad y eficiencia, nos sirve para demostrar el peso de las *EmbeddingBags* en todo el modelo. Aunque se usen *MLPs* de hasta *557 MBs*, como en este caso, estas no representan una carga de trabajo significativa si las *EmbeddingBags* son tan pequeñas. Sin embargo, en el segundo caso de estudio, donde usamos *MLPs* de inferior tamaño y aumentamos considerablemente el peso de las *EmbeddingBags* vemos que el número de escrituras y lecturas es considerablemente mayor.

El tamaño de las *EmbeddingBags* resulta en tiempos de cómputo, E/L a memoria y número de ciclos muy dispares. Los experimentos que presentamos a continuación están orientados a mostrar como el modelo se comporta cuando realizamos cargas significativamente superiores a las que hemos visto hasta el momento, especialmente al aumentar el peso del componente *EmbeddingBag*.

A continuación se introducen el resto de grupos con sus respectivos casos de estudio, puesto que son casos de cara a la investigación y análisis del modelo que presentamos, vamos a usar nombres descriptivos y únicos para cada uno de ellos, además algunos parámetros tendrán valores fijos por defecto (ver Anexo 10.2).

Las *MLPs* van a mantener el tamaño en todos los casos de estudio que presentamos, como ya hemos visto con el grupo de evaluación la mayor o menor intensidad de cómputo de las *MLPs* viene condicionada por el tamaño de *EmbeddingBags* que se use. Debido a este razonamiento hemos decidido mantener un valor fijo en las *MLPs*.

A continuación mostramos el sistema de nombres que vamos a usar durante los siguientes casos de estudio:

NumEmbeddings–Índices/Embedding–{data-size}–{num-indices-lookup}

Como por ejemplo: 10-1M-20-100, lo cual nos indica que existen 10 *EmbeddingBags*, cada una con 1 millón de índices, van a realizarse 20 búsquedas en cada *EmbeddingBag* y en cada una van a seleccionarse 100 índices como máximo.

En los grupos 1 y 2, el peso de cada *EmbeddingBag* es de 128 *MBs* y el de cada una de las *MLPs* de 57 *KBs*. En el último grupo, las *EmbeddingBags* aumentan a 1,28 *GBs* mientras que las *MLPs* mantienen su tamaño inicial. Con este último experimento queremos poner a prueba el modelo DLRM para ver como se comporta ante cargas de trabajo similares a las que podrían ejecutarse en un entorno real (ver Tabla 15).

Una vez entendidas las cargas de trabajo presentadas anteriormente, vamos a comenzar a evaluar diferentes casos de estudio para ver como el modelo evoluciona ante diferentes cargas de trabajo (en el Anexo 10.4 se encuentra la lista completa de los experimentos). En este primer grupo vamos a aumentar considerablemente el número de *EmbeddingBags* en cada caso de estudio, comenzaremos por 10 y acabaremos con 50. Este primer grupo está orientado a ver como el modelo se comporta cuando aumentamos el número de tablas *EmbeddingBags*.

En la Figura 43 podemos ver el número de ciclos consumidos por cada uno de los casos de estudio.

Entendiendo el sistema de nombres introducido anteriormente, se observa que cada caso de estudio tiene más *EmbeddingBags* que el anterior. El número de ciclos consumidos por las *EmbeddingBags* es de al rededor del 40-45 %.

De este primer grupo, tal y como aprendimos con el grupo de evaluación, al aumentar el tamaño de las *EmbeddingBags*, los ciclos que consumían las *MLPs* aumentan notablemente.

Grupo	**Ejecución**	**Tam. *EmbeddingBag***
1	10-1M-20-100	128 *MBs*
	25-1M-20-100	128 *MBs*
	100-1M-20-100	128 *MBs*
2	10-1M-80-100	128 *MBs*
	25-1M-80-100	128 *MBs*
	50-1M-80-100	128 *MBs*
3	50-10M-20-100	1,28 *GBs*
	50-10M-80-100	1,28 *GBs*
	50-10M-20-1000	1,28 *GBs*
	50-10M-80-1000	1,28 *GBs*

Tabla 15: Cargas de trabajo sobre el modelo DLRM.

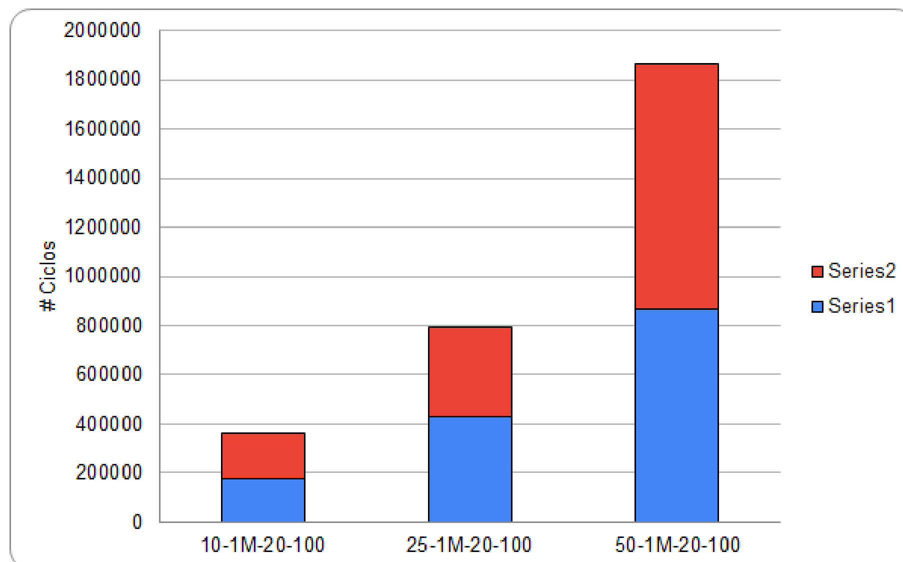

Figura 43: Ciclos de ejecución, grupo 1.

En las Figuras 44 y 45 se muestran las lecturas y escrituras, respectivamente.

En este primer grupo de experimentos de carga ligera vamos a aumentar considerablemente el número de *EmbeddingBags*, comenzaremos por 10 y acabaremos con 50. Nos centraremos en observar como el modelo se comporta cuando este parámetro aumenta y mantenemos el resto estáticos.

Como se observa, las lecturas de la *Bottom-MLP* son las mismas para los tres

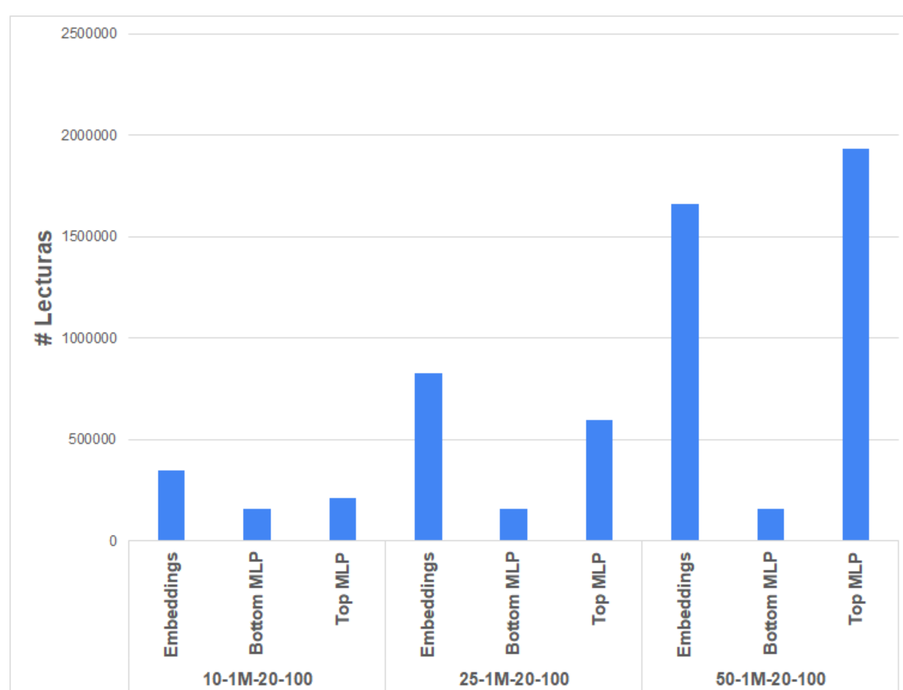

Figura 44: Número de lecturas, grupo 1.

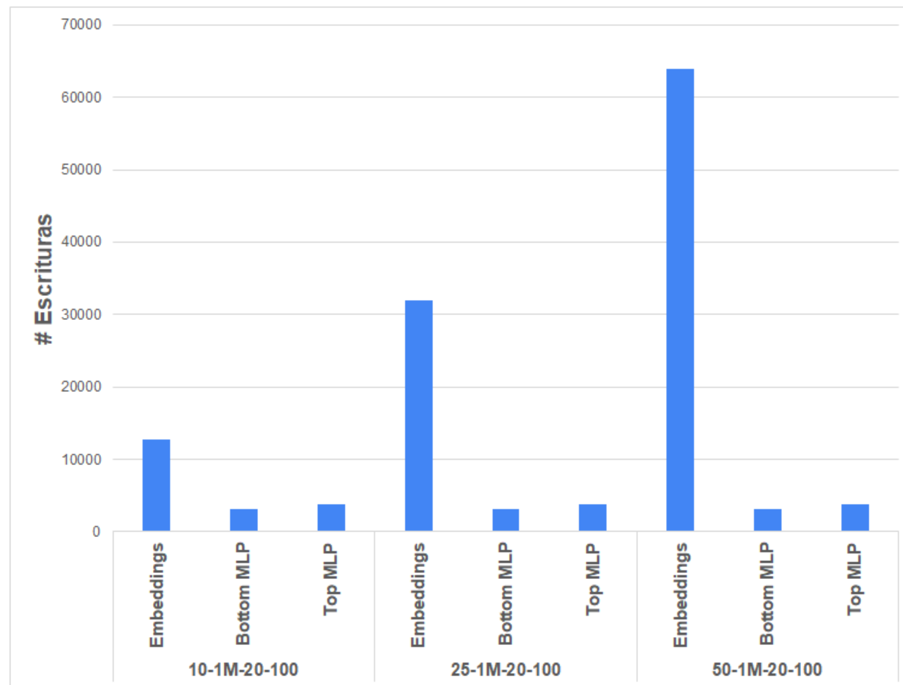

Figura 45: Número de escrituras, grupo 1.

experimentos. Esto se debe a los parámetros *mlp-bot* y *data-size*, puesto que en este primer grupo estos valores no se modifican, el número de lecturas para la *Bottom-MLP* va a ser el mismo en los tres experimentos. Con esto ponemos de manifiesto la relación que existe entre estos parámetros para determinar el número de lecturas de la *Bottom-MLP*.

Sin embargo, las lecturas de la *Top-MLP* sí varían en los tres experimentos. A pesar de especificar un tamaño fijo para la *Top-MLP*, si recordamos, el propio modelo DLRM realiza una operación en la que teniendo en cuenta el número de *EmbeddingBags* y las neuronas de la última capa de la *Bottom-MLP* añade una capa extra a la *Top-MLP*. Como en los tres casos aumentamos el número de *EmbeddingBags*, las neuronas de esta capa extra es diferente en los tres casos, como resultado, el número de lecturas es diferente.

Además, hay que tener en cuenta que el cómputo que ocurre en esta *Top-MLP* viene condicionado por el resultado de la fase anterior, *Feature-Interaction*, como en esta fase se procesa la matriz resultante del algoritmo *EmbeddingBag*, al aumentar el número de *EmbeddingBags* o *data-size*, se aumenta el tamaño de la matriz resultante, por ende, las lecturas que se realizan en la *Top-MLP* también aumentan.

En el caso de las escrituras, el mayor número se encuentra en las *EmbeddingBags*, se observa que al aumentar el número de tablas, aumentan las escrituras. Al aumentar el número de *EmbeddingBags* se realizarán más operaciones *GEMM sparse*, como consecuencia, se generarán más escrituras.

En el caso de la *Bottom-MLP* y *Top-MLP*, ambos componentes realizan el mismo número de escrituras en los tres experimentos. A pesar de añadir una capa extra y diferente a la *Top-MLP* en cada experimento, esta realiza las mismas escrituras. Cuando presentemos el siguiente grupo analizaremos los resultados y los compararemos con los obtenidos.

Teniendo este primer grupo en mente, vamos a introducir el grupo 2 con una carga de trabajo media. Se va a seguir la misma práctica que en el grupo 1, vamos a ir aumentando el número de *EmbeddingBags*, comenzando por por 10 y terminando con 50. Sin embargo, en vez de tener un valor de 20 para el parámetro *data-size*, vamos a aumentarlo a 80 (4x). Es decir, en vez de realizar 20 búsquedas por *EmbeddingBag*, vamos a realizar 80.

En la Figura 46 podemos ver el número de ciclos consumidos por cada uno de los casos de estudio.

A priori, podemos fijarnos en que la escala de la gráfica de ciclos aumenta desde 2 millones hasta 8 millones, aumenta 4 veces. ¿Casualidad que aumentemos en 4 las búsquedas por *EmbeddingBag* (de 20 a 80)? Vamos a ver que

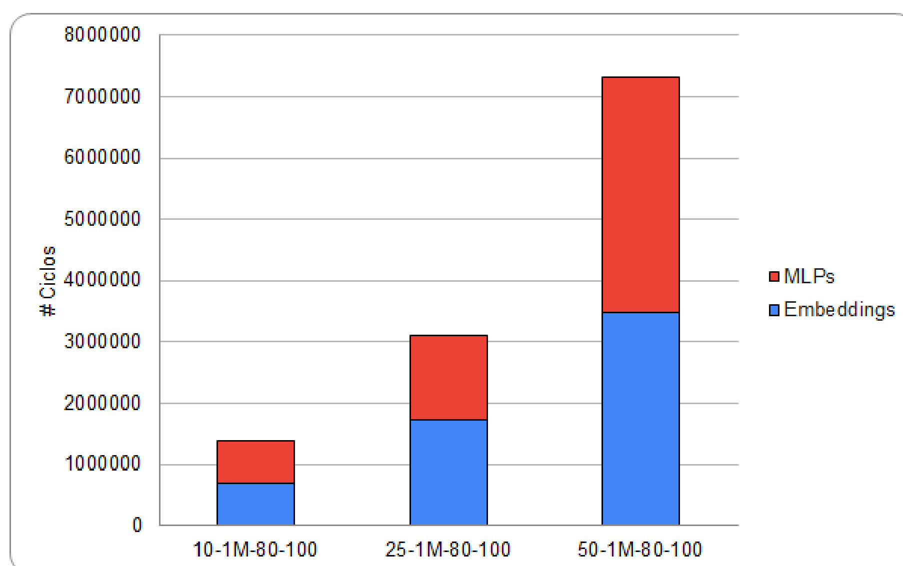

Figura 46: Ciclos de ejecución, grupo 2.

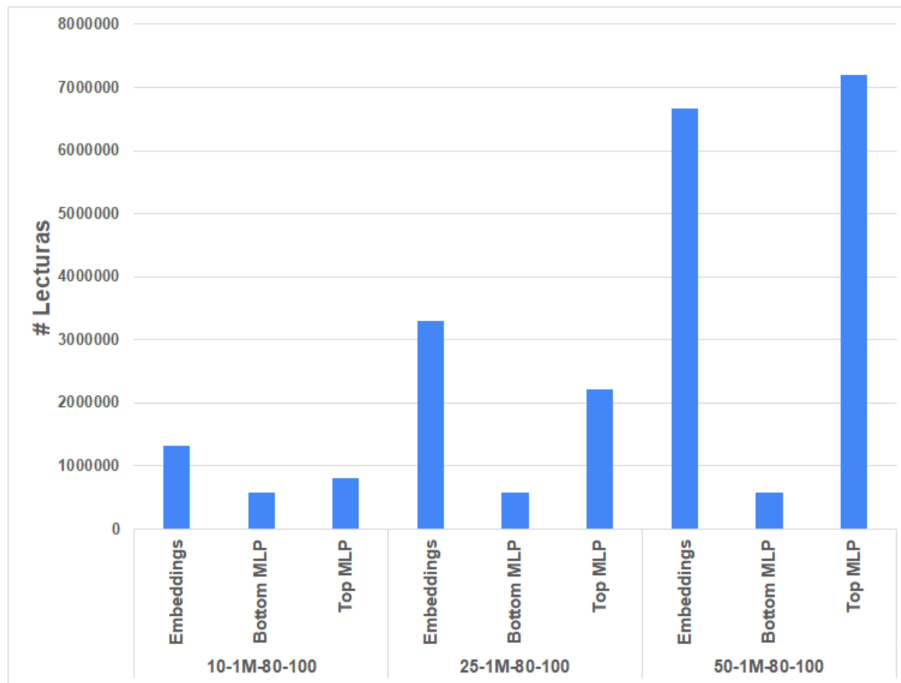

Figura 47: Número de lecturas, grupo 2.

ocurre con las lecturas y escrituras.

En las Figuras 47 y 48 se muestran las lecturas y escrituras, respectivamente.

En el caso de las lecturas de la *Bottom-MLP*, si aplicamos la teoría que hemos desarrollado, puesto que este vector tiene que tener la misma dimensión

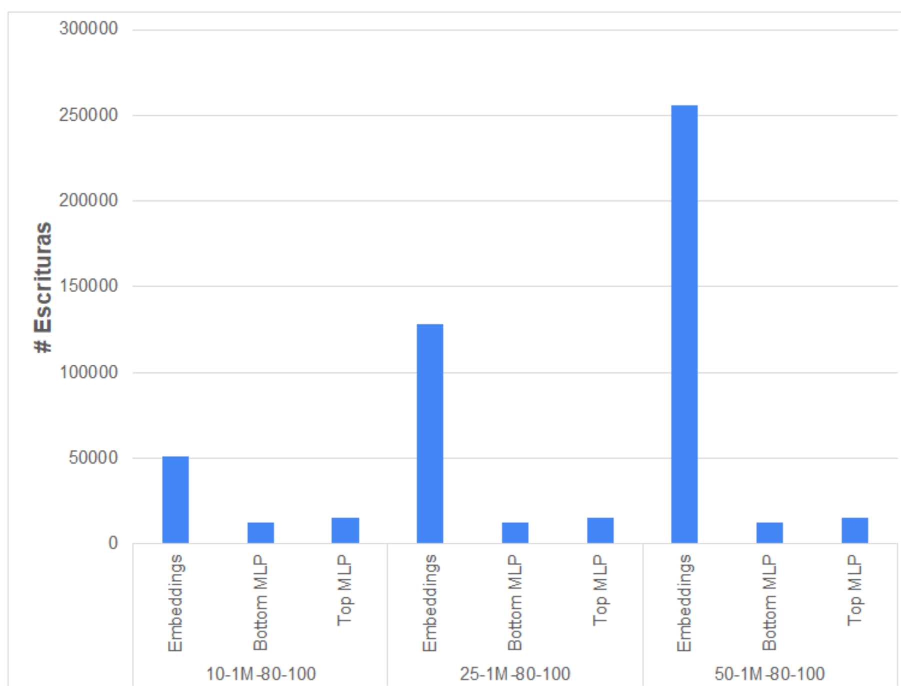

Figura 48: Número de escrituras, grupo 2.

(índices) que la matriz resultante de las *EmbeddingBags* (columnas). Al aumentar el número de búsquedas por *EmbeddingBag*, el vector dense que genera la *Bottom-MLP* ha de ser mayor. Concluyendo en que, al aumentar el parámetro *data-size* de 20 a 80 (4x) el número de lecturas de la *Bottom-MLP* aumenta en el mismo grado (4x) en los tres experimentos de este segundo grupo. El aumento no es exactamente lineal puesto que en cada *lookup* se selecciona un numero diferente de índices (ver parámetro *num-indices-lookup* en Anexo 10.2).

De este modo corroboramos que las lecturas asociadas a este componente *Bottom-MLP* van en función de los parámetros *data-size* y *mlp-bot*.

Para las lecturas de las *EmbeddingBags*, al mantener el resto de parámetros estáticos y realizar 4 veces más búsquedas por *EmbeddingBag* el número de lecturas aumenta notablemente y sigue un crecimiento similar a los resultados obtenidos en las lecturas de las *EmbeddingBags* del grupo 1 (Figura 44).

Para el caso de la *Top-MLP* seguimos en la línea con lo ya comentado anteriormente. Al no modificar el número de *EmbeddingBags* en cada experimento y mantener el mismo tamaño de *Bottom-MLP* esta capa extra que añade el modelo DLRM a la *Top-MLP* no se modifica y se aprecia un crecimiento similar al obtenido en el grupo 1.

Para el caso de las escrituras, como el número de *EmbeddingBags* se mantiene entre ambos grupos el número de escrituras va a ser el mismo, con la peculiaridad de, al realizar 4 búsquedas más, el número de escrituras crece exactamente 4 veces más respecto a los resultados obtenidos del grupo 2.

Esta aproximación se puede seguir con el resto de componentes, *Bottom-MLP* y *Top-MLP* que ocurre de igual manera, las escrituras que se realizan en estos componentes crece de manera lineal con las escrituras obtenidas en la fase anterior. Con esto concluimos que las lecturas que se realizan en la *Bottom-MLP* y *Top-MLP* vienen condicionadas por el tamaño (parámetros *mlp-bot* y *mlp-top*) y el *data-size* de cada ejecución.

Estos experimentos nos sirven para darnos cuenta de como el parámetro *data-size* ó búsquedas/*embedding* afecta al cómputo del modelo.

Hasta ahora hemos visto que los ciclos de ejecución que consumen las *MLPs*

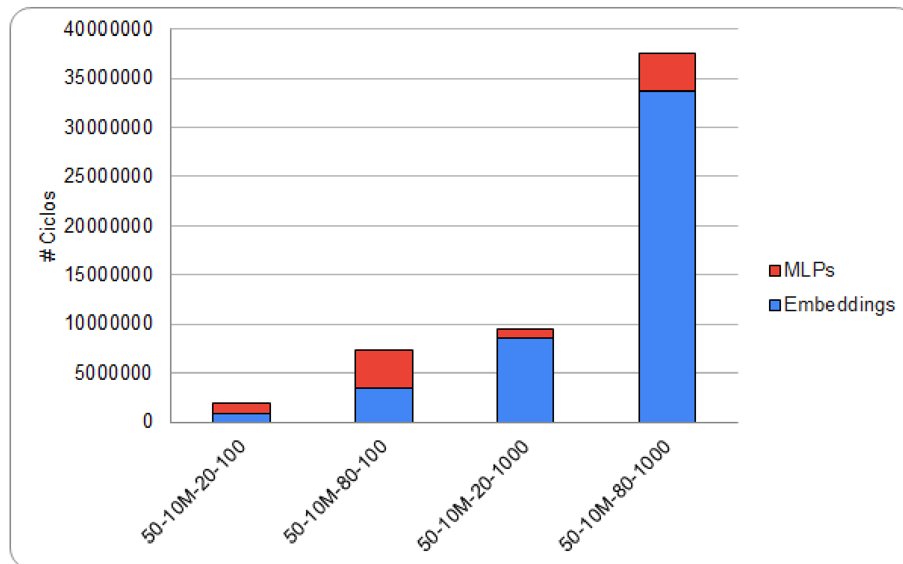

Figura 49: Ciclos de ejecución, grupo 3.

siempre solía ser mayor que el de las propias *EmbeddingBags*, y esto tiene sentido, ya que las *MLPs* son intensivas en cómputo. Sin embargo, hacemos especial atención a la necesidad de soportar grandes cargas de datos. Para ver como el modelo se comporta introducimos último grupo.

A parte de incrementar notablemente el tamaño de las tabla *EmbeddingBags*, vamos a modificar los *num-indices-lookup*, que hasta ahora habían tenido siempre un valor por defecto de 100.

En la Figura 49 podemos ver el número de ciclos consumidos por cada uno de los casos de estudio que vamos a presentar a continuación.

Este tercer grupo de casos de estudio presenta la mayor carga de cómputo para las *EmbeddingBags*, hemos aumentado 10 veces los índices de cada embedding, por lo que en vez de ocupar 128 *MBs*, ocupan 1,28 *GBs*. El resto de parámetros no se especifican puesto que pueden ser leídos desde la gráfica, con el sistema de nombres que presentamos párrafos atrás.

Si nos fijamos en las dos primeras ejecuciones, a pesar de haber aumentado 10 veces el número de índices de las *EmbeddingBags* estas siguen sin suponer un gran porcentaje de los ciclos totales de cómputo (ocupan en torno al 46 y 47 % de los ciclos totales). Este porcentaje de cómputo es el mismo que el obtenido en anteriores experimentos a pesar de aumentar las *EmbeddingBags* a 1,28 *GBs* (Figura 43 y 46).

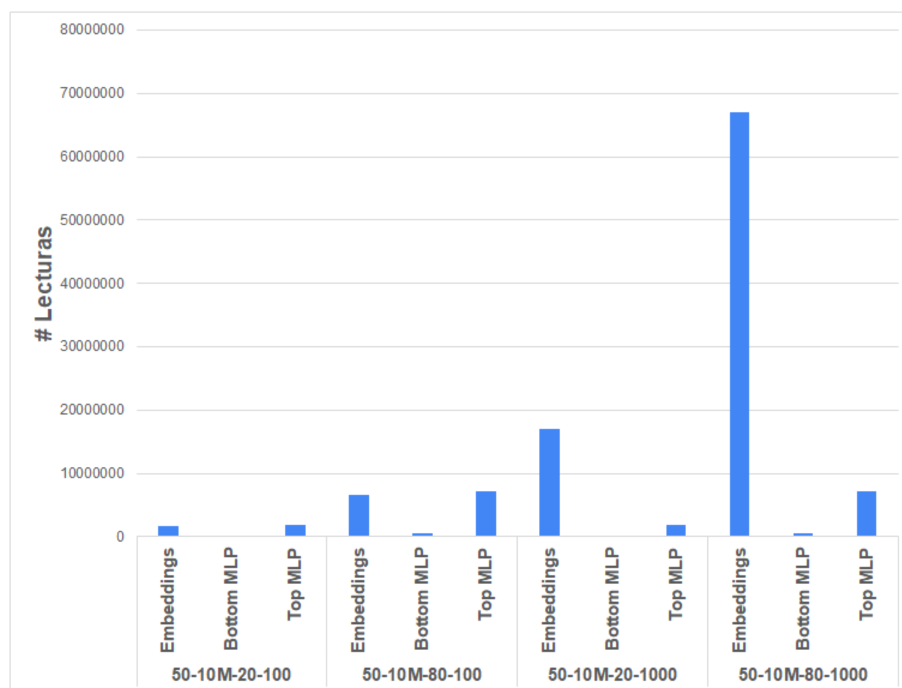

Figura 50: Número de lecturas, grupo 3.

En este punto introducimos el parámetro *num-indices-lookup*. Hasta ahora, para tablas de hasta 1 millón de índices este valor ha sido siempre de 100 unidades, lo que significa, que por cada *lookup* que se realiza en la *EmbeddingBag* como máximo se pueden coger 100 índices (0.0001 % de 1 millón de índices). Con el fin de realizar una mejor predicción, tendríamos que coger más índices, lo que se refleja en realizar más *lookups*. En los últimos dos experimentos se ha incrementado este número a 1000 y ha supuesto un impacto en el porcentaje de ciclos totales consumidos por las *EmbeddingBags* de un 89 % y 87 % respectivamente.

En las Figuras 50 y 51 se muestran las lecturas y escrituras de cada componente.

Basándonos en todo lo explicado anteriormente, vamos a explicar como este parámetro afecta al cómputo del modelo. Las lecturas de la *Bottom-MLP* y *Top-MLP* son iguales para el primer y tercer experimento y para el segundo y cuarto respectivamente. Si nos fijamos, se mantienen los mismos parámetros que para el resto de experimentos (experimento 3 de los anteriores dos grupos), solo variamos el parámetro *num-indices-lookup*. Concluimos con las lecturas diciendo que este parámetro no afecta en el cómputo de las *MLPs*.

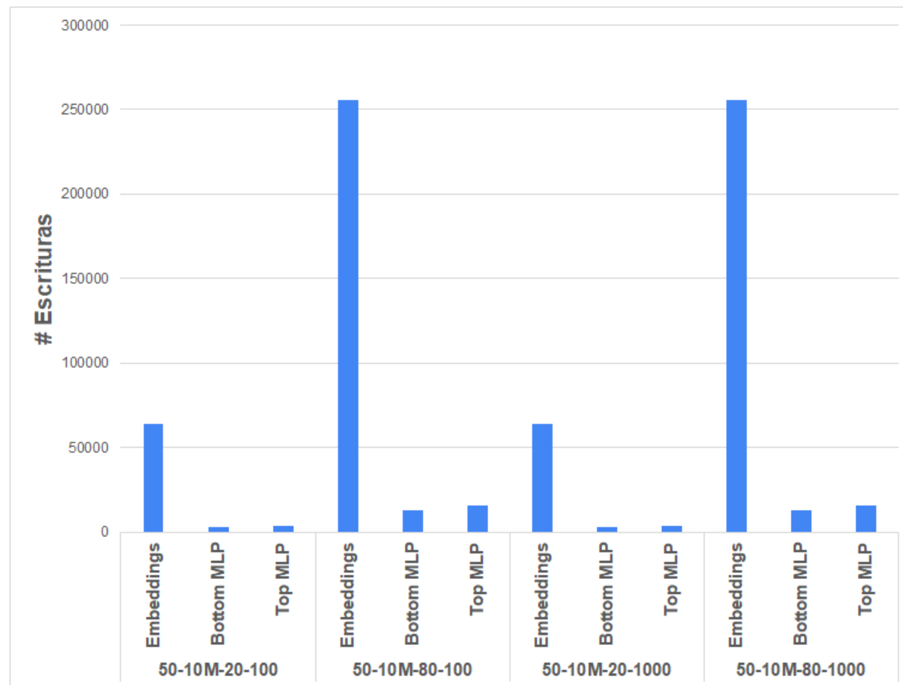

Figura 51: Número de escrituras, grupo 3.

Si nos fijamos en las *EmbeddingBags*, al aumentar el número de índices de las *EmbeddingBags* y el parámetro *num-indices-lookup*, todo esto acompañado de aumentar el parámetro *data-size*, número de *EmbeddingBags*, etc. se observa un gran aumento en el número de lecturas, aproximadamente en un factor de x10, puesto que hemos aumentado el parámetro *num-indices-lookup* en la misma relación. Este parámetro genera un fuerte impacto en el cómputo de las *EmbeddingBags* y el número de lecturas.

En el caso de las escrituras, seguimos en línea con lo comentado anteriormente. El número de escrituras de las *MLPs* viene determinado por el parámetro *data-size*, *mlp-bot* y *mlp-top*. Puesto que no se modifican, se obtienen las mismas escrituras que en los experimentos anteriores. Las escrituras de las *MLPs* del experimento 1 y 3 de este grupo, se corresponden con las obtenidas en el grupo 1; y los experimentos 2 y 4 con las obtenidas en el grupo 2.

Para las escrituras de las *EmbeddingBags* ocurre algo peculiar. Las escrituras en los experimentos 1 y 3 de este grupo 3, se corresponden con las obtenidas en el experimento 3 del grupo 1 (mismo valor del *data-size*, 20, y mismo número de *EmbedingBags*, 50). Por otra parte, las escrituras obtenidas en los experimentos 2 y 4 de este grupo (los más intensivos en cómputo), se corresponden con las

que se han obtenido en el experimento 3 del grupo 2 (mismo *data-size* y mismo número de *EmbeddingBags*).

Tras esta densa explicación presentamos el resumen de las conclusiones obtenidas tras haber realizado los experimentos. Vamos a mostrar el impacto de los diferentes parámetros en el cómputo final del modelo.

- Si comparamos el tiempo total empleado por los diferentes experimentos, el componente más complejo, donde más tiempo se ha empleado ha sido en el procesamiento de las *EmbeddingBags*.

- En el caso de las lecturas, sabemos que el mayor o menor número de lecturas en las *EmbeddingBags* viene condicionado en función del número de *EmbeddingBags*, de los índices de cada *EmbeddingBag*, del parámetro *data-size* y del parámetro *num-indices-lookup*.

- En el caso de la *Bottom-MLP*, las lecturas vienen determinadas por el parámetro *data-size* y por el tamaño del parámetro *mlp-bot*, que determina el número de capas y las neuronas por capa.

- Para la *Top-MLP*, recordemos que el modelo añade una capa extra previa a las que añadimos nosotros. De esta manera, las lecturas vienen condicionadas por, el número de *EmbeddingBags*, las neuronas de la última capa de la *Bot-MLP* y el parámetro *data-size*. Obviamente, dejando estos parámetros estáticos y modificando el valor del parámetro *mlp-top* también se verá reflejado un aumento o disminución en las lecturas.

- El número de escrituras en las *EmbeddingBags* viene determinado por el número de *EmbeddingBags* que existan y por el parámetro *data-size*. El aumento del parámetro *num-indices-lookup* ha demostrado no suponer ningún tipo de impacto en las escrituras.

- Para el caso de las *MLPs* es mas sencillo, puesto que las escrituras vienen determinadas por el número de capas, es decir, *mlp-{top, bot}* y el parámetro *data-size*.

Concluimos así con las tareas 05.04 y 05.05, esto nos sirve para dar pie a la conclusión que se presenta en la Sección 8.

Como aporte final, queremos incluir todas las ejecuciones que hemos realizado, junto con todos los parámetros que hemos modificado. De cara a si alguien quiere repetir los experimentos, no encuentre limitaciones para entender como los parámetros han sido modificados. Esto es algo novedoso ya que muy pocos artículos comparten y publican de manera completa las ejecuciones que han realizado. Esto se puede encontrar en el Anexo 10.4.

Con todo el trabajo que se ha desarrollado hasta la fecha y las pruebas obtenidas se ha redactado un artículo científico para el congreso “Sociedad de Arquitectura y Tecnología de Computadores” (SARTECO, 2021). Este artículo se encuentra en el Anexo 10.5.

6.4.3 Sprint retrospective

La tarea, 05.02 se ha retrasado ya que en las investigaciones que se han realizado para encontrar cargas de trabajo, muchas de ellas no especifican el valor de todos los parámetros que usan para la ejecución. Como consecuencia se ha asumida un valor por defecto para aquellos parámetros que no mencionan, como por ejemplo, el *num-indices-lookup*. Ya que la carga de trabajo presentada por el artículo *DLRM* fija este valor a 100, para todos los experimentos se ha asumido el mismo valor, salvo en los últimos casos, donde ponemos de manifiesto la intensidad en memoria que requieren las *EmbeddingBags*.

De igual manera ocurre en algunos experimentos con el *data-size*, que india el número de *lookups* por *EmbeddingBag*. Para obtener casos de estudio lo más variados posibles varios valores, entre los cuales se encuentra el *data-size*, se han combinado de diferentes maneras.

Respecto a la ejecución de los casos de estudio en *STONNE*, 05.03 puesto que algunas cargas de trabajo llegaban a *GBs* de memoria, se ha optado por hacer uso del software *Screen*, que permite ejecutar en segundo plano una consola de comandos. De esta manera se puede ejecutar la simulación y cerrar el cliente *SSH*.

Un problema que se ha tenido que afrontar tiene que ver con *STONNE*. Puesto que simula arquitecturas aceleradoras y cada una devuelve un *output* en formato *JSON*, en algunas cargas, al ser cargas tan pequeñas, las salidas colisionaban y hacían que se sobreescribiesen. Para poder solucionar este problema se ha añadido un *delay* de 1 segundo, mediante *time.sleep*, otra librería de *Python*, para así evitar este problema.

El apartado 05.03 como ya se ha comentado anteriormente, se ha hecho uso de un *script* escrito en *CLI* para poder leer todos los ficheros output devueltos por *STONNE*. La creación de este script también ha supuesto una carga de trabajo extra.

También se han tenido que desarrollar múltiples casos de estudio diferenciados en diferentes grupos con el fin de demostrar el impacto del modelo y de la arquitectura que presentamos. Los experimentos se han tenido que repetir varias veces, ya que tuvimos que sacar múltiples datos de todos los experimentos cuando ya los habíamos borrado.

Finalmente destacar el hecho de escribir un artículo para las jornadas de *SARTECO*. Esto ha tomado una gran parte del tiempo inesperada y como consecuencia se ha tenido que añadir horas extras a la tarea 05.04, dentro de la cual situamos la tarea de escribir el artículo.

En la Tabla 16 se muestra la duración final del Sprint a partir de las horas estimadas inicialmente, una estimación ideal y de las horas reales. Como ya hemos mencionado, el escribir el artículo ha tomado una gran parte del tiempo, aún así al tratarse del último sprint y al tener fecha límite de entrega para el proyecto se han tenido que dedicar horas extra.

Finalmente, en la Figura 52 se muestra el diagrama *burn down* realizado a partir de la estimación inicial (Sección 5.3). Consideramos una jornada media, aunque las últimas dos semanas se trabaja un tiempo muy superior a 4 horas diarias con el fin de alcanzar el objetivo. Tal y como se observa en la Tabla 16, la última tarea que se planeaba para las dos últimas semanas se dedicó también al desarrollo del artículo ya mencionado. Ha habido un incremento de 73 horas de trabajo que se han tenido que abordar en este último sprint.

ID	Tarea	P.H.	Duración (h)	
			Estimada	**Real**
05.01	Investigar como realizar trabajo reales/sintéticas	7	18	23
05.02	Investigar cargas de cargas de trabajo personalizadas	5	13	25
05.03	Realizar ejecución de pruebas en *STONNE*	15	39	46
05.04	Valorar y analizar el resultado de las pruebas	10	26	96
Total		37	86	200

Tabla 16: Duración final Sprint 4.

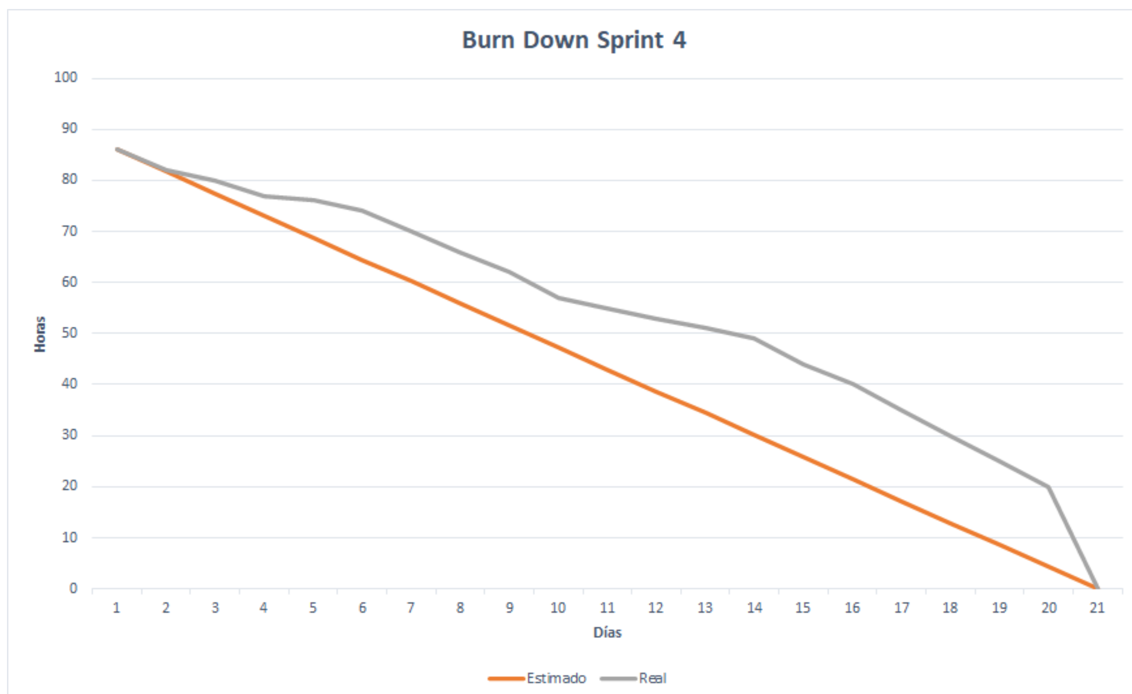

Figura 52: Gráfico de *burn down* para el Sprint 4.

7 Despliegue y prueba de la implementación

7.1 Plan de pruebas

Al finalizar cada sprint se ha de desarrollar un plan de casos de prueba para comprobar que las tareas propuestas se han completado con éxito. En la Tabla 17 se ha elaborado un plan de pruebas de acuerdo a cada historia de usuario con el fin de verificar la calidad del incremento creado. Cada caso de prueba será comprobado por el desarrollador del proyecto.

ID	Descripción	Historia de Usuario	Propósito	Superada
01	Comprobar instalación local de *Python*	02	Codificar el proyecto	Sí
02	Comprobar instalación local de *Anaconda*	02	Codificar el proyecto	Sí
03	Comprobar instalación local de *PyTorch*	02	Codificar el proyecto	Sí
04	Comprobar el funcionamiento de *DLRM* nativo	02	Codificar el proyecto	Sí
05	Comprobar instalación y compilación de *STONNE*	02	Codificar el proyecto	Sí
06	Comprobar el funcionamiento de *STONNE* en el entorno local	02	Codificar el proyecto	Sí
07	Comprobar funcionamiento del algoritmo *CSR*	03	Conocer el funciona-miento de *DLRM*	Sí
08	Comprobar funcionamiento de las *EmbeddingBags*	03	Conocer el funciona-miento de *DLRM*	Sí
09	Comprobar salida del *benchmark* con *DLRM*	03	Conocer el funciona-miento de *DLRM*	Sí
10	Comprobar funcionamiento de las *EmbeddingBags* en *STONNE*	04	Integrar el modelo *DLRM* en *STONNE*	Sí
11	Comprobar datos de entrada *STONNE*	04	Integrar el modelo *DLRM* en *STONNE*	Sí
12	Comprobar salida del modelo *DLRM* en *STONNE*	04	Integrar el modelo *DLRM* en *STONNE*	Sí
13	Comprobar salida del modelo usando arquitecturas aceleradoras	04	Integrar el modelo *DLRM* en *STONNE*	Sí
14	Comprobar paso de cargas personalizadas a *DLRM*	05	Caracterizar el modelo propuesto	Sí

Tabla 17: Plan de pruebas.

A continuación se muestra la especificación realizada para cada caso de prueba, indicando el propósito, precondiciones, procedimiento utilizado, etc. Como se observa, los primeros sprints orientados a la instalación del entorno local y al desarrollo del *benchmark* tienen asignados una mayor cantidad de casos de prueba que en los últimos sprints, donde ya se ha validado su ejecución y solamente habría que comprobar que se pueden crear *datasets* personalizados.

Caso de prueba 02	Comprobar instalación local de *Anaconda*
Rol	Desarrollador
Historia de usuario	02
Propósito	Codificar el proyecto
Tipo	Validación
Precondiciones	Ninguna
Procedimiento	Mediante el comando "*conda*" comprobar la versión instalada
Resultados esperados	Se ejecute correctamente y devuelve la versión instalada
Artífice	Nicolás Meseguer Iborra
Resultado	Aprobado

Tabla 19: Caso de prueba 02.

Caso de prueba 01	Comprobar instalación local de *Python*
Rol	Desarrollador
Historia de usuario	02
Propósito	Codificar el proyecto
Tipo	Validación
Precondiciones	Ninguna
Procedimiento	Comprobar la versión de *python* con *–version*
Resultados esperados	Se ejecute correctamente y devuelve la versión instalada
Artífice	Nicolás Meseguer Iborra
Resultado	Aprobado

Tabla 18: Caso de prueba 01.

Caso de prueba 03	Comprobar instalación local de *PyTorch*
Rol	Desarrollador
Historia de usuario	02
Propósito	Codificar el proyecto
Tipo	Validación
Precondiciones	Tener instalado *Python* y *Anaconda*
Procedimiento	Usando *pip*/*conda* instalar la librería de *PyTorch*
Resultados esperados	Importar el módulo *PyTorch* correctamente
Artífice	Nicolás Meseguer Iborra
Resultado	Aprobado

Tabla 20: Caso de prueba 03.

Caso de prueba 04	Comprobar el funcionamiento de *DLRM* nativo
Rol	Desarrollador
Historia de usuario	02
Propósito	Codificar el proyecto
Tipo	Validación
Precondiciones	Tener instalado *PyTorch*
Procedimiento	Ejecutar el fichero *dlrm.py* del modelo
Resultados esperados	El modelo se ejecuta correctamente
Artífice	Nicolás Meseguer Iborra
Resultado	Aprobado

Tabla 21: Caso de prueba 04.

Caso de prueba 05	Comprobar instalación y compilación de *STONNE*
Rol	Desarrollador
Historia de usuario	02
Propósito	Codificar el proyecto
Tipo	Validación
Precondiciones	Tener instalado *Anaconda*
Procedimiento	Compilar el frontend de *STONNE* escrito en *PyTorch*
Resultados esperados	La compilación se realiza satisfactoriamente
Artífice	Nicolás Meseguer Iborra
Resultado	Aprobado

Tabla 22: Caso de prueba 05.

Caso de prueba 06	Comprobar funcionamiento de *STONNE* en el entorno local
Rol	Desarrollador
Historia de usuario	02
Propósito	Codificar el proyecto
Tipo	Validación
Precondiciones	Haber compilado las librerías de *STONNE*
Procedimiento	Ejecutar simulación test en *STONNE*
Resultados esperados	La ejecución se realiza satisfactoriamente
Artífice	Nicolás Meseguer Iborra
Resultado	Aprobado

Tabla 23: Caso de prueba 06.

Caso de prueba 07	Comprobar funcionamiento del algoritmo *CSR*
Rol	Desarrollador
Historia de usuario	03
Propósito	Codificar el proyecto
Tipo	Validación
Precondiciones	Entender como se conforma un *multi-hot encoding*
Procedimiento	Crear un algoritmo *CSR* que genere un *multi-hot* vector a partir de 2 vectores (*indices* y *offsets*)
Resultados esperados	El algoritmo funciona correctamente
Artífice	Nicolás Meseguer Iborra
Resultado	Aprobado

Tabla 24: Caso de prueba 07.

Caso de prueba 08	Comprobar funcionamiento de las *EmbeddingBags*
Rol	Desarrollador
Historia de usuario	03
Propósito	Codificar el proyecto
Tipo	Validación
Precondiciones	Algoritmo *CSR* funciona correctamente
Procedimiento	Simular el funcionamiento de las *EmbeddingBags*
Resultados esperados	La salida se corresponde con la del modelo original
Artífice	Nicolás Meseguer Iborra
Resultado	Aprobado

Tabla 25: Caso de prueba 08.

Caso de prueba 09	Comprobar salida del *benchmark* con *DLRM*
Rol	Desarrollador
Historia de usuario	03
Propósito	Codificar el proyecto
Tipo	Validación
Precondiciones	Algoritmo *Embedding* funcione correctamente
Procedimiento	Realizar una ejecución con los mismos datos de entrada sobre el benchmark
Resultados esperados	La salida se corresponde con la del modelo original
Artífice	Nicolás Meseguer Iborra
Resultado	Aprobado

Tabla 26: Caso de prueba 09.

Caso de prueba 10	Comprobar funcionamiento de las *EmbeddingBags* en *STONNE*
Rol	Desarrollador
Historia de usuario	04
Propósito	Integrar *DLRM* en *STONNE*
Tipo	Validación
Precondiciones	Haber validado el algoritmo *Embedding*
Procedimiento	Realizar una ejecución con los mismos datos de entrada sobre el modelo en *STONNE*
Resultados esperados	La salida de las *EmbeddingBags* se corresponde con la del modelo original
Artífice	Nicolás Meseguer Iborra
Resultado	Aprobado

Tabla 27: Caso de prueba 10.

Caso de prueba 11	Comprobar datos de entrada *STONNE*
Rol	Desarrollador
Historia de usuario	04
Propósito	Integrar *DLRM* en *STONNE*
Tipo	Validación
Precondiciones	Conocer el funcionamiento del algoritmo *CSR*
Procedimiento	Comprobar que los datos *sparse* se transforman a *multi-hot vector* y los dense se envían a la *Bottom-MLP*
Resultados esperados	Los datos se tratan correctamente.
Artífice	Nicolás Meseguer Iborra
Resultado	Aprobado

Tabla 28: Caso de prueba 11.

Caso de prueba 12	Comprobar salida del modelo *DLRM* en *STONNE*
Rol	Desarrollador
Historia de usuario	04
Propósito	Integrar *DLRM* en *STONNE*
Tipo	Validación
Precondiciones	El cómputo de las *EmbeddingBags* ha de ser correcto
Procedimiento	Comparar la salida de ambos modelos usando los mismos datos de entrada
Resultados esperados	La salida de ambos modelos es la misma.
Artífice	Nicolás Meseguer Iborra
Resultado	Aprobado

Tabla 29: Caso de prueba 12.

Caso de prueba 13	Comprobar salida del modelo *DLRM* en *STONNE* usando Arq. aceleradoras
Rol	Desarrollador
Historia de usuario	04
Propósito	Integrar *DLRM* en *STONNE*
Tipo	Validación
Precondiciones	La salida del modelo *DLRM* en *STONNE* ha de ser correcta.
Procedimiento	Comparar la salida de ambos modelos usando los mismos datos de entrada, siendo esta vez ejecutados sobre las Arq. aceleradoras
Resultados esperados	La salida de ambos modelos es la misma.
Artífice	Nicolás Meseguer Iborra
Resultado	Aprobado

Tabla 30: Caso de prueba 13.

Caso de prueba 14	Comprobar paso de cargas personalizadas al modelo *DLRM*
Rol	Desarrollador
Historia de usuario	05
Propósito	Caracterizar el modelo propuesto
Tipo	Validación
Precondiciones	La salida del modelo *DLRM* acelerado en *STONNE* ha de ser correcta.
Procedimiento	Utilizar distintas configuraciones para los componentes del modelo mediante el paso de parámetro por CLI.
Resultados esperados	El modelo acepta los parámetros y se ejecuta correctamente.
Artífice	Nicolás Meseguer Iborra
Resultado	Aprobabdo

Tabla 31: Caso de prueba 14.

7.2 Escalabilidad y mantenimiento/soporte

La mayoría de los modelos actuales ejecutados sobre *deep-learning* son reconfigurables y por tanto escalables, permitiendo así aumentar el numero de componentes que se ejecutan, número de capas, añadir componentes, etc. Si nos fijamos en el presente proyecto, en el que proponemos una arquitectura aceleradora, ocurre de igual manera el modelo es completamente escalable. De hecho, para realizar el presente trabajo, el modelo DLRM nativo ha sido ampliado para poder integrarlo en STONNE.

La herramienta de simulación *STONNE* que hemos utilizado para simular los *ASICS* de *MAERI* y *SIGMA* es modular y reconfigurable, se puede cambiar el controlador de memoria, el tamaño de los *buffers*, número de *PEs* de cada arquitectura, ancho debanda, etc.

Podemos concluir en que el proyecto realizado ostenta una gran escalabilidad, permite a los desarrolladores cambiar parámetros internos (tanto del modelo *DLRM*, como de la herramienta de simulación *STONNE*), con el fin de configurar distintas topologías y probar sobre ellas cargas de trabajo reales.

Un aspecto crucial a tener en cuenta, puesto que se ha contribuido sobre el desarrollo de la herramienta *STONNE* (se le ha dotado de la capacidad de ejecutar el componente *Embedding*), y el modelo *DLRM* también se encuentra integrado dentro del proyecto *STONNE* se han seguido los documentos *PEP 8* y *PEP 257*. El código se encuentra documentado en inglés facilitando así su posterior desarrollo en caso de ser necesario. Además, las funciones implementadas y las librerías desarrolladas (*SimulatedSparse*) se han realizado conforme al resto de librerías ya implementadas en *STONNE*, facilitando la escalabilidad y mantenimiento.

Finalmente, el framework utilizado *PyTorch* es muy flexible, y recibe constantes actualizaciones, por lo que el mantenimiento del propio lenguaje es relativamente sencillo y cabe la posibilidad de añadir nuevas funcionalidades, para posteriormente ser portadas a *STONNE*.

7.3 Plan de formación de usuarios

Con el fin de atraer a nuevos desarrolladores y/o usuarios al proyecto, se han desarrollado tres diferentes manuales de usuario dependiendo de lo que se quiera conseguir:

- Manual de instalación *STONNE*. Manual de instalación para la herramienta de simulación *STONNE*. Como preconocían se ha de tener la distribución de *Anaconda* instalada (la versión de este proyecto se puede encontrar en la Sección 6.2. El manual se encuentra disponible en el Anexo 10.1.

- Manual de usuario del modelo *DLRM*. Manual que explica como realizar ejecuciones sobre el modelo *DLRM* ya integrado en *STONNE*. También se detalla como se puede cambiar el fichero de configuración, para probar distintas arquitecturas (*TPU*, etc.). Finalmente, se explica como se pueden rea-

lizar cargas de trabajo personalizadas y los distintos parámetros. Este manual está orientado a usuarios avanzados que quieran desarrollar/proponer nuevas arquitecturas para modelos de recomendación. El manual se encuentra disponible en el Anexo 10.2.

- *Scripts* pruebas *STONNE*. Este último manual está orientado a como leer la salida de los datos que *STONNE* devuelve. Incluye los *scripts* que se han utilizado y como configurar la variable de *entorno* en Linux para cambiar la carpeta de salida. El manual se encuentra disponible en el Anexo 10.3.
- Ejecuciones realizadas, con el fin de ayudar a nuevos desarrolladores a *testear* nuevas arquitecturas propuestas o modificaciones en las que presentamos en este trabajo, incluimos en el Anexo 10.4 una lista completa y detallada de todas las ejecuciones realizadas en este trabajo.

8 Conclusiones

8.1 Objetivos alcanzados

Llegados a este punto, podemos concluir con que el proyecto ha finalizado dentro de los márgenes de tiempo estipulados. Como paso final, vamos a comprobar los objetivos que definimos en la Sección 1.3 para verificar que han sido completados.

Destacar que el objetivo general era implementar y posteriormente evaluar el modelo DLRM en una arquitectura aceleradora híbrida usando la herramienta de simulación STONNE. Este es el objetivo principal que ha guiado el presente TFG, ha representado una gran carga de trabajo e investigación. Finalmente concluir con que se ha logrado implementar el modelo de recomendación en STONNE y se ha conseguido proponer, evaluar y caracterizarlo sobre una arquitectura aceleradora híbrida para el cómputo en la fase de inferencia.

A continuación vamos a estudiar los objetivos específicos propuestos al inicio del proyecto para verificar que se han completado.

- Objetivo específico 1. STONNE da soporte para ejecutar las capas del modelo de recomendación (*MLPs y EmbeddingBags*) de manera completa y nativa. Este objetivo ha supuesto una gran carga trabajo e investigación, especialmente sobre el modelo de recomendación DLRM, la herramienta STONNE y técnicas de DNNs.

- Objetivo específico 2. Se ha conseguido conectar DLRM con STONNE haciendo uso de la API escrita en *PyTorch*. Esta conexión también ha supuesto un gran trabajo debido a la necesidad que tiene el modelo de tratar con variables contínuas y categóricas, lo que ha supuesto el estudio de formatos CSR y como tratar este tipo de dato para ser enviado a la herramienta de simulación.

- Objetivo específico 3. Tras haber realizado los pasos anteriores, se ha conseguido realizar una ejecución completa del modelo DLRM sobre STONNE obteniendo un resultado idéntico a la ejecución en nativo del modelo (ver

Figura 36). Este paso es un poco la continuación del objetivo específico anterior.

- Objetivo específico 4. Tras haber comprobado que el modelo DLRM ha sido implementado en STONNE, se han configurado los parámetros de las arquitecturas aceleradoras, gracias al estudio que se hizo sobre estas en el sprint 1 (Sección 6.1). Para ver más información a cerca de la configuración realizada, ver la Sección 6.3.2. Una vez los aceleradores han sido configurados (ver Tabla 11) se han explorado diferentes cargas de trabajo sobre el modelo con el fin de comprobar su eficiencia energética y computacional (ver Sección 6.4.2). Podemos concluir con que gracias a este análisis se han identificado cuellos de botella (número y tamaño de las *EmbeddingBags* y las *lookups* por *EmbeddingBag*) en la ejecución del modelo DLRM.

8.2 Conclusiones del trabajo y personales

Este proyecto tiene un carácter investigador, poniendo de manifiesto la necesidad de desarrollar arquitecturas de uso específico para las partes intensivas en cómputo para modelos de *deep-learning*. Comprender y aplicar las diferentes técnicas que se han usado a lo largo de este trabajo (ya no solo sobre DLRM, sino sobre STONNE, modelos de deep-learning, arquitectura aceleradoras, etc) ha supuesto un reto fuera de cualquier expectativa.

Gracias al trabajo desarrollado y la planificación estimada, se podrá acelerar la velocidad de trabajo y eficiencia de los futuros trabajos que se realicen ya que este proyecto, al ser el primero de gran envergadura, ha servido para poner en práctica todos los conocimientos aprendidos. Además, se han obtenido competencias transversales de lectura y comprensión acerca del campo del DL, debido a la necesidad que ha habido de leer y entender muchos de los artículos de los que mencionamos.

Han habido algunas dificultades durante el desarrollo del proyecto, la complejidad de instalar y compilar algunas librerías en el servidor de desarrollo, las dependencias que necesitaban y el bajo nivel de permisos que teníamos han supuesto un reto. También la dificultad de usar una herramienta que no ha sido

desarrollada por mí y que se encuentra actualmente en fase de desarrollo también ha supuesto un desafío. Además de la complejidad que ya supone de por sí el presente TFG.

El desarrollo de este proyecto ha dado lugar a la escritura de un artículo científico con el nombre de: "Evaluación de un sistema de recomendación en un Acelerador Híbrido con STONNE" para el congreso nacional: XXI Jornadas de Paralelismo organizadas por "La Sociedad de Arquitectura y Tecnología de Computadores" (SARTECO) (SARTECO, 2021). El artículo se encuentra en el Anexo 10.5.

Durante el desarrollo del presente trabajo se ha tenido muy en cuenta las competencias asociadas a las Tecnologías de la Información (TI) y a la Ingeniería del Software (IS). Es importante realizar una planificación previa a la hora de comenzar un proyecto para poder efectuar un ciclo de vida del desarrollo del software y su gestión.

Cuando comencé este proyecto junto a D. José Luis Abellán sabía que iba a ser duro. El trabajo invertido en este proyecto ha sido desmesurado con el que se había estipulado, pero cada hora ha merecido completamente la pena.

8.3 Vías futuras

Como ya hemos mencionado desde el principio de este proyecto, hacemos uso de arquitecturas aceleradoras reconfigurables. Especialmente en el campo del *deep-learning*, donde los *rigid accelerator*, aceleradores rígidos, están siendo reemplazados por nuevas arquitecturas flexibles, que son capaces de adaptarse a distintos flujos de datos para maximizar el rendimiento-por-vatio. El trabajo presentado es altamente escalable y modular. Cualquier desarrollador que quiera retomar el proyecto, en la Sección de Anexos tiene diferentes manuales para facilitar su desarrollo.

Dentro de los sistemas de recomendación, podríamos considerar como trabajo futuro la investigación de aceleradores aún más concretos con el fin de desarrollar modelos de recomendación específicos, intentando lograr una mayor eficiencia computacional. También se podría considerar acelerar la fase de *training*, realizar una caracterización usando un sistema o diseño específico.

También, como propuesta a largo plazo sería ver el impacto de los diferentes diseños seleccionados, *MAERI* y *SIGMA*, variando su configuración, es decir, ajustando parámetros a más bajo nivel, entre los que encontraríamos ajustes como el *tiling*, *sparsity*, etc. también se podría realizar un estudio sobre la configuración a más bajo nivel en *STONNE*.

Gracias a los resultados de la evaluación realizada, como trabajo futuro se va a mejorar la arquitectura del acelerador híbrido propuesto implementando una jerarquía de memoria que eficientemente soporte la inmensa cantidad de lecturas y escrituras de memoria que el modelo DLRM necesita durante su procesamiento *sparse*.

9 Referencias

Agiles, P. (2020). *Qué es scrum.* Descargado de https://proyectosagiles.org/que-es-scrum/

Ajitsaria, A. (2018). *Recommendation model with collaborative filtering.* Descargado de https://realpython.com/build-recommendation-engine-collaborative-filtering/

APD, R. (2019). *En qué consiste la metodología kanban y cómo utilizarla.* Descargado de https://www.apd.es/metodologia-kanban/

Arrabales, R. (2016). *Deep learning: qué es.* Descargado de https://www.xataka.com/robotica-e-ia/deep-learning-que-es-y-por-que-va-a-ser-una-tecnologia-clave-en-el-futuro-de-la-inteligencia-artificial

Carrasco, L. (2020). *Salario de los ingenieros informáticos.* Descargado de https://blog.infoempleo.com/a/sueldos-de-los-informaticos-en-espana/

Ceja, A. (2020). *Deep learning vs machine learning.* Descargado de https://www.neoland.es/blog/machine-learning-vs-deep-learning

Costa, C. D. (2020). *Python libraries for ml.* Descargado de https://towardsdatascience.com/best-python-libraries-for-machine-learning-and-deep-learning-b0bd40c7e8c

Custódio, M. (2018). *Roi: ¿qué es?* Descargado de https://www.rdstation.com/es/blog/roi/

Damorelos. (2019). *Metodología tradicional o ágil.* Descargado de https://www.scio.com.mx/blog/metodologia-tradicional-o-agil-software/

Facebook. (2021). *Github: Dlrm for personalization & recommendation systems.* Descargado de https://github.com/facebookresearch/dlrm

Garzás, J. (2001). *Agile methods y extreme programming.* Descargado de https://www.slideserve.com/jgarzas/extreme-programming-y-m-todos-giles-powerpoint-ppt-presentation

Garzás, J. (2011). *El manifiesto ágil cumple 10 años.* Descargado de https://www.javiergarzas.com/2011/02/aniversario-manifiesto-agil.html

Garzás, J. (2014). *Las metodologías ágiles no existen.* Descargado de https://www.javiergarzas.com/2014/10/las-metodologias-agiles-existen.html

GNU. (2016). *Gnu screen.* Descargado de https://www.gnu.org/software/screen/

GoogleDevelopers. (2018). *Recommendations; what and why?* Descargado de https://developers.google.com/machine-learning/recommendation/overview

Gupta, U., Hsia, S., Saraph, V., Wang, X., Reagen, B., Wei, G., … Wu, C. (2020). Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. *CoRR*, *abs/2001.02772*. Descargado de http://arxiv.org/abs/2001.02772

Gupta, U., Wang, X., Naumov, M., Wu, C., Reagen, B., Brooks, D., … Zhang, X. (2019). The architectural implications of facebook's dnn-based personalized recommendation. *CoRR*, *abs/1906.03109*. Descargado de http://arxiv.org/abs/1906.03109

Haldar, M. y. o. (2019). Applying deep learning to airbnb search. En (p. 1927–1935). New York, NY, USA: Association for Computing Machinery. Descargado de https://doi.org/10.1145/3292500.3330658 doi: 10.1145/3292500.3330658

Hale, J. (2019). *Which deep learning framework is growing fastest?* Descargado de https://www.kdnuggets.com/2019/05/which-deep-learning-framework-growing-fastest.html

Hwang, R., Kim, T., Kwon, Y., y Rhu, M. (2020). Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations. *CoRR*, *abs/2005.05968*. Descargado de https://arxiv.org/abs/2005.05968

Indeed. (2021). *Salario de un scrum master.* Descargado de https://es.indeed.com/career/scrum-master/salaries

IONOS. (2020). *Deep learning vs machine learning.* Descargado de https://www.ionos.es/digitalguide/online-marketing/marketing-para-motores-de-busqueda/deep-learning-vs-machine-learning/

Ironhack. (2021). *What is machine learning?* Descargado de https://www.ironhack.com/en/data-analytics/what-is-machine-learning

Jouppi, N. P. y. o. (2017). In-datacenter performance analysis of a tensor processing unit. En *44th int'l symp. on computer architecture.*

Karbhari, N. y. o. (2017). Recommendation system using content filtering: A case study for college campus placement. En *2017 international conference on energy, communication, data analytics and soft computing (icecds)* (p. 963-965). doi: 10.1109/ICECDS.2017.8389579

Kwon, H., Samajdar, A., y Krishna, T. (2018, marzo). MAERI: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*.

Manager, S. (2021). *Estimación de póquer.* Descargado de https://www.scrummanager.net/bok/index.php?title=Estimaci%C3%B3n_de_p%C3%B3quer

Martin, A. (2018). *Historias de usuario scrum.* Descargado de https://urtanta.com/historias-de-usuario/

Martínez, F. M., Abellán, J. L., Acacio, M. E., y Krishna, T. (2020, junio). Stonne: A detailed architectural simulator for flexible neural network accelerators. *arXiv preprint arXiv:2006.07137v1*.

Martínez, F. M., Abellán, J. L., Acacio, M. E., y Krishna, T. (2021). *Stonne: A simulation tool for neural networks engines.* Descargado de https://github.com/stonne-simulator/stonne

Meng, Z. (2020). *Biologicla vs artificial neuron.* Descargado de https://www.researchgate.net/figure/A-biological-neuron-in-comparison-to-an-artificial-neural-network-a-human-neuron-b_fig2_339446790

Menzinsky, A. (2015). *¿qué es el sprint 0?* Descargado de https://scrum.menzinsky.com/2015/03/que-es-el-sprint-0-actualmente-hay.html

Naumov, M., Mudigere, D., Shi, H. M., Huang, J., Sundaraman, N., Park, J., … Smelyanskiy, M. (2019). Deep learning recommendation model for personalization and recommendation systems. *CoRR*, *abs/1906.00091*. Descargado de http://arxiv.org/abs/1906.00091

NextU. (2020). *¿qué es github?* Descargado de https://www.nextu.com/blog/que-es-github/

Petrova, S. (2019). *The latest reports about the popular mindset.* Descargado de https://adevait.com/blog/remote-work/adopting-agile-the-latest-reports-about-the-popular-mindset

Peñalvo, F. J. G., Holgado, A. G., y Ingelmo, A. V. (2019). Metodologías de ingeniería de software. *Ingeniería de Software*, *I*, 1-3.

Python. (2021). *Acerca de python.* Descargado de https://www.python.org/

PyTorch. (2021a). *Características de pytorch.* Descargado de https://pytorch.org/features/

PyTorch. (2021b). *Módulo sparse pytorch.* Descargado de https://pytorch.org/docs/stable/_modules/torch/nn/modules/sparse.html

Rehkopf, M. (2021). *Historias de usuario.* Descargado de https://www.atlassian.com/es/agile/project-management/user-stories

Requena, A. (2021). *Redes artificiales.* Descargado de https://www.um.es/LEQ/Atmosferas/Ch-VI-3/F63s4p3.htm

Rodriguez, A. (2018). *Youtube's recommendations drive around 70 %.* Descargado de https://qz.com/1178125/youtubes-recommendations-drive-70-of-what-we-watch/

RZone. (2020). *El mejor terminal para windows con cliente ssh.* Descargado de https://www.redeszone.net/analisis/software/mobaxterm-terminal-windows/

SARTECO. (2021). *Sociedad de arquitectura y tecnología de computadores.* Descargado de https://sarteco.org/

Schafer, B. y. o. (2007, 01). Collaborative filtering recommender systems..

Schwaber, K., y Sutherland, J. (2020). The scrum guide. *The Definitive Guide to Scrum*, *I*, 4.

Slack. (2021). *¿qué es slack?* Descargado de https://slack.com/intl/es-es/help/articles/115004071768-%C2%BFQu%C3%A9-es-Slack-

Toro, L. (2020). *Anaconda distribution.* Descargado de https://blog.desdelinux.net/ciencia-de-datos-con-python/

Wrike. (2021). *¿qué es wrike?* Descargado de https://www.wrike.com/es/

Yan, M., Deng, L., Hu, X., Liang, L., Feng, Y., Ye, X., … Xie, Y. (2020). Hygcn: A GCN accelerator with hybrid architecture. *CoRR*, *abs/2001.02514*. Descargado de http://arxiv.org/abs/2001.02514

y otros, E. Q. (2020, marzo). SIGMA: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. *Int'l Symp. on High-Performance Computer Architecture*.

10 Anexos

10.1 Manual de instalación *STONNE*

Como ya hemos mencionado a lo largo de este proyecto, *STONNE* es un simulador con precisión a nivel de ciclo que puede conectarse a cualquier modelo de *deep-learning* y actuar como un acelerador.

Figura 53: Logotipo de *STONNE* (Martínez y cols., 2020).

Para realizar la instalación es necesario tener el entorno anaconda instalado en la máquina, se puede comprobar fácilmente escribiendo *conda –version*.

El siguiente paso será realizar un *git clone* del repositorio del simulador *STONNE*. Dependiendo de la fecha en la que se realice este paso, el repositorio se encontrará en una versión u otra (Martínez y cols., 2021). Este repositorio se descargará en la carpeta *STONNE-SIM*, nos referiremos a esta carpeta durante la instalación.

Una vez tenemos el simulador y la herramienta de gestor de paquetes *Anaconda*, tenemos que compilar 2 librerías diferentes. El *frontend* de *STONNE* escrito en *PyTorch* y posteriormente las librerías escritas en *PyTorch* que conectan con la *API* de *STONNE*.

10.1.1 Compilar el *frontend* de *STONNE*

Para compilar esta primera librería tenemos que usar el entorno de anaconda, este se puede activar escribiendo:

```
conda activate
```

Posteriormente, nos dirigimos a:

```
~/STONNE-SIM/pytorch-frontend
```

En este directorio escribiremos, dentro del entorno *conda*, el siguiente comando:

```
python setup.py install
```

En este punto se comenzará a instalar/compilar el frontend de *STONNE* escrito en *PyTorch*. El primer paso de la instalación se ha realizado con éxito.

10.1.2 Compilar las librerías de conexión a *STONNE*

Una vez el *frontend* se ha compilado correctamente, vamos a compilar las librerías que sirven de conexión entre *PyTorch* y *STONNE*. Para este segundo paso, es muy importante que nos salgamos del entorno de *Anaconda*, para ello escribiremos:

```
conda deactivate
```

A continuación, nos dirigimos al directorio:

```
~/STONNE-SIM/pytorch-frontend/stonne-connection
```

Y ejecutamos el siguiente comando:

```
python setup.py install
```

Tras compilarse y mostrarnos algunos *warnings*, las librerías que usa *STONNE* estarán listas para usarse y el simulador habrá quedado instalado correctamente para su uso.

Atención, en el caso de realizar algún cambio en las librerías de *STONNE*, como *SimulatedSparse* con el fín de añadir nueva funcionalidad al simulador, se tendrá que compilar nuevamente ambas librerías, es decir, se tendrán que repetir ambos pasos.

Finalmente, para demostrar que *STONNE* se ha instalado correctamente, ejecutaremos una operación *GEMM Sparse* en el simulador. Para ello nos dirigiremos a:

```
~/STONNE-SIM/stonne
```

Y ejecutaremos el siguiente comando:

```
./stonne -DenseGEMM -M=20 -N=20 -K=256 -num_ms=256 -dn_bw=64 -rn_bw=64
-T_K=64 -T_M=2 -T_N=1
```

En la Figura 54 se muestra el resultado esperado de *STONNE*.

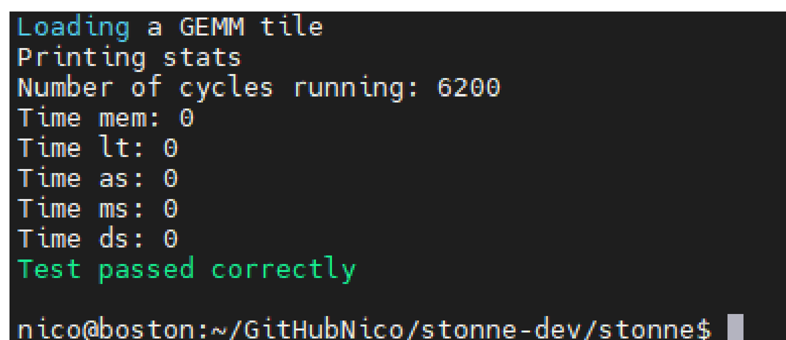

Figura 54: Salida test de *STONNE*.

10.2 Manual de usuario del modelo *DLRM*

En este momento, el simulador *STONNE* ya contará con *DLRM* integrado, por lo que al realizar la instalación anterior, ya habremos instalado el modelo *DLRM* listo para ejecutarse sobre *STONNE*.

En este manual explicaremos como se pueden realizar diferentes casos de prueba y que parámetros se pueden modificar, acompañados de una breve explicación.

El modelo *DLRM* se encuentra ya integrado dentro de *STONNE*, para comprobar la carpeta donde se encuentra, nos dirigimos a:

```
~/STONNE-SIM/benchmarks/dlrm
```

En esta carpeta se encuentra el modelo *DLRM* escrito en *PyTorch* junto con el resto de ficheros necesarios para su ejecución. De este modelo se pueden modificar los siguientes parámetros:

- *embedding-size* Determina el número de *EmbeddingBags* que existirán y por cada *EmbeddingBag*, cuantos índices(filas) tendrá. Por ejemplo, *100-100-100* nos indica que existirán 3 *EmbeddingBags* y cada una con 100 índices (filas). Este parámetro no tiene un valor predefinido.

- *sparse-feature-size* Determina el tamaño de columnas que tendrán las *EmbeddingBags*. Ya que posteriormente se concatenan con la salida de la *Bottom-MLP*, este número ha de ser el mismo que el de la última capa de la *Bottom-MLP*. Valor por defecto, 32, extraído de Centaur (ver Sección 6.4.2).

- *mlp-bot* Indica cuantas capas y las neuronas/capa que existirán. Por ejemplo, *128-64-32*, tres capas, y cada una con 128, 64 y 32 neuronas. La última capa se corresponde con el tamaño de *sparse-feature-size*. Valor por defecto, 128-64-64-32.

- *mlp-top*, de igual manera para la *Top-MLP*. Como la última capa es una sigmoidea, solo puede existir 1 neurona. Recordar que el modelo DLRM añade una capa extra en base al número de *EmbeddingBags* y del número de neuronas de la última capa de la *Bottom-MLP*. Valor por defecto, 128-64-1.

- *batch-size*, indica el número de índices que se cogerán por cada pasada. Este parámetro por defecto es *128* de acuerdo a la propuesta en DLRM.

- *num-indices-lookup*, en cada búsqueda, cuantos índices se podrán coger como máximo de la *EmbeddingBag*. Por ejemplo, 15, indica que como máximo se cogerán 15 índices de la *EmbeddingBag* por búsqueda, sin importar el tamaño de la matriz de pesos. Valor por defecto, 100.

- *data-size* Determina el número de *lookups* ó búsquedas que se realizarán por *EmbeddingBag*. Por ejemplo, 10 *lookups* por *EmbeddingBag* significará una intensidad de cómputo mayor, que 1. Este parámetro no tiene un valor por defecto.

Para realizar una ejecución test, podemos ejecutar el siguiente comando, en el directorio del modelo *DLRM*:

```
python dlrm_s_pytorch.py --arch-embedding-size=1000-1000-1000-1000
--arch-sparse-feature-size=32 --arch-mlp-bot=64-32
--arch-mlp-top=64-1 --mini-batch-size=128 --num-indices-per-lookup=50
--data-size=5
```

En la Figura 55 se muestra la salida del modelo *DLRM* con la ejecución mencionada:

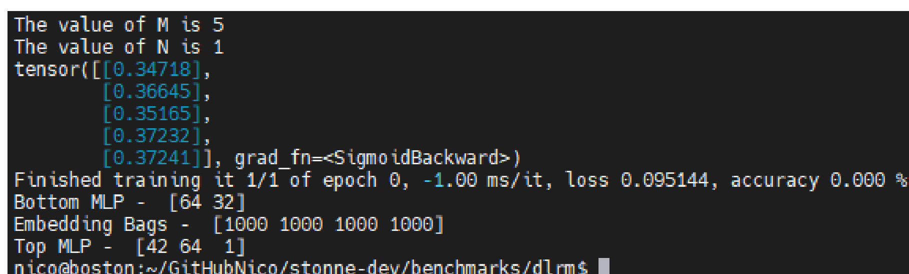

Figura 55: Salida test de *DLRM*.

En caso de querer realizar pruebas de mayor magnitud, se recomienda usar el software *GNU Screen*, de tal manera que se pueda ejecutar en segundo plano sin necesidad de mantenerse conectado mediante *SSH*.

10.2.1 Cambiar la arquitectura aceleradora

También cabe la posibilidad de utilizar diferentes arquitecturas aceleradoras, a parte de *MAERI* y *SIGMA*, por ejemplo la *TPU*. Pero esta parte está destinada a desarrolladores de arquitecturas/compiladores, etc. No se recomienda para el usuario medio.

Para poder cambiar la arquitectura aceleradora a usar, basta con cambiar el archivo de configuración *stonne config* que se encuentra en el constructor de las *Emebddings* o *MLPs*.

En la Figura 56 se muestra como las *EmbeddingBags* usan la arquitectura de *SIGMA* para el cómputo en el simulador *STONNE*.

```
else:
    ''' EmbeddingBag @ STONNE - SimulatedSparse'''
    EE = nn.SimulatedEmbeddingBag(n, m, '../../simulation_files/magma_128mses_128_bw.cfg', m
```

Figura 56: Uso de *SIGMA* como fichero de configuración.

Una lista completa de los archivos de configuración que *STONNE* soporta se encuentra en:

```
~/STONNE-SIM/simulation_files
```

En la Figura 57 se muestra el contenido del directorio, mostrando algunas de las configuraciones que *STONNE* permite simular.

```
total 48
-rw-rw-r-- 1 nico nico 157 dic 13 12:04 maeri_128mses_128_bw.cfg
-rw-rw-r-- 1 nico nico 155 dic 13 12:04 maeri_128mses_64_bw.cfg
-rw-rw-r-- 1 nico nico 155 dic 13 12:04 maeri_256mses_128_bw.cfg
-rw-rw-r-- 1 nico nico 157 dic 13 12:04 maeri_256mses_256_bw.cfg
-rw-rw-r-- 1 nico nico 157 dic 13 12:04 maeri_256mses_64_bw.cfg
-rw-r--r-- 1 nico nico 155 mar  1 11:23 magma_128mses_128_bw.cfg
-rw-rw-r-- 1 nico nico 124 dic 13 12:04 sigma_128mses_128_bw.cfg
-rw-rw-r-- 1 nico nico 122 dic 13 12:04 sigma_128mses_64_bw.cfg
-rw-rw-r-- 1 nico nico 124 dic 13 12:04 sigma_256mses_128_bw.cfg
-rw-rw-r-- 1 nico nico 124 dic 13 12:04 sigma_256mses_256_bw.cfg
-rw-rw-r-- 1 nico nico 122 dic 13 12:04 sigma_256mses_64_bw.cfg
-rw-rw-r-- 1 nico nico 121 dic 13 12:04 sigma_64mses_64_bw.cfg
```

Figura 57: Archivos de configuración en *STONNE*.

10.3 *Scripts* pruebas *STONNE*

Para poder probar la eficiencia del modelo con la arquitectura simulada es necesario saber como leer la salida de *STONNE* y donde esta se genera.

El simulador cuenta con una variable de entorno que te permite especificar donde queremos que la salida de *STONNE* se genere, para ello, dentro del fichero *.bashrc* escribimos lo siguiente:

```
export OUTPUT_DIR="Directorio"
```

En la Figura 58 se puede ver el directorio donde se ha guardado la salida de *STONNE* durante el presente proyecto. Se recomienda aislarlo a un directorio separado, puesto que por cada ejecución distinta sobre el modelo *DLRM*, el simulador *STONNE* genera diferentes archivos de salida. Como al final vamos a leerlos todos, entre simulación y simulación vamos a borrar la salida.

```
export OUTPUT_DIR="/home/nico/STONNE_OUTPUT_DIR"
```

Figura 58: Configuración variable de entorno STONNE.

Si tras una ejecución, como la que se puede observar en el Anexo 10.2, nos dirigimos al directorio que hemos especificado, veremos que tenemos diferentes archivos, unos de tipo *.counters* y otros de tipo *.txt*, esto se debe al propio simulador y a como está modelado.

Para poder realizar el cómputo de ciclos o lecturas/escrituras al Global Buffer, utilizaremos los siguientes comandos *bash*:

Ciclos consumidos por las *EmbeddingBags*:

```
for i in $(ls *SimulatedEmbeddingBag*.counters) ; do   cat $i
| head -1 | cut -f 2 -d '=' ; done | paste -s -d "+" | bc
```

Ciclos consumidos por las *MLPs*:

```
for i in $(ls *SimulatedLinear*.counters) ; do   cat $i
| head -1 | cut -f 2 -d '=' ; done | paste -s -d "+" | bc
```

Como se observa, ambos *scripts* son iguales, únicamente cambia el nombre del fichero a leer, puesto que las *EmbeddingBags* se ejecutan en la librería *SimulatedEmbeddingBag* y las *MLPs* en *SimulatedLinear*.

Para leer las escrituras y lecturas al *GlobalBuffer* únicamente hay que especificar el tipo de archivo a leer, *SimulatedLinear* ó *SimulatedEmbeddingBag*. Para mantener el manual de usuario limpio, se ha especificado el script para un caso general, cambiar el *NombreArchivo* por el nombre de la librería que se quiera leer.

Lecturas al *GlobalBuffer*:

```
for i in $(ls *NombreArchivo*.counters) ; do   cat $i
| grep "GLOBALBUFFER" | grep -o "READ=[[:digit:]]*" |
cut -d '=' -f 2 ; done | paste -s -d "+" | bc
```

Escrituras al *GlobalBuffer*:

```
for i in $(ls *NombreArchivo*.counters) ; do   cat $i
| grep "GLOBALBUFFER" | grep -o "WRITE=[[:digit:]]*" |
cut -d '=' -f 2 ; done | paste -s -d "+" | bc
```

10.4 Ejecuciones realizadas sobre el modelo *DLRM*

A continuación se presentan en 4 diferentes grupos las cargas de trabajo que se han utilizado en este proyecto. La explicación al significado de los nombres se encuentra en el sprint review 4 (Sección 6.4.2).

10.4.1 Grupo de evaluación

Dentro de este grupo se encuentran tres diferentes casos de estudio. Casos de estudio relacionados con el artículo *Centaur* (Hwang y cols., 2020).

- DLRM(1) Centaur

```
python dlrm_s_pytorch.py --arch-embedding-size=1000000-1000000-
1000000-1000000-1000000 --arch-sparse-feature-size=32
--arch-mlp-bot=128-64-64-32 --arch-mlp-top=128-64-1
--mini-batch-size=128 --num-indices-per-lookup=100
--data-size=20
```

- DLRM(3) Centaur

```
python dlrm_s_pytorch.py --arch-embedding-size=1000000-1000000-
1000000-1000000-1000000 --arch-sparse-feature-size=32
--arch-mlp-bot=128-64-64-32 --arch-mlp-top=128-64-1
--mini-batch-size=128 --num-indices-per-lookup=100
--data-size=80
```

- DLRM(6) Centaur

```
python dlrm_s_pytorch.py --arch-embedding-size=1000000-1000000-
1000000-1000000-1000000 --arch-sparse-feature-size=32
--arch-mlp-bot=512-256-32 --arch-mlp-top=256-256-128-128-64-64
--mini-batch-size=128 --num-indices-per-lookup=100 --data-size=2
```

10.4.2 Grupo 1

Este primer caso de estudio se centra en aumentar el número de *Embedding-Bags* progresivamente y mantener los *lookups/embeddingBag* en 20. Grupo de peso ligero o *lightweight*.

- 10-20-100

```
python dlrm_s_pytorch.py --arch-embedding-size=1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000
--arch-sparse-feature-size=32 --arch-mlp-bot=128-64-64-32
--arch-mlp-top=128-64-1 --mini-batch-size=128 --num-indices-
per-lookup=100 --data-size=20
```

- 25-20-100

```
python dlrm_s_pytorch.py --arch-embedding-size=1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000
--arch-sparse-feature-size=32 --arch-mlp-bot=128-64-64-32
--arch-mlp-top=128-64-1 --mini-batch-size=128 --num-indices-per-
lookup=100 --data-size=20
```

- 50-20-100

```
python dlrm_s_pytorch.py --arch-embedding-size=1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000
--arch-sparse-feature-size=32 --arch-mlp-bot=128-64-64-32
--arch-mlp-top=128-64-1 --mini-batch-size=128 --num-indices-per-
lookup=100 --data-size=20
```

10.4.3 Grupo 2

Este tercer caso de estudio, mantiene el orden de crecida de las *Embedding-Bags*, sin embargo aumenta los *lookups/embeddingBag* a 80. Grupo de peso medio.

- 10-80-100

```
python dlrm_s_pytorch.py --arch-embedding-size=1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000
--arch-sparse-feature-size=32 --arch-mlp-bot=128-64-64-32
--arch-mlp-top=128-64-1 --mini-batch-size=128 --num-indices-
per-lookup=100 --data-size=80
```

- 25-80-100

```
python dlrm_s_pytorch.py --arch-embedding-size=1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000
--arch-sparse-feature-size=32 --arch-mlp-bot=128-64-64-32
--arch-mlp-top=128-64-1 --mini-batch-size=128 --num-indices-per-
lookup=100 --data-size=80
```

- 50-80-100

```
python dlrm_s_pytorch.py --arch-embedding-size=1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000-
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000-
```

```
1000000-1000000-1000000-1000000-1000000-1000000-1000000-1000000
--arch-sparse-feature-size=32 --arch-mlp-bot=128-64-64-32
--arch-mlp-top=128-64-1 --mini-batch-size=128 --num-indices-per-
lookup=100 --data-size=80
```

10.4.4 Grupo 3

Este último caso de estudio lleva al límite los experimentos acercándonos a entornos de cargas de trabajo reales. Los índices/tabla de las *EmbeddingBags* han aumentado en 9 millares (10 millones índices/tabla). Grupo de peso alto.

- 50-20-100

```
python dlrm_s_pytorch.py --arch-embedding-size=10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000 --arch-
sparse-feature-size=32 --arch-mlp-bot=128-64-64-32 --arch-mlp-
top=128-64-1 --mini-batch-size=128 --num-indices-per-lookup=100
--data-size=20
```

- 50-80-100

```
python dlrm_s_pytorch.py --arch-embedding-size=10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
```

```
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000 --arch-
sparse-feature-size=32 --arch-mlp-bot=128-64-64-32 --arch-mlp-
top=128-64-1 --mini-batch-size=128 --num-indices-per-lookup=100
--data-size=80
```

- 50-20-1000

```
python dlrm_s_pytorch.py --arch-embedding-size=10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000 --arch-
sparse-feature-size=32 --arch-mlp-bot=128-64-64-32 --arch-mlp-
top=128-64-1 --mini-batch-size=128 --num-indices-per-lookup=1000
--data-size=20
```

- 50-80-1000

```
python dlrm_s_pytorch.py --arch-embedding-size=10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000-10000000-
10000000-10000000-10000000-10000000-10000000-10000000 --arch-
```

```
sparse-feature-size=32 --arch-mlp-bot=128-64-64-32 --arch-mlp-
top=128-64-1 --mini-batch-size=128 --num-indices-per-lookup=1000
--data-size=80
```

10.5 Artículo redactado para *SARTECO*

Por cuestiones de tamaño y estilos de página (estilo del TFG es distinto al del artículo) y con la finalidad de no deformar el artículo (ajustarlo a esta página), el artículo aparece en la siguiente página.

Evaluación de un Sistema de Recomendación en un Acelerador Híbrido

Nicolás Meseguer-Iborra,[1] Francisco Muñoz-Martínez[2], Manuel E. Acacio[2], José L. Abellán[1]

Resumen— **Los sistemas de recomendación basados en técnicas de *Deep Learning* (DL) es un área en constante evolución. Los sistemas actuales han de ser capaces de generar una salida (por ejemplo, la probabilidad de hacer clic sobre un determinado anuncio) de la manera más precisa (volver a mostrarlo o no) y dentro de un estricto margen de tiempo para que las empresas digitales maximicen sus beneficios. Sin embargo, cuando estos modelos ya entrenados se despliegan sobre Datacenters con un gran volumen de datos a procesar (cientos de miles de usuarios haciendo clic sobre anuncios), el tiempo de cómputo y la demanda de recursos computacionales (unidades de cómputo, consumo de memoria, etc.) aumenta considerablemente. Como resultado de esto, surge la necesidad de crear nuevas arquitecturas aceleradoras especializadas en procesar sistemas de recomendación. En este trabajo se va a realizar una caracterización de la ejecución del sistema de recomendación DLRM (Facebook) sobre una arquitectura aceleradora unificada híbrida (arquitectura para cómputo *sparse* y otra para cómputo *dense*; etapas en la ejecución de DLRM) que hemos propuesto para acelerar esta carga de trabajo. Para ello se usará STONNE, un simulador arquitectural para aceleradores de DL. La evaluación de DLRM se realizará mediante cargas de trabajo similares a las esperadas en un entorno real. Este análisis permitirá detectar cuellos de botella clave en la ejecución de DLRM, de modo que nos permita, como trabajo futuro, el diseño de un acelerador más optimizado para DLRM.**

Palabras clave— **Sistema de recomendación, DLRM, Caracterización de rendimiento, Acelerador para Deep Learning, Herramienta de simulación.**

I. Introducción

Los sistemas de recomendación han emergido como herramientas clave para abordar tareas de personalización y recomendación [1]; como por ejemplo, vídeos que podrían interesarte, personas que quizá conozcas o anuncios basados en tus intereses. De esta forma, a día de hoy, existe la necesidad de recomendar una gran cantidad de datos (cientos de *gigabytes*) a un público determinado de usuarios dentro de un estricto margen de tiempo. Es por ello que se emplean grandes y costosos Datacenters para su ejecución con multitud de nodos heterogéneos (compuestos por CPU+FPGA/GPU) [2].

Para alcanzar la mayor eficiencia computacional posible en la ejecución de estos sistemas de recomendación, tanto desde el punto de vista del rendimiento (tiempo en realizar una recomendación) así como en cuanto a consumo energético, es necesario realizar un co-diseño SW/HW. De este modo, una vez conocidas las características de ejecución (e.g., flujo de datos, tipos de operaciones, cuellos de botella existentes, etc) del sistema de recomendación (SW), se está en disposición de comprender cómo mejorar el diseño/arquitectura de una plataforma de cómputo especializada (HW) que acelere su procesamiento. Es decir, desarrollar un acelerador de dominio específico (del inglés *Domain-Specific Accelerator* o DSA) para sistemas de recomendación.

Desde una perspectiva algorítmica (SW), hasta hace poco los sistemas de recomendación hacían uso de técnicas como *content filtering* [3] y *collaborative filtering* [4] (ver Sección II), las cuales se basan en métodos como *k-vecinos* o *asociación por grupos*, logrando obtener resultados aceptables [5], [6]. Sin embargo, recientemente los sistemas de recomendación han empezado a hacer uso de técnicas de *aprendizaje profundo* (del inglés *deep-learning* o DL), integrando en sus modelos redes neuronales profundas (del inglés, *Deep Neural Networks* o DNNs), que superan ampliamente la precisión obtenida en los modelos de recomendación tradicionales mencionados anteriormente [1].

Las DNNs tienen dos fases de ejecución: una primera fase de entrenamiento o *training*, donde se entrena la red para que aprenda a realizar una determinada tarea (e.g., detección de señales viales en una carretera), ajustando los pesos de la red neuronal; y una segunda fase de predicción o inferencia, donde el modelo DNN se despliega para ser utilizado (por ejemplo, que la DNN entrenada se instale en una cámara en un coche y ayude en la conducción automática del vehículo). En este trabajo nos vamos a centrar en estudiar la fase de inferencia de los sistemas de recomendación.

Con el objetivo de maximizar el rendimiento-por-vatio en la ejecución de la fase de inferencia, actualmente están desarrollándose multitud de co-diseños SW/HW, propiciando el desarrollo de una gran cantidad de DSAs para DL [7], [8], [9], [10], [11], [12]. Un ejemplo claro de estas propuestas es la *Tensor Processing Unit* o TPU [13] de Google, que maximiza el rendimiento-por-vatio en el procesamiento de las operaciones de multiplicación de matrices (clave en procesamiento de DNNs) mediante una arquitectura sistólica.

Los modelos de recomendación actuales como DLRM difieren de otros modelos de DL por la necesidad de tratar con ingentes cantidades de variables categóricas (e.g., la categoría de los vídeos que el usuario ha visualizado, el género de las películas, tipo de anuncios, etc), entendidas como *sparse features* porque normalmente se representan en formato vector/matriz y constan de una gran cantidad de ceros

[1]Dpto. de Grado en Ingeniería Informática, Universidad Católica de Murcia, e-mail: {nmeseguer2,jlabellan}@ucam.edu.

[2]Dpto. de Ingeniería y Tecnología de Computadores, Universidad de Murcia, e-mail: {francisco.munoz2,meacacio}@um.es.

(es decir, una matriz dispersa o *sparse*). Por otro lado, el modelo DLRM también utiliza variables continuas (e.g., la matriz en DLRM que contiene los pesos ya entrenados, historial de estadísticas, predicciones de otros modelos [14], etc.) que constituyen *dense features* (matriz o vector con pocos ceros).

Debido a los grandes retos que se presentan a la hora de ejecutar este modelo DLRM (e.g., patrón de accesos a memoria irregular a la hora de acceder a las *sparse features*, heterogeneidad de las operaciones sparse y dense, etc) y a la tremenda popularidad y demanda que está teniendo este modelo, recientemente se han presentado algunos trabajos de caracterización que tratan de averiguar cómo se comporta este modelo DLRM en arquitecturas tradicionales como CPUs o GPUs [2], [15]. Sin embargo, todavía no se ha realizado una caracterización de ejecución detallada de su fase de inferencia sobre dispositivos DSA aceleradores empotrados. Éste constituye el objetivo principal de este trabajo.

En particular, dado el procesamiento *sparse* y *dense* que requiere DLRM, vamos a basar la caracterización en una arquitectura aceleradora híbrida especializada en procesamiento *sparse* y *dense*. En particular, el diseño utilizado está inspirado en HyGCN [16], un arquitectura acelerador para procesamiento de GCNs (*Graph Convolutional Neural Networks*) que integra dos aceleradores dedicados para distinto tipo de cómputo (*sparse* y *dense*) dentro de un mismo chip con el fin de obtener alto rendimiento y bajo consumo de energía. En nuestro caso, la arquitectura evaluada combina en un mismo acelerador la arquitectura MAERI [8], para el cómputo *dense*, y la arquitectura SIGMA [9], para el cómputo *sparse*.

Para poder realizar la caracterización de DLRM sobre este acelerador para inferencia propuesto, vamos a utilizar el simulador STONNE [17], que nos permite simular a nivel de ciclo distintos tipos de arquitecturas aceleradoras para DL (actualmente, MAERI, SIGMA y TPU), y permite realizar simulaciones del procesamiento de DNNs reales y completas ya que STONNE está conectado con el framework para DL *PyTorch*.

Para nuestro estudio, hemos realizado un análisis detallado de la ejecución de DLRM sobre el acelerador híbrido propuesto mediante cargas de trabajo sintéticas similares a las que se producen en entorno real, utilizando métricas como el tiempo de ejecución y el número de accesos a memoria. En concreto, en este trabajo hemos descubierto el fuerte peso que representa el cómputo de las variables categóricas, *sparse*, ya que el cómputo del resto del modelo varía en función del tamaño de éstas.

II. Background y Trabajo Relacionado

A. El sistema de recomendación DLRM

Los sistemas de recomendación se usan en la actualidad para una gran variedad de tareas en grandes compañías, incluyendo tasas de clic por anuncio (CTR), predicciones, *rankings*, etc. Tradicionalmente, dos son las perspectivas que se han utilizado para el desarrollo de modelos de personalización y recomendación. La primera viene de las diferentes técnicas de los sistemas de recomendación, como *content-based filtering*, en los que si un usuario A ve dos vídeos de animales, el sistema le recomienda vídeos de animales; y *collaborative filtering*, si un usuario A es similar a un usuario B (por los vídeos que ha visto) y el usuario A indica que le gusta un vídeo, entonces el vídeo se le mostrará al usuario B. Pero en los últimos años se ha comenzado a hacer uso de técnicas híbridas que combinan *content-based filtering* con *collaborative filtering* por las ventajas que ofrecen [1], [2]. La segunda perspectiva que ha contribuido al desarrollo de modelos de recomendación viene del análisis predictivo, técnicas estadísticas que analizan los datos para clasificar o predecir la probabilidad de que un evento suceda. Los modelos predictivos han evolucionado desde modelos simples (regresión lineal o logística) a modelos que incorporan redes profundas (DL). El modelo DLRM surge de la combinación de ambas perspectivas, utilizando técnicas híbridas de sistemas de recomendación y modelos predictivos basados en DL para calcular la probabilidad de que un evento ocurra.

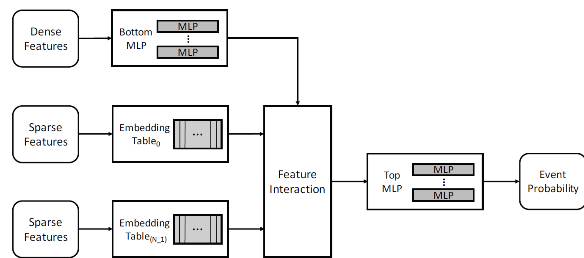

Fig. 1: Arquitectura del modelo DLRM [1].

Como se observa en la Figura 1, a alto nivel, el modelo hace uso de dos tipos diferentes de datos; *dense features* y *sparse features*; el modelo usa N *Embedding Tables* para procesar *sparse features*, que representan variables categóricas, nos referiremos a este componente como *EmbeddingBags*. Y un *multilayer perceptron*, MLP, para procesar las *dense features*, que son variables continuas. Posteriormente la salida de ambos algoritmos se combina en una fase denominada *Feature-Interactions* y se post-procesa en una *Top-MLP*, que determina la probabilidad de que un evento suceda.

En el ámbito matemático, *sparse* y *dense* son términos frecuentemente referidos al número de ceros contenidos en un vector o matriz. Por ejemplo, una matriz *sparse* estará compuesta mayoritariamente de ceros, mientras que en un vector dense, la mayoría de sus elementos serán no-nulos. Las entradas *dense* son procesadas por una *Bottom-MLP*, un *multilayer perceptron*, que se encarga de hacer una *GEneral Matrix Multiplication*, GEMM de las matrices de entrada. Como resultado se generará un vector *dense* que se enviará a la próxima fase, *Feature-interaction*. Por otra parte las entradas *sparse* serán procesadas por *EmbeddingBags*, un algoritmo que haciendo uso del formato CSR realizará operaciones GEMM sparse-

	Dense	Sparse	CSR	Reconfigurable
MAERI	✓	✗	✗	✓
TPU	✓	✗	✗	✗
SIGMA	✗	✓	✓	✓

Tabla I: Características de las arquitecturas aceleradoras.

dense, que explicaremos más adelante (Sección III), como por ejemplo, procesar productos escalares (*dot products* en inglés). Las matrices resultantes se agruparán formando una matriz dense que pasará a la siguiente fase.

En la fase denominada *Feature-Interactions*, se producen dos operaciones. Primero, la salida de la *Bottom-MLP*, un vector *dense*, se concatena con la salida del algoritmo *EmbeddingBag*, una matriz *dense*; es muy importante remarcar un aspecto, el vector dense ha de tener la misma dimensión (índices) que columnas tenga la matriz resultante con el fin de concatenarse (esto implicará un mayor o menor cómputo en la *Bottom-MLP*). A continuación, se realiza una *Matrix Factorization*. El resultado de esta fase, una matriz *dense*, se envía a una *Top-MLP* que devuelve un vector con la probabilidad de que un determinado evento suceda.

B. Aceleradores específicos para DLRM

Para acelerar el cómputo del modelo DLRM de manera eficiente, se necesitan arquitecturas aceleradoras especializadas tanto en el cómputo *dense* como en el cómputo *sparse*. Dada la reciente aparición del modelo DLRM, actualmente existen unos pocos trabajos relacionados que proponen plataformas de cómputo para aceleración de DLRM.

Por un lado, *Centaur* [2] es un acelerador híbrido compuesto por CPU y GPU integrado mediante un sistema de *multi-chiplet* (múltiples *chips* dentro de un mismo *chip*) (*CPU-Chiplet* + *FPGA-Chiplet*) para operaciones *sparse-dense*. En la primera parte del artículo ponen de manifiesto cómo de intensivas en términos de memoria son las *EmbeddingBags* y cómo de intensivas en cómputo son las MLPs. *Centaur* propone un subsistema de comunicación entre *chiplets* mediante un bus PCIe, donde los módulos encargados de realizar las operaciones *sparse-dense* han sido diseñados específicamente dentro de la *FPGA*.

Por otro lado, en otro trabajo presentado por *Facebook* [15], se pone de manifiesto la poca atención que han recibido las arquitecturas de los sistemas de recomendación, en especial la del modelo DLRM, a pesar de su importancia. Así, en este trabajo se presentan diferentes cargas de trabajo con el objetivo de demostrar la ineficiencia de los diferentes componentes. Estas cargas de trabajo se asemejan a cargas reales que podrían estar siendo ejecutadas en centros de computación actuales.

Un estudio de la literatura acerca de aceleradores para DNNs especializados en cómputo dense y sparse, nos lleva a diseños como el de la TPU, MAERI y SIGMA. Para procesamiento *dense*, serían adecuadas tanto la arquitectura TPU, como la de MAERI, mientras que para cómputo *sparse* está más especializada la arquitectura SIGMA, ya que permite procesar accesos a memoria en datos formateados en CSR (clave para el procesamiento de DLRM). Al ser tanto MAERI como SIGMA arquitecturas reconfigurables en tiempo de compilación, es decir, se pueden modificar las conexiones entre los distintos componentes de procesamiento (mediante pequeños conmutadores en sus redes de interconexión) para que el flujo de datos y procesamiento se adapte mejor a las características de cada etapa computacional (por ejemplo, una capa de una red neuronal), se van a usar MAERI y SIGMA para implementar un acelerador híbrido para acelerar la fase de inferencia de DLRM.

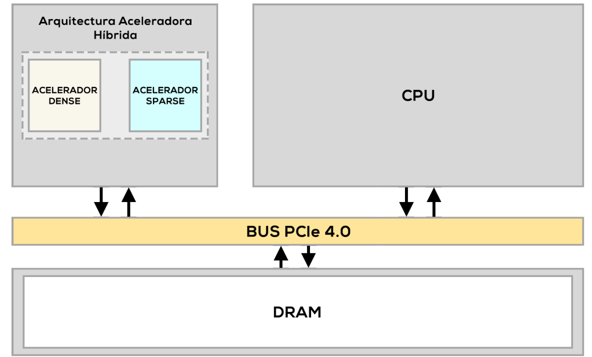

Fig. 2: Arquitectura híbrida propuesta.

La Tabla I muestra las características de cada una de las arquitecturas aceleradoras mencionadas.

III. Integración de DLRM en STONNE

La Figura 2 muestra los componentes básicos del acelerador híbrido que proponemos para acelerar la fase de inferencia de DLRM. Este acelerador híbrido integra dos aceleradores de uso específico para inferencia, uno para el cómputo dense (es decir, procesar las *MLPs*) y otro pare el cómputo *sparse* (es decir, procesar las *EmbeddingBags*). Como veremos, estos dos aceleradores se instanciarán en MAERI y SIGMA, respectivamente.

Para simular este acelerador y poder realizar la evaluación de DLRM, hemos partido de la herramienta de simulación STONNE[1], la cual nos permite modelar arquitecturas aceleradoras recientes tales como *MAERI*, *SIGMA* y la *TPU*. Además, nos permite hacer ejecuciones *end-to-end* de modelos de DL (e.g., las DNNs *Alexnet*, *Squeezenet*, *BERT*, etc). A pesar de esto, el simulador STONNE actualmente no está adaptado para implementar la arquitectura del acelerador híbrido propuesto.

Por otra parte, tenemos el modelo DLRM, desarrollado en *PyTorch*, que cuenta con 2 componentes fundamentales: las MLPs y las EmbeddingBags. STONNE no da soporte para DLRM, por lo que, en primer lugar tenemos que implementar las capas de DLRM necesarias y realizar una validación *end-to-end* (un benchmark) para comprobar su funcionamiento. Posteriormente, combinar la ejecución de ambos aceleradores para poder dar soporte completo a la ejecución del modelo DLRM.

[1]Repositorio - https://github.com/stonne-simulator/stonne

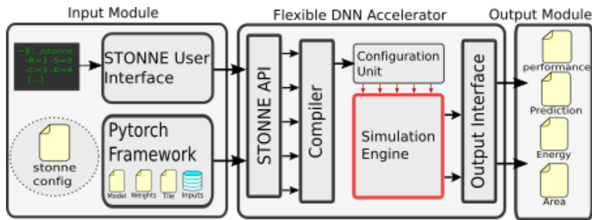

Fig. 3: El framework STONNE [17]

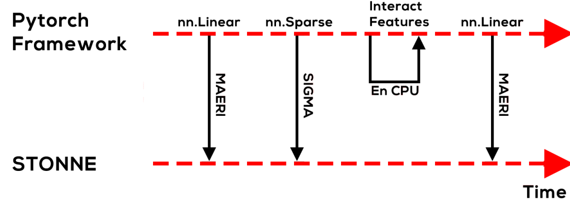

Fig. 4: Mapeo del modelo DLRM en la herramienta STONNE con las arquitecturas aceleradores MAERI y SIGMA.

STONNE permite conectarse a cualquier modelo de DL y actuar como acelerador (como si fuera una GPU o TPU real). En más detalle, la Figura 3 muestra los bloques principales de STONNE. Cuenta con un *front-end* escrito en *PyTorch*, que permite compilar código escrito en este lenguaje a código escrito en C++ para su posterior ejecución en el simulador. Mediante llamadas a esta API lograremos que el cómputo de las diferentes fases se realice en el simulador utilizando una arquitectura aceleradora (el *Simulation Engine*).

Como se explicó en la Sección II-A, el modelo DLRM cuenta con tres fases de ejecución bien identificadas: las MLPs, las *EmbeddingBags* y una fase denominada *Feature-Interaction*. Tras evaluar y comprobar el impacto en el cómputo que supone ejecutar esta última fase en CPU, se ha decidido no implementarla dentro del simulador por su baja influencia de cómputo. Como ya hemos mencionado, STONNE no da soporte para el modelo de recomendación DLRM, ya que la fase para el cómputo *sparse* (*EmbeddingBags*) no se encuentra implementada. Sin embargo, el simulador sí cuenta con la ejecución de MLPs de manera nativa (llamando a la API se puede realizar el cómputo *dense*), por lo que no va a ser necesario su integración.

Una vez llegados a este punto solo necesitaríamos integrar el cómputo *sparse*. Tenemos que dotar a STONNE de la funcionalidad para poder llamar a la API y que esta ejecute el algoritmo de las *EmbeddingBags*. Para este algoritmo necesitamos, primero, un conversor CSR que permita transformar los vectores CSR (*indices y offsets*) a un vector denominado *multi-hot encoding*. Y posteriormente una unidad de cómputo que se encargue de realizar una operación GEMM sparse del vector *multi-hot encoding* por la matriz de pesos de la *EmbeddingBag*. Este es el cómputo *sparse* del modelo.

Para realizar esta integración se ha duplicado la librería encargada de realizar las *EmbeddingBags*, llamada *nn.Sparse*, por *SimulatedSparse* (nos referiremos a ella como nn.Sparse por simplicidad). En el *forwarding* de la clase *EmbeddingBag* se ha llamado a la API de STONNE encargada de realizar esa operación GEMM sparse enviando como parámetros la matriz de pesos, el vector *multi-hot encoding* y otros parámetros necesarios del simulador. Una vez ambos componentes (MLPs y *EmbeddingBags*) han sido implementados, tendremos que especificar en STONNE una arquitectura a simular por STONNE (fichero *stonne config* de la Figura 3).

Existe una peculiaridad que es muy importante destacar de cara a la evaluación. En la *Top-MLP* además de las capas que especificaremos cuando confeccionemos los experimentos, el modelo DLRM realiza un cálculo interno; en base al número de *EmbeddingBags* y a la última capa de la *Bottom-MLP*, el modelo añadirá una capa extra con un determinado número de neuronas a la *Top-MLP*. Esta capa extra se ejecutará previamente al resto de las capas que nosotros le indiquemos.

IV. Evaluación

El simulador STONNE permite diferentes configuraciones para los aceleradores que simula. A continuación, pondremos de manifiesto qué metodología se ha llevado a cabo para realizar ciertos experimentos y posteriormente los resultados que hemos obtenido de integrar este modelo de recomendación dentro del simulador.

A. *Metodología*

Antes de presentar la metodología que vamos a seguir, es necesario contextualizar qué versión de *Python* y *frameworks* vamos a usar. En el caso de *Python* se ha utilizado la versión 3.8.0, el framework de *PyTorch* 1.7.0 y finalmente, para la gestión de librerías se ha usado *Anaconda* en la versión 4.9.2.

STONNE nos permite obtener un reporte de estadísticas de ejecución muy detallado tales como el número de ciclos de ejecución de cada componente de la arquitectura, el número de lecturas y de escrituras a las unidades de memoria internas (e.g., *Global Buffer*), el número de bits transmitidos por las redes dentro del chip, entre otras.

Como se ha mencionado en la Sección II-B, en este artículo se ha usado la arquitectura aceleradora *MAERI* [8] para el cómputo *dense* (MLPs) y la arquitectura aceleradora *SIGMA* [9] para el cómputo *sparse* de las *EmbeddingBags*. En la Figura 4 se puede ver la ejecución del modelo DLRM ya integrado completamente en STONNE. Además se representa la asignación de framework-acelerador para una ejecución *end-to-end* del modelo.

Para que el simulador STONNE simule dichas arquitecturas aceleradoras es necesario que se lo especifiquemos a través de un fichero de configuración (i.e., *stonne config*). Puesto que hemos usado las arquitecturas *MAERI* y *SIGMA*, cada una respectivamente para un componente diferente del modelo, hemos utilizado dos ficheros diferentes de configuración. En estos ficheros se especifican parámetros como el tamaño de los multiplicadores, el tipo de red de

	Número *PEs*	*Bandwidth* RD	*Bandwidth* RR	CM
MAERI	256	128	128	DENSE
SIGMA	128	128	128	SPARSE

Tabla II: Archivo de configuración para las arquitecturas. RR - Red de reducción. RD - Red de distribución. CM - Controlador de memoria

reducción, ancho de banda de la red de distribución y reducción, el controlador de memoria, etc.

En la Tabla II se muestran los parámetros más relevantes que hemos configurado para simular las arquitecturas *MAERI* y *SIGMA* de las que consta el acelerador híbrido propuesto. Como podemos observar, hemos escogido valores de parámetros similares a los considerados en sus trabajos. La diferencia principal entre ambas configuraciones se observa en el número de elementos de procesamiento (en inglés *Processing Elements* o *PEs*). En el caso de MAERI, hemos utilizado 256 PEs, mientras que en *SIGMA* este valor es reducido a la mitad, puesto que al explotar el *sparsity* se necesitan menos unidades de cómputo (i.e., se necesitan procesar menos operaciones en cada fase debido al gran número de operaciones que son innecesarias).

Además, puesto que el objetivo de este trabajo es el de realizar un análisis detallado del flujo de datos on-chip de la ejecución de DLRM sobre nuestras arquitecturas aceleradoras, para evitar entrar en resultados ligados a una jerarquía de memoria concreta entre el Global Buffer y la memoria principal, vamos a asumir un tamaño de Global Buffer ilimitado.

Para validar la integración del modelo en el simulador se han usado cargas de datos sintéticas similares a las propuestas en el artículo de *Centaur*. Los resultados de estas pruebas de validación han puesto de manifiesto la intensidad de cómputo de las MLPs y de memoria de las *EmbeddingBags*. Además, la mayor o menor intensidad de cómputo de las MLPs viene dada por el tamaño de *EmbeddingBags* que se use. Por este razonamiento hemos decidido mantener un valor fijo para ambas MLPs, teniendo en cuenta la capa extra que añade el modelo[2].

Para estudiar cargas de trabajo de DLRM similares a cargas reales necesitamos crear un modelo de recomendación con cierta envergadura en cuanto a número de *EmbeddingBags*, tamaño de las MLPs, etc. Para ello ha sido necesario investigar qué parámetros pueden modificarse y qué ejemplos de cargas de trabajo han sido propuestas por otros

[2]Parámetros *mlp-bot* = 128-64-64-32 y *mlp-top* = 128-64-1

Parámetro	Funcionalidad
embedding-size	Determina el número de *embeddings* que existirán y por cada embedding, cuántos índices(filas) tendrá.
mlp-{bot, top}	Indica cuántas capas y las neuronas por capa.
data-size	Determina el número de *lookups* que se realizarán en cada *EmbeddingBag*.
num-indices-lookup	Indica cuántos índices de la matriz de pesos de la *EmbeddingBag* se podrán seleccionar en cada *lookup*.

Tabla III: Parámetros configurables del modelo DLRM.

Grupo	Ejecución	Tam. *EmbeddingBag*
1	10-1M-20-100	128 *MBs*
	25-1M-20-100	128 *MBs*
	100-1M-20-100	128 *MBs*
2	10-1M-80-100	128 *MBs*
	25-1M-80-100	128 *MBs*
	50-1M-80-100	128 *MBs*
3	50-10M-20-100	1,28 *GBs*
	50-10M-80-100	1,28 *GBs*
	50-10M-20-1000	1,28 *GBs*
	50-10M-80-1000	1,28 *GBs*

Tabla IV: Cargas de trabajo sobre el modelo DLRM.

artículos [1], [2], [15]. En la Tabla III figuran los parámetros que vamos a modificar y el significado de cada uno de ellos.

Para referirnos a las diferentes cargas de trabajo dentro de cada grupo usaremos un sistema de nombres; el nombre de cada experimento vendrá definido por el número de *EmbeddingBags*, el número de índices por cada *EmbeddingBag*, su *data-size* y el *num-indices-lookup*.

Finalmente, en la Tabla IV presentamos las diferentes cargas de trabajo que vamos a ejecutar sobre el modelo DLRM. Se han usado tres grupos para clasificar los experimentos, el primero representa una carga de trabajo de "peso ligero", donde se usan 10, 25 y 50 *EmbeddingBags* respectivamente, manteniendo el parámetro *data-size* con un valor de 20. El segundo grupo representa una carga de "peso medio". El número de *EmbeddingBags* sigue siendo 10, 25 y 50 respectivamente, pero el parámetro *data-size* aumenta a 80. Esto tiene la finalidad de estudiar cómo afecta este parámetro a las lecturas y escrituras al *Global Buffer*. Finalmente, el tercer grupo representa una carga de trabajo "pesada", donde el número de *EmbeddingBags* es 50 en todos los casos, aumentamos el número de índices por *EmbeddingBag* y alteramos el valor *data-size*. Finalmente, en los dos últimos experimentos de este cuarto grupo modificamos el parámetro *num-indices-lookup* para estudiar su comportamiento sobre el modelo.

En los dos primeros grupos, el tamaño de cada *EmbeddingBag* es de 128 *MBs* y el de cada una de las MLPs es de 57 *KBs*. En el último grupo, las *EmbeddingBags* aumentan a 1,28 *GBs* mientras que las MLPs mantienen su tamaño inicial. Con este último experimento queremos poner a prueba el modelo DLRM, para ver cómo se comporta ante cargas de trabajo similares a las que podrían ejecutarse en un entorno real.

B. Resultados

A continuación presentamos los resultados obtenidos de los diferentes casos de estudio que hemos presentado anteriormente. En la Figura 5 se muestra el número de ciclos consumidos por cada una de las fases de ejecución, teniendo en cuenta el término MLPs como la suma de ciclos para la *Bottom-MLP* y la *Top-MLP*.

A primera vista, podemos observar como las MLPs

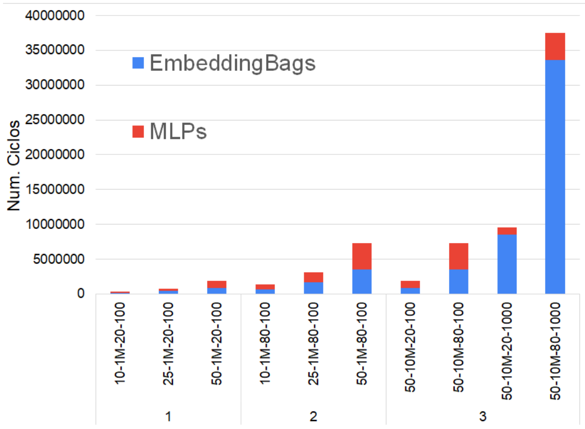

Fig. 5: Número de ciclos obtenidos.

(*Bottom* y *Top*) son intensivas en cómputo (ocupan la mayor parte de los ciclos). Si comparamos los ciclos de ejecución de las *EmbeddingBags* con los de las MLPs (experimentos del grupo 1 y 2), observamos cómo el cómputo *sparse* toma aproximadamente un 35-40 % del cómputo total, estos números se corresponden con los resultados esperados [2], [15]. Sin embargo, cuando aumentamos considerablemente el tamaño de las *EmbeddingBags*, al ser éstas tan intensivas en memoria, en los últimos experimentos (últimos dos experimentos del grupo 3) los ciclos de cómputo de este componente se llevan la mayor parte.

Además de obtener el número de ciclos para cada experimento, también se ha confeccionado una gráfica donde se muestra el número de escrituras y lecturas que se han realizado al *Global Buffer*. En la Figura 6 se muestra la gráfica. En este primer grupo de experimentos de carga ligera vamos a aumentar considerablemente el número de *EmbeddingBags*: comenzaremos por 10 y acabaremos con 50. Nos centraremos en observar la forma en la que el modelo se comporta conforme se aumenta este parámetro y mantenemos el resto estáticos.

Como se observa, las lecturas de la *Bottom-MLP* son las mismas para los tres experimentos. Esto se debe a los parámetros *mlp-bot* y *data-size*, cuyos valores no se modifican en este primer grupo, y por lo tanto, el número de lecturas para la *Bottom-MLP* va a ser el mismo en los tres casos. Con esto ponemos de manifiesto la relación que existe entre estos parámetros para determinar el número de lecturas de la *Bottom-MLP*.

Sin embargo, las lecturas de la *Top-MLP* sí varían en los tres experimentos. A pesar de especificar un tamaño fijo para la *Top-MLP*, si recordamos, el propio modelo DLRM realiza una operación en la que teniendo en cuenta el número de *EmbeddingBags* y el número de neuronas de la última capa de la *Bottom-MLP* añade una capa extra a la *Top-MLP*. Como en los tres casos aumentamos el número de *EmbeddingBags*, las neuronas de esta capa extra es diferente en los tres casos, como resultado, el número de lecturas es diferente.

Además, hay que tener en cuenta que el cómputo que ocurre en esta *Top-MLP* viene condicionado por el resultado de la fase anterior, *Feature-Interaction*. Dado que en esta fase se procesa la matriz resultante del algoritmo *EmbeddingBag*, al aumentar el número de *EmbeddingBags* o *data-size*, se aumenta el tamaño de la matriz resultante, por ende, las lecturas que se realizan en la *Top-MLP* también aumentan.

En el caso de las escrituras, dado que el mayor número se encuentra en las *EmbeddingBags*, se observa que estas se incrementan al aumentar el número de tablas. E concreto, al aumentar el número de *EmbeddingBags* se realizarán más operaciones GEMM sparse, como consecuencia, se generarán más escrituras.

En el caso de la *Bottom-MLP* y *Top-MLP*, ambos componentes realizan el mismo número de escrituras en los tres experimentos. A pesar de añadir una capa extra y diferente a la *Top-MLP* en cada experimento, esta realiza las mismas escrituras. Cuando presentemos el siguiente grupo, analizaremos los resultados y los compararemos con los obtenidos.

Teniendo en mente este primer grupo, vamos a introducir el grupo 2 con una carga de trabajo media. Se va a seguir la misma práctica que en el grupo 1, es decir vamos a ir aumentando el número de *EmbeddingBags*, comenzando por 10 y terminando con 50. Sin embargo, en vez de tener un valor de 20 para el parámetro *data-size*, vamos a aumentarlo a 80 (4×). Es decir, en vez de realizar 20 búsquedas por *EmbeddingBag*, vamos a realizar 80.

Si nos fijamos en la gráfica de las lecturas (gráfica superior de la Figura 6b) y la comparamos con la obtenida del grupo 1 (Figura 6a) vemos que comparte un aspecto similar. Las lecturas crecen de igual forma. En el caso de las lecturas de la *Bottom-MLP*, si aplicamos la teoría que hemos desarrollado antes, puesto que este vector tiene que tener la misma dimensión (índices) que la matriz resultante de las *EmbeddingBags* (columnas), al aumentar el número de búsquedas por *EmbeddingBag*, el vector dense que genera la *Bottom-MLP* ha de ser mayor. Concluyendo en que, al aumentar el parámetro *data-size* de 20 a 80 (4×) el número de lecturas de la *Bottom-MLP* aumenta en el mismo grado (4×) en los tres experimentos de este segundo grupo.

De este modo, corroboramos que las lecturas asociadas a este componente *Bottom-MLP* van en función de los parámetros *data-size* y *mlp-bot*.

Para las lecturas de las *EmbeddingBags*, al mantener el resto de parámetros estáticos y realizar 4 veces más búsquedas por *EmbeddingBag*, el número de lecturas aumenta notablemente y sigue un crecimiento similar a los resultados obtenidos en las lecturas de las *EmbeddingBags* del grupo 1 (Figura 6a).

Para el caso de la *Top-MLP* seguimos en la línea de lo ya comentado anteriormente. Al no modificar el número de *EmbeddingBags* en cada experimento y mantener el mismo tamaño de *Bottom-MLP*, esta capa extra que añade el modelo DLRM a la *Top-MLP* no se modifica y se aprecia un crecimiento similar al obtenido en el grupo 1.

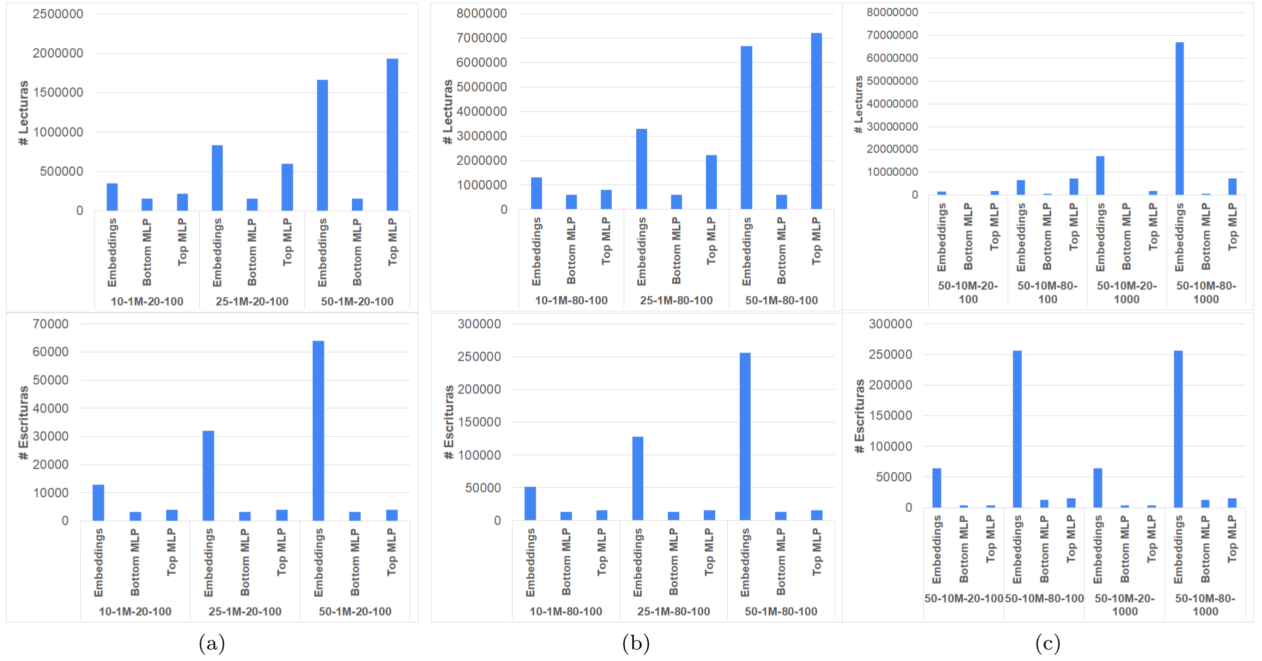

Fig. 6: (a) L/E para el grupo 1. (b) L/E para el grupo 2. (c) L/E para el grupo 3. Todo en valor absoluto.

Para el caso de las escrituras, como el número de *EmbeddingBags* se mantiene entre ambos grupos, el número de escrituras va a ser el mismo, con la peculiaridad de que, al realizar 4 búsquedas más, el número de escrituras crece exactamente 4 veces más con respecto a los resultados obtenidos del grupo 1.

Como el valor de los parámetros *mlp-bpt* y *mlp-top* no se ha modificado y sabemos que las escrituras varían en función del valor de *data-size*, al haber aumentado este valor cuatro veces, las escrituras obtenidas en este grupo 2, son similares a las obtenidas en el grupo 1 multiplicadas por cuatro.

Estos experimentos nos sirven para darnos cuenta de cómo el parámetro *data-size* o búsquedas/*embedding* afecta al cómputo del modelo.

Hasta ahora hemos visto que le número de ciclos de ejecución que consumen las MLPs siempre solía ser mayor que el de las propias *EmbeddingBags*, y esto tiene sentido, ya que las MLPs son intensivas en cómputo. Sin embargo, hacemos especial atención a la necesidad de soportar grandes cargas de datos. Para ver cómo el modelo se comporta introducimos el último grupo.

En este tercer grupo introducimos una serie de experimentos con una carga de trabajo pesada, aproximando lo que podría considerarse como un *dataset* real.

Si nos fijamos en las dos primeras ejecuciones de este grupo, a pesar de haber aumentado 10 veces el número de índices de las *EmbeddingBags*, estas siguen sin suponer un gran porcentaje de los ciclos totales de cómputo (ocupan entorno al 46 % y 47 % de los ciclos totales). Este porcentaje de cómputo es el mismo que el obtenido en anteriores experimentos a pesar de aumentar las *EmbeddingBags* a 1,28 *GBs* (Figura 5).

En este punto introducimos el parámetro *num-indices-lookup*. Hasta ahora, para tablas de hasta 1 millón de índices este valor ha sido siempre de 100 unidades, lo que significa que por cada *lookup*/búsqueda que se realiza en la *EmbeddingBag*, como máximo se pueden coger 100 índices (0.0001 % de 1 millón de índices). Con el fin de realizar una mejor predicción, tendríamos que coger más índices, lo que se refleja en realizar más *lookups*. En los últimos dos experimentos se ha incrementado este número a 1000 y ha supuesto un impacto en el porcentaje de ciclos totales consumidos por las *EmbeddingBags* de un 89 % y 87 % respectivamente.

Basándonos en todo lo explicado anteriormente, vamos a explicar cómo este parámetro afecta al cómputo del modelo. Las lecturas de la *Bottom-MLP* y *Top-MLP* son iguales para el primer y tercer experimento, y para el segundo y cuarto, respectivamente (Figura 6c). Si nos fijamos, se mantienen los mismos parámetros que para el resto de experimentos (tercer caso de estudio del grupo 1 Figura 6a y grupo 2 Figura 6b), solo variamos el parámetro *num-indices-lookup*. Concluimos con las lecturas diciendo que este parámetro no afecta al número de lecturas en las MLPs.

Si nos fijamos en las *EmbeddingBags*, al aumentar el número de índices de las *EmbeddingBags* y el parámetro *num-indices-lookup*, todo esto acompañado de aumentar el parámetro *data-size*, número de *EmbeddingBags*, etc. se observa un gran aumento en el número de lecturas, aproximadamente en un factor de 10 (10×), puesto que hemos aumentado el parámetro *num-indices-lookup*. Este parámetro genera un fuerte impacto en el cómputo de las *EmbeddingBags* y el número de lecturas.

En el caso de las escrituras, seguimos en línea con lo comentado anteriormente. El número de escrituras de las MLPs viene determinado por el parámetro

data-size, *mlp-bot* y *mlp-top*. Puesto que no se modifican, se obtienen las mismas escrituras que en los grupos anteriores. Las escrituras de las MLPs para los experimentos 1 y 3 de este grupo, se corresponden con las obtenidas en el grupo 1 (tercer caso de estudio Figura 6a); y los experimentos 2 y 4 con las obtenidas en el grupo 2 (tercer caso de estudio Figura 6b).

Para las escrituras de las *EmbeddingBags* ocurre algo peculiar. Las escrituras en los experimentos 1 y 3 de este grupo, se corresponden con las obtenidas en el experimento 3 del grupo 1 (Figura 6a) (mismo valor del *data-size* (20) y mismo número de *EmbedingBags*(50). Por otra parte, las escrituras obtenidas en los experimentos 2 y 4 (los más intensivos en cómputo), se observa como las escrituras se corresponden con las que se han obtenido en el experimento 3 del grupo 2 (Figura 6b) (mismo *data-size* y mismo número de *EmbeddingBags*).

C. Resumen del Análisis de Resultados

A continuación presentamos un resumen de las conclusiones obtenidas tras haber realizado los experimentos:

- Si comparamos el tiempo total empleado por los diferentes experimentos, el componente más costoso, donde más tiempo se ha empleado, ha sido en el procesamiento de las *EmbeddingBags*.
- En el caso de las lecturas, sabemos que el mayor o menor número de lecturas en las *EmbeddingBags* viene condicionado por el número de *EmbeddingBags*, de los índices de cada *EmbeddingBag*, del parámetro *data-size* y del parámetro *num-indices-lookup*.
- En el caso de la *Bottom-MLP*, las lecturas vienen determinadas por el parámetro *data-size* y por el tamaño del parámetro *mlp-bot*, que determina el número de capas y las neuronas por capa.
- Para la *Top-MLP*, recordemos que el modelo añade una capa extra previa a las que añadimos nosotros. De esta manera, las lecturas vienen condicionadas por el número de *EmbeddingBags*, las neuronas de la última capa de la *Bot-MLP* y el parámetro *data-size*. Obviamente, dejando estos parámetros estáticos y modificando el valor del parámetro *mlp-top*, también se verá reflejado un aumento o disminución en las lecturas.
- El número de escrituras en las *EmbeddingBags* viene determinado por el número de *EmbeddingBags* que existan y por el parámetro *data-size*. El aumento del parámetro *num-indices-lookup* ha demostrado no suponer ningún tipo de impacto en las escrituras.
- Para el caso de las MLPs es mas sencillo, puesto que las escrituras vienen determinadas por el número de capas, es decir, *mlp-*{*top,bot*} y el parámetro *data-size*.

V. Conclusiones

En este trabajo hemos realizado una evaluación de la ejecución del proceso de inferencia de DLRM sobre un acelerador híbrido propuesto simulado con STONNE. Este acelerador híbrido está compuesto por dos aceleradores, uno para cómputo *sparse* y otro para cómputo *dense*, de uso específico para inferencia. Hemos escogido las arquitecturas MAERI y SIGMA, ya que son reconfigurables y ofrecen notables ventajas respecto a otras arquitecturas, como podría ser la TPU.

De los resultados obtenidos podemos observar cómo el tiempo de ejecución aumenta notablemente al aumentar el tamaño y número de *EmbeddingBags*, así como el *data-size*. Por otra parte, las lecturas y escrituras de los diferentes componentes del modelo están correlacionadas con el tamaño de las *EmbeddingBags*. Como ya hemos visto en los experimentos, el mayor o menor número de ciclos, escrituras o lecturas durante el procesamiento viene dado por el tamaño de *EmbeddingBags*, que es el componente responsable del cómputo *sparse*.

Gracias a los resultados de la evaluación realizada, como trabajo futuro se va a mejorar la arquitectura del acelerador híbrido propuesto implementando una jerarquía de memoria que soporte de forma eficiente la inmensa cantidad de lecturas y escrituras de memoria que el modelo DLRM necesita durante su procesamiento *sparse*.

Agradecimientos

Trabajo financiado por RTI2018-098156-B-C53 (MCIU/AEI/FEDER,UE), NSF OAC 1909900 y US Department of Energy ARIAA co-design center. Francisco Muñoz-Martínez ha sido financiado mediante la beca 20749/FPI/18 de Fundación Séneca, Agencia de Ciencia y Tecnología de la Región de Murcia.

Referencias

[1] Dheevatsa Mudigere Maxim Naumov et al., “Deep learning recommendation model for personalization and recommendation systems,” *CoRR*, vol. abs/1906.00091, 2019.

[2] Taehun Kim Ranggi Hwang et al., “Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations,” *CoRR*, vol. abs/2005.05968, 2020.

[3] Nishigandha Karbhari et al., “Recommendation system using content filtering: A case study for college campus placement,” in *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, 2017, pp. 963–965.

[4] Ben Schafer et al., “Collaborative filtering recommender systems,” 01 2007.

[5] Robert M Bell and Yehuda Koren, “Improved neighborhood-based collaborative filtering,” in *KDD cup and workshop at the 13th ACM SIGKDD international conference on knowledge discovery and data mining*. Citeseer, 2007, pp. 7–14.

[6] Yehuda Koren, “Factorization meets the neighborhood: A multifaceted collaborative filtering model,” New York, NY, USA, 2008, KDD ’08, p. 426–434, Association for Computing Machinery.

[7] Norman P. Jouppi et al., “In-datacenter performance analysis of a tensor processing unit,” in *44th Int’l Symp. on Computer Architecture (ISCA)*, June 2017, pp. 1–12.

[8] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna, “MAERI: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects,” *Int’l*

Conf. on Architectural Support for Programming Languages and Operating Systems, Mar. 2018.

[9] Eric Qin et al., "SIGMA: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," *Int'l Symp. on High-Performance Computer Architecture*, Mar. 2020.

[10] Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K. Gupta, and Hadi Esmaeilzadeh, "SnaPEA: Predictive early activation for reducing computation in deep convolutional neural networks," *International Symposium on Computer Architecture (ISCA)*, pp. 662–673, June 2018.

[11] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292 – 308, June 2019.

[12] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," *International Symposium on Computer Architecture (ISCA)*, pp. 27–40, June 2017.

[13] Norman P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *44th Int'l Symp. on Computer Architecture*, 2017.

[14] Malay Haldar et al., "Applying deep learning to airbnb search," New York, NY, USA, 2019, KDD '19, p. 1927–1935, Association for Computing Machinery.

[15] Xiaodong Wang Udit Gupta et al., "The architectural implications of facebook's dnn-based personalized recommendation," *CoRR*, vol. abs/1906.03109, 2019.

[16] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie, "Hygcn: A gcn accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 15–29.

[17] Francisco Muñoz Matrínez, José L. Abellán, Manuel E. Acacio, and Tushar Krishna, "Stonne: A detailed architectural simulator for flexible neural network accelerators," *arXiv preprint arXiv:2006.07137v1*, June 2020.