Universidad de Murcia

Facultad de Informática

Técnicas Hardware para el Diseño de Aceleradores de Inferencia Eficientes de Redes Neuronales Profundas

Hardware Techniques for the Design of Efficient Inference Accelerators of Deep Neural Networks

Francisco Muñoz Martínez

2022

Universidad de Murcia

Facultad de Informática

Departamento de Ingeniería y
Tecnología de Computadores

Técnicas Hardware para el Diseño de Aceleradores de Inferencia Eficientes de Redes Neuronales Profundas

Tesis Doctoral

Autor:
Francisco Muñoz Martínez

Directores:
Manuel Eugenio Acacio Sánchez
José Luis Abellán Miguel

Murcia, Septiembre de 2022

Resumen

Durante los últimos 50 años, la potencia de cómputo proporcionada por los procesadores de propósito general ha experimentado un increíble incremento de rendimiento debido principalmente a la mejora de la tecnología que se utiliza para construir los chips, cumpliendo la ley empírica predicha por Gordon Moore en el año 1975 [38] que indicaba que el número de transistores en un chip se duplicaría cada dos años. Los arquitectos hardware, se han aprovechado de este hecho diseñando cauces de procesamiento cada vez más complejos o en la última década añadiendo más y más núcleos de procesamiento en un mismo chip. Desafortunadamente, debido a restricciones físicas prácticamente insuperables, el tamaño del transistor está llegando a su límite y por tanto el incremento de la potencia de cómputo está siendo comprometida, duplicándose según las tendencias actuales cada 20 años [59]. Además, este límite es alcanzado en el momento más demandante de cómputo de la historia de los computadores: la era de la inteligencia artificial, y en concreto de las redes neuronales profundas (*Deep Neural Networks* o DNNs).

Estas DNNs constituyen hoy en día un avance disruptivo para un gran número de aplicaciones de inteligencia artificial, como reconocimiento de imagen y vídeo, conducción autónoma, reconocimiento del lenguaje natural, traducción de texto, detección temprana de enfermedades, predicción meteorológica, etc. [117]. El inconveniente es que estas DNNs requieren de gran cantidad de datos y realizan millones de operaciones, demandando una masiva cantidad de cómputo, uso de memoria y energía.

Generalmente, una DNN tiene dos fases fundamentales de operación: una primera fase de *entrenamiento*, donde un conjunto de pesos son derivados 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. Generalmente, la fase de entrenamiento se lleva a

cabo utilizando GPUs en grandes centros de datos [20], 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 [61], [76], [70], [31], [33].

La clave que hay detrás de todas estas arquitecturas aceleradoras ha sido capturar los diferentes patrones de datos en lo que se conoce como flujo de datos (*dataflow*) [68], y la explotación de optimizaciones arquitecturales basadas en los datos de entrada para reducir el cómputo o el movimiento de datos [33].

La primera generación de aceleradores *sistólicos* del procedimiento de inferencia para DNNs (e.g., la TPU de Google [61]) basaban su diseño en grupos (clústeres) de unidades de multiplicación y suma (unidades MACs) de tamaño fijo, interconectados mediante una red de interconexión específicamente diseñada para soportar un flujo de datos particular [23].

Desafortunadamente, con el avance y desarrollo de nuevos modelos DNNs, estos aceleradores sistólicos se han quedado obsoletos, debido principalmente a su incapacidad de adaptarse a nuevas características como: i) mayor variedad de tipos de DNNs, lo cual conlleva a mayor heterogeneidad en las demandas de cómputo; y ii) la presencia de sparsity (gran cantidad de valores cero), que viene siendo cada vez más notable en estos modelos. Explotar estas nuevas características hace que los aceleradores sistólicos compuestos por topologías *rígidas* no puedan adaptarse, traduciéndose en una baja utilización de las unidades de cómputo y en dificultad para escalar adecuadamente estas arquitecturas aceleradoras.

Con el objetivo de superar estas limitaciones, recientemente han surgido diseños de arquitecturas aceleradoras como FlexFlow [76], MAERI [70] o SIGMA [31]. Estos aceleradores utilizan topologías *flexibles* capaces de adaptar su sustrato hardware a las distintas demandas computacionales requeridas por los nuevos modelos de DNNs. Para ello, estas arquitecturas permiten configurarse para crear cualquier número de clústeres de unidades de cómputo y de cualquier tamaño, permitiendo una mejor adaptación a cualquier demanda en el mismo sustrato hardware.

Naturalmente, estas arquitecturas flexibles descritas anteriormente, aunque mejoran significativamente el aprovechamiento de las unidades de cómputo, son bastante más complejas que una simple arquitectura sistólica. Además, el número de mapeos que se pueden utilizar por cada capa es casi infinito y el espectro de evaluación y mejora del diseño hardware es mucho más amplio que en una arquitectura simple. Por este motivo, para los arquitectos de computadores es inviable utilizar un diseño hardware directamente escrito en HDL para realizar pruebas o evaluaciones sobre el mismo.

Hasta ahora, los arquitectos de computadores han utilizado herramientas analíticas para evaluar sus diseños, empleando un conjunto de fórmulas que les permiten estimar el rendimiento y eficiencia energética en unos pocos segundos. SCALE-Sim [30], MAESTRO [68] y TimeLoop [29] han sido propuestos recientemente como ejemplos de este tipo de herramientas que permiten el análisis de diferentes flujos de datos en arquitecturas específicas para DNNs. Estas herramientas son muy potentes para explorar de forma rápida los detalles de alto nivel de la arquitectura, ya que se basan en modelos analíticos que calculan una estimación del número de ciclos de ejecución, grado de reutilización de datos y eficiencia energética mediante un conjunto de fórmulas sencillas. Este tipo de herramientas funcionan con precisión cuando se trata de arquitecturas rígidas, ya que son lo suficientemente simples como para ser representadas por un conjunto de fórmulas. Sin embargo, cuando la complejidad del acelerador crece y/o el cómputo no sigue patrones regulares, estos modelos arquitecturales no logran capturar fielmente el comportamiento exacto de la arquitectura. Esto hace, como demostramos en esta tesis, que la precisión que ofrecen estas herramientas esté muy lejos de ser óptima, ya que no son capaces de capturar detalles microarquitecturales que tienen su efecto durante las ejecuciones reales. Además, estos modelos analíticos no ejecutan realmente un modelo de DNN, por lo que no pueden utilizar los datos de entrada (por ejemplo, el número y posición de los valores cero) para evaluar optimizaciones basadas en datos.

Es necesario, por tanto, al igual que ha ocurrido anteriormente con la evaluación de diseños de CPUs [14] y GPUs [13], [115], el uso de simuladores arquitecturales con precisión a nivel de ciclo que permitan realizar cambios en el diseño de forma rápida y sencilla, y poder comprobar el efecto de dichos cambios de forma precisa.

Para suplir esa demanda, como primera contribución de esta tesis diseñamos, presentamos y evaluamos el simulador STONNE (*Simulation TOol of Neural Network Engines*), un simulador microarquitectural a nivel de ciclo para aceleradores de inferencia de DNN liberado como código abierto bajo los términos de la licencia MIT. Con el objetivo de permitir evaluaciones de modelos de DNNs completos, STONNE está conectado con el conocido framework para DNNs PyTorch [5] (además de con Caffe [2]). De esta forma, STONNE puede ejecutar completamente cualquier modelo DNN denso y *sparse* soportado por el framework de DNNs que utiliza como front-end. El simulador ha sido escrito completamente en C++, siguiendo los conocidos principios de programación

GRASP y SOLID del diseño orientado a objetos [79]. Esto ha simplificado su desarrollo y facilita la implementación de cualquier tipo de módulo microarquitectural de acelerador de inferencia de DNN, mapeos de configuración flexibles o la incorporación de distintos flujos de datos.

STONNE está constituido por 3 módulos principales: una plataforma de simulación que constituye el bloque principal, ya que incluye la implementación de distintos bloques hardware con el objetivo de simular de forma precisa diferentes aceleradores tanto rígidos como flexibles. Un módulo de entrada que alimenta la plataforma de simulación utilizando cualquiera de los conocidos frameworks de DNNs ya disponibles (e.g. PyTorch [5]). Finalmente, la plataforma también cuenta con un módulo de salida que se utiliza para informar sobre las estadísticas de la simulación, tales como el rendimiento, la utilización de las unidades de cómputo, y el recuento de la actividad de los diferentes componentes hardware, tales como cables, colas FIFO o el uso de la SRAM (es decir, el número de accesos). Además, utilizando estas estadísticas de actividad, este módulo de salida también informa de la cantidad de energía consumida y del área en el chip requerida por la arquitectura simulada. Para ello, STONNE emplea un modelo de área y energía basado en una tabla similar al de Accelergy [119], calculando la energía total utilizando las estadísticas de actividad a nivel de ciclo para cada módulo y una tabla particular con los diferentes costes de energía y área por operación en cada módulo. Obviamente, estas estadísticas dependen, por ejemplo, del formato particular de datos utilizado para representar los parámetros del modelo DNN (por ejemplo, fp16 o int8). Así, STONNE trae consigo diferentes tablas con costes de energía que pueden ser utilizadas por el usuario.

En la plataforma de simulación de STONNE todos los componentes en el chip están interconectados utilizando tres redes de interconexión: 1) Red de distribución (*Distribution Network* o DN); 2) Red de multiplicación (*Multiplier Network* o MN); y 3) Red de reducción (*Reduction Network* o RN). Esta aproximación está inspirada en la taxonomía de los flujos de comunicación en el chip de los aceleradores de DNNs [70]. Estas redes pueden ser configuradas para soportar cualquier topología con el objetivo de modelar diversas arquiteturas de acelerador, como la TPU [61], Eyeriss-v2 [24], ShiDianNao [28], SCNN [98], MAERI [70] o SIGMA [31], entre otros.

Para ello, las tres redes pueden ser organizadas para modelar cualquier acelerador mediante un flujo de datos muy sencillo. En primer lugar, para calcular todas las operaciones MAC de una determinada capa de la DNN, la DN distribuye los pesos, activaciones y sumas parciales requeridas desde una memoria SRAM (*Global Buffer* o GB) hacia la MN. Para permitir todos los tipos de flujos de datos, la DN debe proporcionar soporte para la entrega de datos *unicast*, *multicast* y *broadcast*. Esto se consigue a través de diferentes posibles configuraciones de los conmutadores de distribución (*Distribution Switch* o DS) que integran la DN. Tras la distribución, los multiplicadores en la MN realizan las operaciones de multiplicación, generando un conjunto de sumas parciales a acumular. Finalmente, la red RN está equipada con sumadores que implementan las acumulaciones requeridas.

STONNE soporta distintas DNs como *Tree-Network*, *Benes-Network* o *Point-to-point network*. De forma similar, las MNs incluidas de forma nativa son la *Linear Multiplier Network* y la *Disabled Multiplier Network*. Finalmente, STONNE soporta distintas RNs para poder realizar acumulaciones de sumas parciales flexibles, lo que da soporte a distintos aceleradores. STONNE soporta una red de tipo árbol de reducción (*Reduction Tree* o RT) que denominamos árbol de reducción aumentado (ART+DIST), una red que combina ART y un *buffer* de Acumulación (ART+ACC) y una red de adelantamiento sumador (*Forwarding Adder Network* o FAN). Con el fin de soportar todos los tipos de aceleradores, en STONNE también implementamos una red de reducción lineal (*Linear Reduction Network* o LRN), que se utiliza en aceleradores rígidos como la TPU [61]. Esta red es uno de los componentes clave a la hora de desarrollar un acelerador, y nuestras contribuciones presentadas en los capítulos 3 y 4 tienen como clave el diseño de una nueva RN.

Como demostramos en el capítulo 2 de esta tesis, nuestro simulador STONNE obtiene un error de precisión con respecto al hardware real de tan solo entre el 0,14 % y el 3,10 % (1,53 % de media), demostrando que STONNE imita fielmente las características de las versiones hardware.

Para demostrar la usabilidad y utilidad del simulador, hemos llevado a cabo el desarrollo de tres casos de estudio. En el primer caso de estudio, hemos realizado una validación de tiempos entre los aceleradores de DNN nativamente modelados: MAERI [70], SIGMA [31] y la TPU [61]. En el segundo caso de estudio, demostramos cómo es posible modificar la plataforma de simulación de STONNE para añadir nuevos aceleradores. Para ello, modelamos el acelerador SnaPEA [33] mostrando la utilidad de STONNE para modelar aceleradores cuyas optimizaciones son dependientes de los datos de entrada. Finalmente, en el tercer caso de estudio mostramos cómo STONNE es útil para desarrollar nuevas técnicas a nivel de compilación. En concreto, en este caso de estudio mostramos que en aceleradores con soporte para *sparsity* como SIGMA [31] es primordial tener en cuenta el número de ceros que se encuentran en los filtros de una DNN y el orden en el que se ejecutan estos filtros. Ejecutar una DNN comprimida con formato *sparse* sin tener en cuenta este hecho implica que la ejecución paralela de varios filtros no sea balanceada. Así, aplicar una técnica de ordenamiento de los filtros antes de la ejecución de la DNN puede tener efectos positivos significativos en el rendimiento de su procesamiento. En este caso de estudio hemos comparado tres técnicas de ordenamiento: *Largest Filter First* (LFF), el cual consiste en ejecutar los filtros más grandes primero, *Random* (RDM) y sin ordenamiento (*non-scheduling* o NS). Entre todos, hemos encontrado que la técnica de ordenamiento LFF es la que mejor funciona, obteniendo mejoras de rendimiento de hasta 13 % en algunas capas.

La segunda contribución de esta tesis se centra en la RN de aceleradores flexibles como MAERI o SIGMA. En concreto, estos diseños proponen como clave principal una RN específica (ART y FAN, respectivamente) para lograr la reducción flexible de distintos grupos de sumas parciales generadas por la MN. Estos grupos de sumas parciales pueden variar en tamaño debido al *sparsity* de la capa de DNN o incluso a sus características internas como su tamaño. En caso de que el grupo de sumas parciales pueda ser mapeado por completo en los multiplicadores, su acumulación es muy sencilla: basta con acumular todos los valores de forma espacial, utilizando los acumuladores del árbol de reducción. El problema que no es abordado en estas propuestas anteriores es lo que ocurre cuando el tamaño del grupo de sumas parciales supera al número de multiplicadores mapeados. En este caso, sumar todos los valores de forma espacial no es suficiente, y hay que recurrir a una suma espacio-temporal. Es decir, dividir el grupo de sumas parciales en distintas iteraciones, acumular espacialmente cada iteración y luego acumular de forma temporal (i.e., en distintos instantes de tiempo) el resultado de cada iteración. Este problema no es abordado de forma correcta en las propuestas de MAERI y SIGMA. Para resolverlo, una opción consiste en utilizar simplemente la RN nativa como ART y reenviar la suma parcial obtenida espacialmente en una iteración a través del *Global Buffer* hacia un multiplicador, para ser calculadas en la siguiente iteración. Este mecanismo denominado como S-Tree tiene dos inconvenientes principales: el primero es que impide mapeos óptimos, ya que al mapear un grupo de sumas parciales hay que dejar espacio para un multiplicador extra que es utilizado para reenviar la suma parcial de la iteración anterior que llega del *Global Buffer*. El segundo inconveniente y principal es que se genera una dependencia temporal entre dos iteraciones consecutivas, impidiendo que las iteraciones puedan ser ejecutadas de forma paralela en *pipeline*. Esto degrada significativamente el rendimiento. La segunda forma de resolver el problema evitando esta degradación de rendimiento es utilizar un esquema ST-Tree que consiste en colocar un acumulador extra en cada nodo del árbol de reducción, permitiendo que la acumulación de distintas iteraciones pueda ser computada en paralelo junto con la acumulación espacial de las mismas. Esto se conoce como un *buffer* de acumulación y se utiliza en arquitecturas como la TPU de Google. El inconveniente de este esquema es que para lograr la flexibilidad que se propone en estas arquitecturas se necesita un acumulador en cada nodo del árbol de reducción, lo cual implica duplicar el número de unidades de suma, duplicando el área y el consumo energético del módulo de RN, el cual constituye hasta un 25 % y 38 % en área y energía, respectivamente, en aceleradores como MAERI y SIGMA. Esta sobrecarga es inasumible cuando el número de multiplicadores se incrementa en la arquitectura.

En esta tesis hemos diseñado una nueva propuesta de RN para solucionar este problema en arquitecturas flexibles. Nuestra propuesta se denomina STIFT (*Spatio-Temporal Integrated Folding Tree*) y consiste en utilizar un árbol de reducción similar a ART, pero añadiendo algunos enlaces extra entre los nodos, que aseguren que para cada nodo del árbol, siempre habrá un nodo conectado libre para realizar la acumulación. Esto permite aprovechar las unidades de suma ya existentes en el árbol y la única sobrecarga se debe a algunos enlaces extras y un nodo padre adicional en el árbol de reducción.

La evaluación de STIFT la hemos realizado desde tres ángulos diferentes: 1) una implementación RTL de S-Tree, ST-Tree y STIFT en Bluespec System Verilog [1] con el objetivo de comparar los resultados a nivel de área y energía. Para una evaluación exhaustiva hemos comparado los diseños para los tamaños de 64, 128, 256, 512 y 1024 multiplicadores. Además, comparamos los diseños para 3 tipos de datos distintos (INT16, FP16 y FP32).

Para evaluar nuestra propuesta a nivel de rendimiento, hemos implementado S-Tree, ST-Tree y STIFT en nuestro simulador STONNE, utilizando como sistema base la RN ART y utilizando 256 multiplicadores 108-KiB de SRAM, 128 elementos/ciclo de ancho de banda de SRAM, y dos módulos de DRAM HBM 2.0, de 512 MiB cada uno y con un ancho de banda de 256 GB/s. Con esta configuración, hemos utilizado un estudio sintético consistente en mapear distintos tamaños de grupos de sumas parciales, así como la ejecución de 7 modelos de DNN reales completos extraídos de la suite de benchmarks MLPerf [107].

Los resultados obtenidos revelan que ST-Tree y STIFT obtienen el mismo rendimiento para todas las ejecuciones sintéticas y DNNs ya que ambas son capaces de ejecutar de forma paralela las disintas iteraciones de sumas parciales. Ambas, obtienen una mejora de rendimiento de hasta 8$\times$ para DNNs como VGG-16. Sin embargo, STIFT obtiene beneficios en términos de área y energía de hasta 32 % y 31 %, con respecto ST-Tree ya que evita la duplicidad de las unidades de suma debido a su topología más inteligente. En otras palabras, STIFT logra una mejora de rendimiento/energía con respecto a S-Tree de 5,13×, mientras que ST-Tree solo logra un 3,67×.

Esta propuesta abordada en esta tesis supone un punto de inflexión en el diseño de aceleradores flexibles para DNNs ya que permite por primera vez ejecutar de forma eficiente grandes DNNs.

Estas propuestas anteriores como MAERI, SIGMA, nuestra propuesta STIFT o incluso otros aceleradores con soporte para ejecución de operaciones *Sparse-Sparse Matrix Multiplication* o SpMSpM (i.e., capas de DNNs cuyas matrices de activaciones y de pesos están comprimidas para eliminar los valores cero) [97, 98, 114, 122, 124], etc., tienen un inconveniente principal: *están diseñados para ejecutar capas de DNNs utilizando un único dataflow*. Por ejemplo, aceleradores como SIGMA [31] o ExTensor [50] ejecutan la operación SpMSpM utilizando un *dataflow Inner Product* (IP), requiriendo de ciertas unidades especiales para buscar los índices de ambas matrices que intersectan. Otros aceleradores como SpArch [124] o Outer-Space [97] están diseñados para ejecutar las capas SpMSpM utilizando un *dataflow Outer Product* (OP). De forma diferente, estos diseños requieren de estructuras tipo árbol para ejecutar una operación de ordenación (i.e., *merge*) entre distintos grupos de sumas parciales generados previamente. El inconveniente de este enfoque es que se generan una gran cantidad de sumas parciales que tienen que ser almacenadas en una memoria temporal. Finalmente, otros aceleradores como GAMMA [122] o MatRaptor [114] utilizan el *dataflow Gustavson's* (Gust), que de forma similar a OP generan sumas parciales y luego las ordenan, pero en este caso la generación y ordenación se hace fila a fila, reduciendo el número de sumas parciales temporales necesarias para ser almacenadas. Cada uno de estos enfoques tiene sus propias ventajas e inconvenientes y el rendimiento obtenido al ejecutar una determinada capa en un determinado *dataflow* depende especialmente del tamaño, y del patrón de sparsity que presente cada capa. Así, no existe un *dataflow* que se adapte bien para todos los tipos de capa y por tanto, todos los diseños mencionados anteriormente no son óptimos para ejecutar una DNN con cientos de capas. Para solucionar este problema, la tercera contribución de esta tesis es el diseño, presentación y evaluación de Flexagon, el primer acelerador específico para DNNs con soporte para la operación SpMSpM capaz de ejecutar los tres *dataflows* y por tanto capaz de adaptarse a cada capa a ejecutar. Flexagon está diseñado utilizando la estructura descrita anteriormente: una DN para distribuir los datos con soporte para los tres *dataflows*, una MN capaz de ejecutar operaciones de multiplicación y una nueva RN denominada

Merging-Reduction Network (MRN) que sirve para ejecutar ambas operaciones de acumulación y de ordenación, y por tanto, válida para adaptarse a los tres *dataflows* descritos anteriormente. Adicionalmente, Flexagon trae consigo un nuevo diseño de jerarquía de memoria que es capaz de adaptarse a los tres *dataflows*. Para ello, el primer nivel de la jerarquía se divide en 3 bloques de SRAM. Un primer bloque diseñado en forma de FIFO y que se utiliza para almacenar los datos que se mantienen más tiempo on-chip. Un segundo bloque de SRAM organizado como una caché tradicional pero utilizando tamaños de etiquetas reducidos y cuyo objetivo es capturar los patrones de acceso de memoria de los 3 *dataflows*, y un tercer bloque de memoria SRAM totalmente rediseñado en esta tesis denominado PSRAM. Esta estructura está especialmente diseñada para el almacenamiento temporal de las sumas parciales generadas en los *dataflows* OP y Gust, permitiendo un acceso eficiente en términos de energía, área y rendimiento. Junto con estos tres bloques de SRAM, hemos diseñado un único controlador de memoria unificado capaz de adaptarse y de generar las direcciones de memoria adecuadas para los 3 *dataflows*.

Para evaluar Flexagon hemos desarrollado un simulador a nivel de ciclo implementado en nuestro framework de simulación STONNE. En concreto, hemos utilizado una versión del simulador conectado al simulador SST [60] y que nos sirve para gestionar las peticiones de memoria que se generan de manera precisa. El simulador SST trae consigo un simulador de estructuras DRAM que nos permite conectar a nuestra jerarquía de memoria. Además, hemos implementado los aceleradores SIGMA (i.e., *dataflow* IP), SpArch (i.e., *dataflow* OP) y GAMMA (i.e., *dataflow* Gust) en el framework de simulación. Con el objetivo de comparar los resultados con Flexagon hemos configurado los 4 aceleradores con 64 multiplicadores, 16 elementos/ciclo de bandwidth, un tamaño para la estructura de memoria FIFO de 256 bytes, y un tamaño para la estructura caché de 1 MiB. Además, modelamos un tamaño de DRAM de 16 GiB y una latencia de 100 ns incorporando la tecnología HBM 2.0. Con estas cuatro configuraciones acelerador hemos realizado la ejecución de todas las capas para 8 modelos de DNNs extraídos de la suite de benchmarks MLPerf [107].

Los resultados obtenidos demuestran por primera vez que, tal y como predecíamos, no existe ningún *dataflow* que consiga el mejor rendimiento para todas las capas. Algunas DNNs como Alexnet, VGG-16, Resnets-50 y SSD-Resnets consiguen un 5,26$\times$ y 1,49$\times$ mejor rendimiento con una arquitectura tipo SpArch que con una arquitectura tipo SIGMA y GAMMA, respectivamente. Sin embargo, otras DNNs como Squeezenet, SSD-Mobilenets, DistilBert o MobileBert consiguen mejor rendimiento con GAMMA (mejora media de 3,28$\times$ y 2,41$\times$ con respecto a SIGMA y SpArch, respectivamente). En consecuencia a este primer hecho, Flexagon supera a todas las propuestas anteriores, ya que es capaz de adaptarse en tiempo de ejecución a cada DNN y a cada capa de DNN. Cuantitativamente, los resultados obtenidos para Flexagon reflejan que nuestra propuesta obtiene beneficios de rendimiento medios de 4,59×, 1,71× y 1,35× respecto a SIGMA, SpArch y GAMMA, respectivamente.

Además, en términos de área, Flexagon únicamente introduce un área adicional de 25 %, 3 % y 14 % con respecto a SIGMA, SpArch y GAMMA, respectivamente. De forma similar, el consumo con Flexagon incrementa un 25 % con respecto a SIGMA, un 9 % con respecto a SpArch y un 21 % con respecto a GAMMA. A pesar de este área y consumo extra introducidos, Flexagon todavía consigue beneficios en términos de rendimiento/área. Específicamente, los resultados revelan que Flexagon obtiene un 18 %, 67 % y 265 % mejor balance rendimiento/area con respecto a GAMMA, SpArch y SIGMA, respectivamente. Esto demuestra que Flexagon puede constituir un punto de inflexion en el futuro del diseño de los aceleradores específicos para DNNs con soporte para ejecución *sparse*.

Índice

A mi pareja y padres

Agradecimientos

El desarrollo de esta tesis doctoral ha sido sin lugar a dudas uno de los retos más complicados y a la vez de mayor aprendizaje de mi vida. Cuando comencé esta aventura académica hace cuatro años jamás habría imaginado la gran cantidad de personas que pasarían por mi vida, y lo mucho que tengo que agradecer a cada una de ellas.

En primer lugar, a ti Yolanda, mi pareja, mi amor. Realmente, es imposible para mí expresar en unas pocas palabras toda la gratitud y el amor que siento hacia ti. Gracias por formar equipo conmigo durante todo este tiempo. Gracias por acompañarme estos nueve años de carrera, máster y doctorado. Por sostenerme cuando tropezaba, levantarme cuando caía, e incluso tirar de mí cuando no podía levantarme. Gracias por escucharme de forma genuina, incluso cuando no entendías nada de lo que estaba diciendo. Gracias por apoyarme en cada curso, en cada clase, en cada examen, en cada trabajo y en cada paper. Eres un ejemplo de esfuerzo, apoyo, compromiso, fortaleza, humildad, honestidad, sinceridad y bondad. Has sido una inspiración para mí, y me siento muy afortunado de haber crecido como profesional y como persona a tu lado. Entre los miles de millones de personas que existen en el mundo, yo no he podido elegir a nadie mejor para ir de mi mano y espero que sepas que cada palabra de esta tesis doctoral también te pertenece, pues sin tu ayuda y apoyo jamás habría sido escrita.

También me gustaría agradecer enormemente a mis padres, por vuestro esfuerzo en cuidar siempre de mí, por la educación y valores que me habeis proporcionado y que tanto me han servido para superar cada uno de los obstáculos en el camino. Gracias por vuestras enseñanzas y lecciones, pues gracias a ellas he llegado al final de esta etapa y me siento capacitado para cualquier reto al que tenga que enfrentarme en la vida. En concreto, a ti mamá, por tu amor incondicional hacia mí durante todas las etapas de mi vida. Por tus charlas por la noche y por el día que tanto me enseñaban y me ayudaban a creer en mí y a ser mejor persona. También a ti papá, por jugar conmigo de pequeño y por sentarte a mi lado a enseñarme matemáticas, ciencias, geografía y mucho más. Fuiste la llama que prendió mi curiosidad y que me inspiró hacia el desarrollo de esta tesis doctoral. A mis abuelos Santos y Kiki, y a mis abuelas Gregoria y Bernarda. Gracias a los que estáis y a los que ya no. Por vuestro amor incondicional, aprendizajes y apoyo durante las distintas etapas de mi vida. Os llevo y llevaré siempre en mi corazón. A todos mis tíos y tías Juanje, Glori, Conchi y Ángel. Gracias por haberme hecho sentir siempre tan querido, por apoyarme en mis decisiones, por creer siempre en mí, y por motivarme hacia esta meta, como hacía tantas otras. A mis dos únicos primos Juan Jesús y Alejandra que para mí siempre han sido y serán como hermanos. A ti Juan Jesús, por acompañarme durante toda mi infancia y cuyo ejemplo siempre me ha inspirado para ser mejor persona. A ti Alejandra, porque a pesar de nuestra diferencia de edad, siempre he podido sentir tu cariño y apoyo incondicional. A mis suegros Paqui y Eduardo, junto con mi cuñado Dani, además de al resto de mi familia política, Antonia y José. Gracias a todos vosotros por cuidar siempre de mí. Por incluirme desde el primer día en vuestra familia como uno más y por tratarme como tal. Gracias por creer en mí todos estos años, por celebrar mis alegrías cómo vuestras y por hacerme sentir que podía lograrlo. Sois también una parte fundamental en mi vida, y me siento muy orgulloso de ello.

A Díaz y Carmen. A ti Díaz, por creer en mí, por permanecer siempre a mi lado desde los tres años de edad y demostrarme que la verdadera amistad es inmutable al tiempo y al espacio. Por ser un ejemplo de fortaleza frente a la vida. También a ti Carmen, por nuestras conversaciones en el autobús durante mis primeros años en la universidad, por empujarme a confiar en mí mismo, y por ayudarme siempre en todo lo que he necesitado durante estos años.

A todos mis "thunderfriends": Luis, Alberto, Javi, Juan Antonio, Puri, Manu, Laura, Lucía, Juan Antonio, Noemi, Morad, y Dani. Gracias por todas las conversaciones que solo nosotros entendemos. Gracias por todos los momentos y fiestas inolvidables que tanto me hacían disfrutar y desconectar. Y gracias por apoyarme durante todo este camino. Aunque últimamente ya no nos podemos ver muy frecuentemente, os llevo siempre conmigo.

A mis directores de tesis, Manolo y José Luis. Gracias a ambos por confiar en mí para la realización de este proyecto. Nunca es fácil comenzar algo desde cero, y sin embargo, vosotros no dudasteis de mí ni un instante. Habéis sido la luz que ha guiado mi camino. Incluso en los momentos más oscuros, habéis sabido encender la llama de la motivación y me habéis sacado de allí de la mano, trabajando siempre a mi lado, de forma muchas veces totalmente altruista.

Además, gracias a ti Manolo, por descubrirme en tercero de carrera. Porque yo no lo sabía, pero aquella primera reunión en tu despacho cambiaría mi trayectoria profesional para siempre. Gracias por abrirme al mundo de la investigación y hacerme conocer las oportunidades que puede ofrecer. También gracias a ti José Luis, por contagiarme tu motivación por la docencia y la investigación. Por hacerme sentir que podía lograrlo, y por todo el esfuerzo, humildad y honestidad que has puesto en cada uno de nuestros trabajos y que siempre ayudaban a mejorarlos.

Special thanks to Professor Tushar Krishna. Thank you for being my third (although not official) thesis director. One day you looked up your mailbox and came across an email from three random researchers that had just started a new project. Other person could have simply ignored the email. However, rather than that, you set up a meeting and started a close collaboration with us. You included us within your group and even welcomed me with open arms in your Synergy lab. You allowed me to live in the USA, which was one of the most enrichment experiences of my whole life. You honor the name of your group by being an example of modesty, inclusion, and synergy. Thank you for everything, Tushar. Moreover, to Michael Pellauer. Thank you for joining us this last year and for adding your enormous experience in this research field. You are one of the best researchers I have ever worked with, and without you, the development of Flexagon, the last contribution of this thesis, would have never been come alive. To all my colleagues at the University of Murcia: Eduardo, David, Ashkan, Sawan, Víctor, Sebastian, Pablo, Alejandro, Vahid and Ayyoub. You are the best I take away from this thesis. Thank you for all our lunches, dinners, and parties. For the endless technical discussions. For listening to me when I needed someone and, in essence, thanks for all your support during this time. To me, you are not just colleagues, you are family, and you will ever be. To all my colleagues at the Georgia Institute of Technology: Eric, Saeed, Abhimanyu, Canlin, Jinsun, Jianming, Geonhwa, and Anand. Thanks for your support and endless help during my life in Atlanta. Special thanks here to Raveesh, for your help and efforts during this thesis, our endless and helpful technical discussions, and for your genuine support during the hardest moments of my stay at Georgia Tech.

También, me gustaría mencionar a aquellas personas que pasaron por mi vida durante el grado y el máster, pues me ayudasteis a superar cada uno de los duros desafíos: Marcelo, Alejandro, Adrián, Victoriano, Puri, Laura, José María, Jesús, Serena, Guillermo y muchos más. Aquí, me gustaría mencionar especialmente a mi compañero de prácticas, José María (a.k.a., Queque). Gracias por la motivación que me diste durante los primeros años. Sin ti, siento que jamás me habría enamorado tanto de la informática y probablemente hoy no

estaría aquí escribiendo estás líneas. Te estaré siempre agradecido por todo tu apoyo.

A mi terapeuta Diego, por acompañarme durante este último año. Gracias por ayudarme a conocerme un poco más y empujarme a realizar los cambios que necesitaba en este momento de mi vida. Eres un gran ejemplo de profesional y me siento afortunado de haberte encontrado.

Finalmente, agradecer también a la Fundación Séneca (Agencia de Ciencia y Tecnología de la Región de Murcia) por su apoyo económico mediante una Beca-Contrato Predoctoral de Formación del Personal Investigador (FPI).

A cada uno de vosotros, muchísimas gracias por hacer todo esto posible.

To all of you, thank you very much for making all this possible.

Universidad de Murcia

Facultad de Informática

Departamento de Ingeniería y
Tecnología de Computadores

Hardware Techniques for the Design of Efficient Inference Accelerators of Deep Neural Networks

PhD Thesis

By
Francisco Muñoz Martínez

Advised by
Manuel Eugenio Acacio Sánchez
José Luis Abellán Miguel

Murcia, Septiembre 2022

Abstract

The design of specialized architectures for accelerating the inference procedure of Deep Neural Networks (DNNs) is a booming area of research nowadays. While first-generation rigid accelerator proposals used simple fixed dataflows tailored for dense DNNs, more recent architectures have argued for flexibility to efficiently support a wide variety of layer types, dimensions, and sparsity. As the complexity of these accelerators grows, the analytical models currently being used for design-space exploration are unable to capture execution-time subtleties, leading to inexact results in many cases as we demonstrate. This opens up a need for cycle-level simulation tools to allow for fast and accurate design-space exploration of DNN accelerators, and rapid quantification of the efficacy of architectural enhancements during the early stages of a design. To this end, the first contribution of this thesis is STONNE (*Simulation TOol of Neural Network Engines*), a cycle-level microarchitectural simulation framework that can plug into any high-level DNN framework as an accelerator device and perform full-model evaluation (i.e. we are able to simulate real, complete, unmodified DNN models) of state-of-the-art rigid and flexible DNN accelerators, both with and without sparsity support. As a proof of concept, we use STONNE in three use cases: *i)* a direct comparison of three dominant inference accelerators using real DNN models; *ii)* back-end extensions and *iii)* front-end extensions of the simulator to showcase the capability of STONNE to rapidly and precisely evaluate data-dependent optimizations. Once, we have a validated simulator to perform our evaluations, the second contribution of this thesis focuses the attention on the flexible architectures for DNNs. DNN accelerators use three separate NoCs within the accelerator, namely distribution, multiplier and reduction networks (or DN, MN and RN, respectively) between the global buffer(s) and compute units (multipliers/adders). These NoCs enable data delivery, and more importantly, on-chip reuse of operands and outputs to minimize the expensive off-chip memory accesses. Among them, the RN, used to generate and reduce the partial sums produced during DNN processing, is what implies the largest fraction of chip area (25% of the total chip area in some cases) and power dissipation (38% of the total chip power budget), thus representing a first-order driver of the energy efficiency of the accelerator.

RNs can be orchestrated to exploit a Temporal, Spatial or Spatio-Temporal reduction dataflow. Among these, the latter is the one that has shown superior performance. However, as we demonstrate, a state-of-the-art implementation of the Spatio-Temporal reduction dataflow, based on the addition of Accumulators (Ac) to the RN (i.e. RN+Ac strategy), can result into significant area and energy expenses. To cope with this important issue, we propose STIFT (that stands for *Spatio-Temporal Integrated Folding Tree*) that implements the Spatio-Temporal reduction dataflow entirely on the RN hardware substrate (i.e. without the need of the extra accumulators). STIFT results into significant area and power savings regarding the more complex RN+Ac strategy, at the same time its performance advantage is preserved.

The third contribution of this thesis increases the flexibility current sparse accelerators by adding support for more dataflows.

Existing Sparse-Sparse Matrix Multiplication (SpMSpM) accelerators are tailored to a particular SpMSpM dataflow (i.e., Inner Product, Outer Product or Gustavson's), that determines their overall efficiency. We demonstrate that this static decision inherently results in a suboptimal dynamic solution. This is because different SpMSpM kernels show varying features (i.e., dimensions, sparsity pattern, sparsity degree), which makes each dataflow better suited to different data sets.

Motivated by this observation, we propose Flexagon, the first SpMSpM reconfigurable accelerator that is capable of performing SpMSpM computation by using the particular dataflow that best matches each case. Flexagon accelerator is based on a novel Merger-Reduction Network (MRN) that unifies the concept of reducing and merging in the same substrate, increasing efficiency. Additionally, Flexagon also includes a 3-tier memory hierarchy, specifically tailored to the different access characteristics of the input and output compressed matrices. Using detailed cycle-level simulation of contemporary DNN models from a variety of application domains, we show that Flexagon achieves average performance benefits of 4.59×, 1.71×, and 1.35× with respect to the state-of-the-art SIGMA-like, SpArch-like and GAMMA-like accelerators (265% , 67% and 18%, respectively, in terms of average performance/area efficiency).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computing power has experienced tremendous performance improvements in the last four decades mainly due to significant advances in technology. In 1974 Robert Dennard observed that power density was constant for a given area of silicon even if the number of transistors was increased because of the smaller dimensions of each transistor. This way, transistors could go faster but use less power. In addition, Moore's Law predicted by Gordon Moore in 1965 indicated that the number of transistors per chip would double every two years. Both observations combined made possible that the performance per watt provided by computers doubled every 18 months. Unfortunately, in 2004 Dennard scaling ended because current and voltage were not able to keep dropping and still maintain the dependability of integrated circuits. This slowed down the computing power growth to about once every 2.6 years and architects started to build multi-core systems to overcome the technology limitations. Nowadays, the scenario is way more challenging. Moore's Law is coming to an end and computer architects are moving from high-performance, general-purpose computers, to domain-specific hardware as the only path left to supply the high computing requirements demanded by the new era of computing, which is characterized by data processing and machine learning algorithms. The envisioned scenario is so exciting and challenging that has been defined as the new golden age for computer architecture [51].

1.1 Towards General-Purpose Processors

The Cambridge Dictionary defines hardware as *the physical and electronic parts of a computer, rather than the instructions it follows*. Given this, non-expert people tend to think of hardware as an independent part of the system that can be touched, while software is a completely separate part that runs on the hardware. Even though this may be considered as true, as explained below, the hardware-software co-design is much more closer today than what non-experts tend to believe.

Until the beginning of the 21st century, the number of computer applications demanded by end users was exponentially growing ranging from office automation tools, client browsers, server services, or video games to high performance scientific applications. There were some exceptions (e.g., embedded systems design, the design of some specific supercomputers to accelerate HPC workloads, or specialized GPUs to accelerate graphic workloads), but, in general, given the wide diversity of applications, computer architects used to design general-purpose processors with ever-complex microarchitectural support in a best-effort attempt to run all of them, i.e., developing specific processors specifically tailored to accelerate some particular applications was not still profitable enough targeting the general end user.

To develop general-purpose processors, during the semiconductor revolution in the 1980s, computer architects took advantage of Moore's Law [38] to transform the increased number of transistors per chip into higher performance. To do so, practitioners focused the attention on increasing the amount of work performed in each cycle allowing more capability to execute multiple instructions from the same program simultaneously, i.e., the extraction of Instruction Level Parallelism or ILP. Exploiting techniques such as out-of-order execution [45], deeper instruction pipelines [39], highly speculative execution [95], or larger on-chip memory hierarchies [90], among others, became the main source of improvement. This constituted the best way to design processors able to efficiently execute a wide variety of high performance computing applications.

Despite these tremendous advances, since year 2000 the trend towards designing very aggresive and complex superscalars processors came to a stop. RAW (read-after-write) data hazards between consecutive instructions, control hazards due to branch instructions, as well as structural hazards which serialize the execution of instructions that try to use the same hardware resource at the same time, made impossible to increase performance without a considerable effort [59]. As a result, every new performance improvement has been empirically close to the square root of the number of required transistors [59]. Besides, increasing the clock frequency in order to obtain a faster circuitry involves important heat problems and high energy consumption. Moreover, the efforts to maintain manageable parameters for the thermal-power issues such as increased threshold voltage to control leakage, or limited supply-voltage scaling, decrease the performance benefits of transistor scaling [58]. All above problems were identified as the *Power*, *Memory*, and *ILP Walls* [62] and led to the second generation of processors in both industry and academia.

The strategy followed by this second generation may be simply summarized as *divide and conquer*. The idea was to take advantage of the Moore's law to introduce several simpler processors into the same chip. Systems of this type are commonly referred to chip multiprocessors (CMPs) and their main goal is to accelerate applications by exploiting Thread-Level Parallelism. CMPs have brought important advantages over very complex, deep-pipelined general-purpose processors. In particular, they provide higher aggregate computational power, multiple clock domains and better power efficiency. They are even able to provide simpler designs which leads to significant reductions in the cost of design and verification of the chips.

Due to its tremendous benefits, the current trend towards gaining better performance has been to increase more and more the number of processor cores per chip. As an example, the 12th-generation Intel Core i7 (i.e., i7-12800HX) and the Intel Core i9 (i.e., i9-12950HX) features 16 cores and can execute up to 24 threads in parallel when the hyper-threading technology is enabled [57]. Even lower-power processors like the 12th-generation Intel Core i3 (i.e., i3-122OP) implements 10 physical cores and can run up to 16 threads.

1.2 Towards Specialized Compute Processors

Increasing more and more the number of processor cores seemed to be an endless source of improvement during the first decade of the 2000's. However, from 2010, this trend started to face some challenges:

- Extracting TLP is way more challenging than extracting ILP. Whereas the compiler and hardware can implicitly extract ILP without the programmer's attention, the extraction of TLP is explicit for the programmer. The creation of parallel programming APIs such as OpenMP [94] and MPI [36] has to some extent simplified this task, but sometimes it is still required to restructure the program, becoming a major burden for programmers.

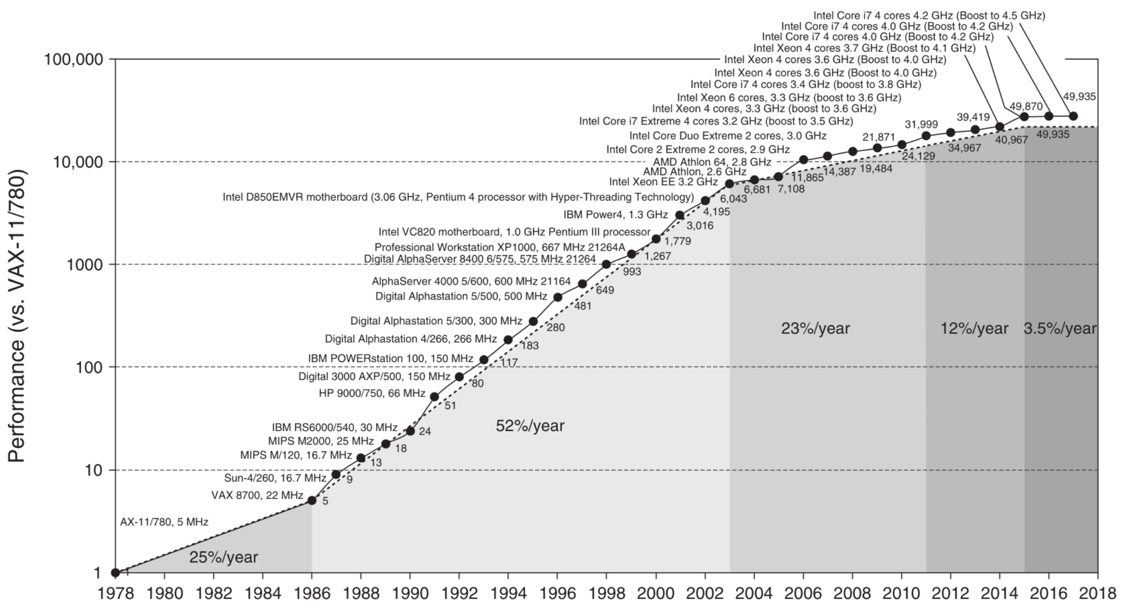


Figure 1.1: Growth in processor performance over the last 40 years. Figure extracted from [59].

- Amdahl's Law prescribes practical limits to the number of useful cores per chip. For example, if 40% of the program is serial and cannot be parallelized at all, then the maximum of obtained parallelism is 60%, regardless of the number of cores that the chip implements.

- Finally, and what is even more important, the end of Moore's Law and Dennard's scaling. As described above, TLP is exploited by putting more and more cores in the same area budget. This has been achieved so far by doubling the number of transistors in the same area with almost constant power density, and therefore, the performance, every 2 years, taking advantage of smaller and smaller technology nodes. However, as described before, this is coming to an end, which can be observed in Figure 1.1, where it is shown the growth in processor performance over roughly the last 40 years. As we can see, from 2011 and 2015, the annual performance improvement was less than 12% which means doubling performance every 8 years. Since 2015, performance improvement has been just 3.5% per year, or doubling every 20 years.

As observed in Figure 1.1, the era of the CMPs has come to an end, and it is no longer possible to produce better and better general-purpose processors.

The only path left towards supplying more an more compute power is the architectural design strategy coined as *specialization*. This strategy has long been extremely successful for efficiently developing specific computing platforms aimed at efficiently running specific applications workloads such as scientific or multimedia workloads. To name a few examples of these specialized designs, already, in the early days of computing, around the last 60's, the well-known vector supercomputers came into light such as STAR-100 [112], TI-ASC [26] or Illiac IV [112], which were machines equipped with a large number of functional units specially built to handle large scientific and engineering calculations. From 1996, specialization is even present in general-purpose processors with the introduction of the MMX TM Technology [99] extension to the Intel Architecture, which was later replaced by SSE [105] and recently by AVX [10]. These extensions were specialized instructions within Intel Architectures specifically designed to accelerate multimedia and communications applications. Traditionally, Graphics Processing Units (i.e., GPUs) were designed as an example of specialization to accelerate graphics pipelines such as OpenGL-based or Vulkan-based video games. Recently, and mostly since the foundation of CUDA [92] these architectures have evolved towards the well-known General-Purpose GPUs (i.e., GPGPUs) aiming to accelerate other domains such as scientific or engineering applications.

In the last decade, we are witnessing a new era of computing built upon Big Data and Artificial Intelligence algorithms, being the latter dominated by Machine Learning techniques such as Deep Learning (DL) based on Deep Neural Networks (DNNs) as its major flagship. Given the myriad of application domains that currently benefit from DNN techniques, and the hundreds of thousands of end users interested in these applications alike, it is the first time when massive development of specialized computing platforms are promptly starting to pay off so as to being clearly profitable to industry. That is why new Client-Server infrastructures based on Edge-, Fog- and Cloud-Computing [61] are continuously evolving for the best user experience when running these new applications, most of them executed over battery-operated and heavily-constrained smartphones.

To provide the highest energy-efficiency and performance to the execution of these DNN-based workloads, an in-depth understanding of their underlying algorithms is strictly necessary for a successful *hardware-software co-design* of the specialized computing platforms. This way, for instance, it can be possible to extract the common compute and memory access patterns covered by a range of deep learning algorithms and design specific architectures for them. This

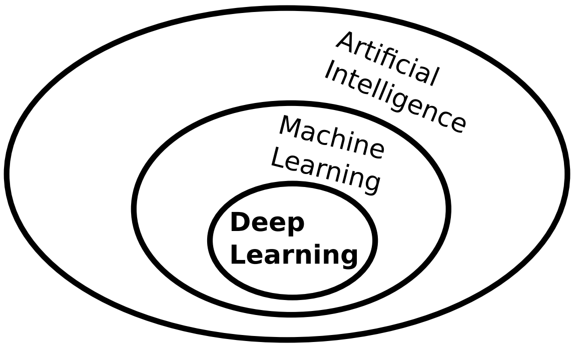


Figure 1.2: AI classification.

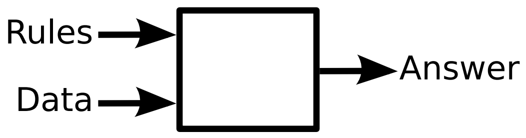


Figure 1.3: Rule-based system.

removes ineffectual hardware and brings to the chip the essential components that meet both performance and power consumption goals. Therefore, before describing how these specialized compute architectures are constructed, it is needed to review how the software side works.

1.3 Machine Learning

Artificial Intelligence (AI) (see Figure 1.2) is a broad concept that encompasses algorithms and ideas invented to solve very sophisticated problems that go beyond of what computers were meant when they were born.

Prior to machine learning, AI systems used to solve these complex tasks by making use of user-defined rules. In this programming paradigm, described in Figure 1.3, programmers fed a system with data and rules that specified how to operate with such data. The goal of this AI system is to report the right answer for a particular question regarding the input data following pre-defined rules.

Machine learning is a field within AI that revolutionized overnight the way computers learn. With this strategy, the rules are no longer fed by the user and instead they are generated (i.e., learnt) by the machine learning system on its

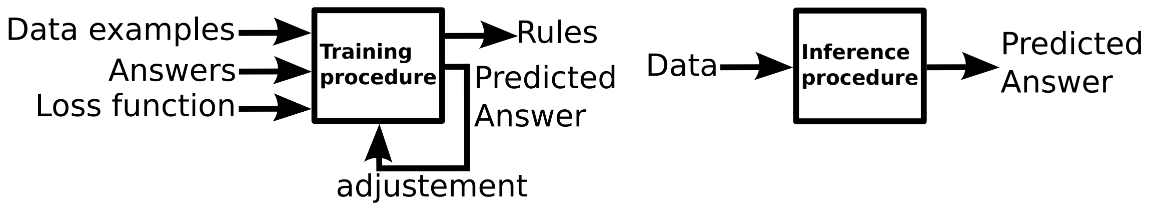


Figure 1.4: ML system.

own. To do so, the system operates in two clearly distinguished procedures which are shown in Figure 1.4 and described as follows:

- **Training**: During this phase, the ML system learns the rules to operate within the next phase. With this aim, the system is fed with input data examples and input correct answers –for simplicity, we assume a supervised learning process. For every example in the dataset, the system predicts an answer based on its internal rules. At the beginning of the training process, these rules will be incorrect and will probably lead to wrong answers. To solve this issue, the system calculates the distance (i.e., *loss*) between the correct and the predicted answer using a *loss function* and uses this metric to learn how to adjust its rules in order to be able to predict the correct answer next time.

- **Inference**: After enough number of examples, the system learns how to predict the correct answer for every single example and if the dataset is large enough, it will be able to generalize to new examples. At this point, the program is ready to operate in production to perform the task for which it was trained for.

1.3.1 Deep Learning

Deep Learning (DL) is a specific subfield of machine learning that has become extremely popular in the last decade due to the high accuracy that is able to offer, overcoming humans in many challenging tasks. The term *deep* comes from the ability of DL to hierarchically divide the learning representation in a high number of successive layers stacked on top of each other. Thus, the *depth* of a DL model indicates that big number of layers that constitute the model.

In DL, these layered representations are loosely brain-inspired algorithms usually called Deep Neural Networks (i.e., DNNs). The term *Neural Network*

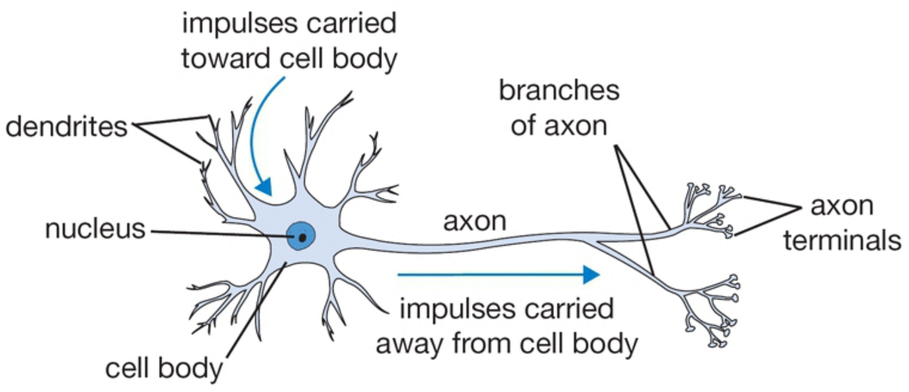

impulses carried toward cell body
dendrites
branches of axon
nucleus
axon
axon terminals
cell body
impulses carried away from cell body

Figure 1.5: Biological neuron. Figure extracted from [21].

comes from the biological neural networks as these models were an attempt to imitate the most powerful machine-learning system ever created: the human brain. In the biological model, a network of specialized cells called neurons is in charge of the learning process (see Figure 1.5). Each of these units comprises three main parts: the body which contains the nucleus, the dendrites which act as input channels to receive information from other neurons, and the axon, responsible for output signals to other neurons. A connection between two neurons is called a synapse and this plays the major role in the learning process of the network. Although the exact behaviour of a neuron is still unknown, it is believed that the natural working process consists of receiving a set of electrical signals called input activations. Then, these signals are manipulated through the synapses and once the signals are within the nucleus, the neuron performs certain biological computation and triggers an output activation signal if the voltage of the manipulated input signals are above a certain threshold. The way the synapses manipulate these input signals is what produces the *learning* of the network.

Inspired by this natural process, in 1958 Frank Rosenblatt developed the perceptron (see Figure 1.6), a mathematical concept that reproduces the behaviour of a neuron [109]. To do so, a perceptron is composed of two terms: a weighted sum of inputs, and a non-linear activation function. The relation between the biological and the artificial computation is obvious: each input corresponds to an activation input in the biological model and the weights correspond with the synapses. As it can be appreciated, the model can be seen as a generalized linear

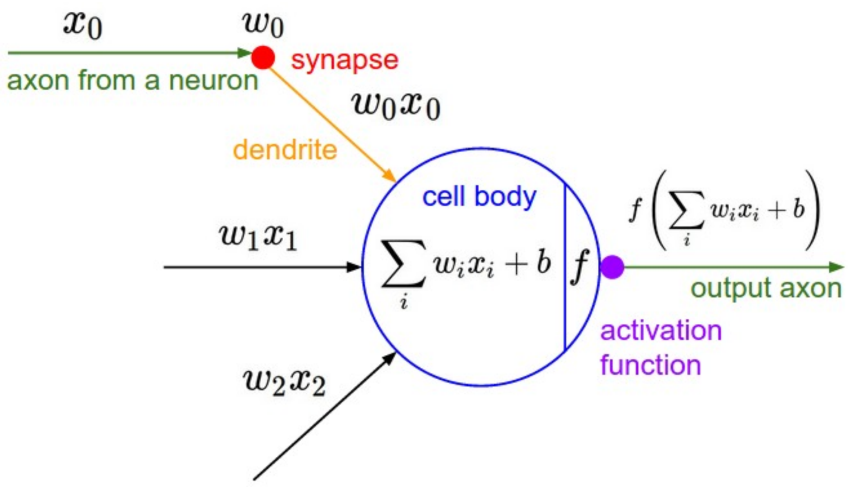


Figure 1.6: Overview of the Perceptron. Figure extracted from [117].

classifier, being:

$$F(x, w) = \sum_i x_i * w_i$$

a linear combination of inputs and f a mapping function that determines the category in which each combination of inputs falls. Based on the value of the weights, the perceptron can learn how to classify inputs for different types of tasks. This is the way the model *learns* and is the principle of the more sophisticated neural networks: the ability to model complex behaviour is not due to sophisticated neurons, but to aggregate behaviour of many simple parts.

Since this invention was developed, a self-reinforcing cycle (see Figure 1.7) formed by three key factors have fueled the field of deep learning. As larger datasets are available due to the massive usage of Internet and applications, new techniques and algorithms are invented. These algorithms usually involve larger and deeper models with larger number of parameters (i.e., weights) and computation which exacerbates the need for more sophisticated hardware processing devices to keep up with the time, energy and memory requirements constrained by the applications that utilize these models.

1.3.2 Deep Neural Networks

As previously described, DNNs take the aforementioned biological inspiration to build complex models based on weighted sums of input values.

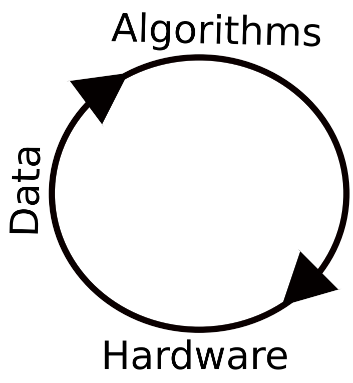


Figure 1.7: Deep learning virtuous cycle.

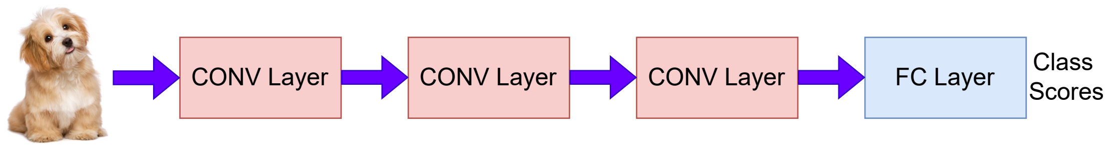


Figure 1.8: High level overview of a DNN.

Figure 1.8 depicts a high level overview of a DNN. As we can see, a model is broken down into consecutive layers. The aim of each layer is to extract features from the input layer and propagate those features to the output layer. According to the location of each layer in the sequence, we can differentiate between three types of layer:

1. Input layer: Represents the input of the DNN model and it is an array of data whose meaning depends on the domain the network is targeted at. For instance, for the image classification domain, this array of data may represent the pixels of an image, or for the audio recognition domain the array of data may represent the encoded audio waves.

2. Hidden layers: These layers connect either the input layer or previous hidden layers, and transform the input into relevant features that are used by subsequent layers.

3. Output layer: Represents the output vector and contains the result of the DNN model. Again, the meaning of this output vector depends on the domain. For the image classification domain, this vector may be a

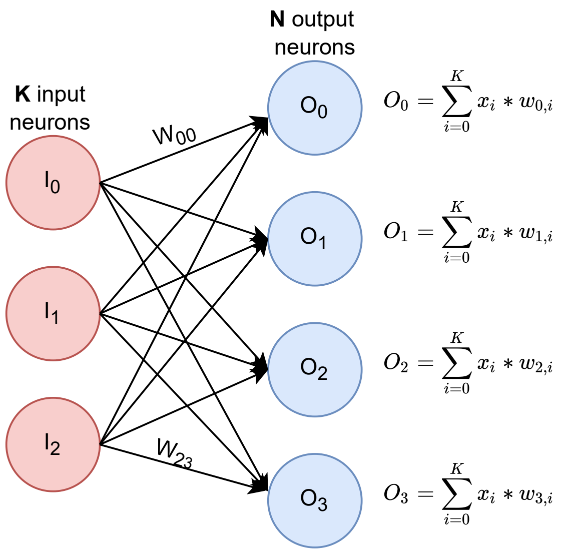

N output neurons
K input neurons
I_0
I_1
I_2
O_0
O_1
O_2
O_3
W_{00}
W_{23}
$O_0 = \sum_{i=0}^{K} x_i * w_{0,i}$
$O_1 = \sum_{i=0}^{K} x_i * w_{1,i}$
$O_2 = \sum_{i=0}^{K} x_i * w_{2,i}$
$O_3 = \sum_{i=0}^{K} x_i * w_{3,i}$

Figure 1.9: Example of a Fully-Connected (FC) layer.

probability vector where each position indicates the probability that the image belongs to a certain class.

Besides, according to the computation involved, we can also distinguish between different types of DNN and layers. The most common DNN layers that can be found in nowadays' DNN models are briefly reviewed next.

1.3.2.1 Fully-Connected Layer

A Fully-Connected (FC) layer extends the concept of the perceptron explained in Section 1.3.1, organizing many parallel neurons into a layer. Figure 1.9 depicts an example of a FC layer composed of `K` input neurons and `N` output neurons. Each neuron is represented as a node and each edge represents the weight that connects two nodes between the inputs and outputs. A FC layer connects every node in a layer to all the nodes in the previous layer. Given this, the computation performed within a particular node O_j is the weighted sum of the nodes from the previous layer and can be generalized as:

$$O_j = \sum_{i=0}^{K} x_i * w_{j,i}$$

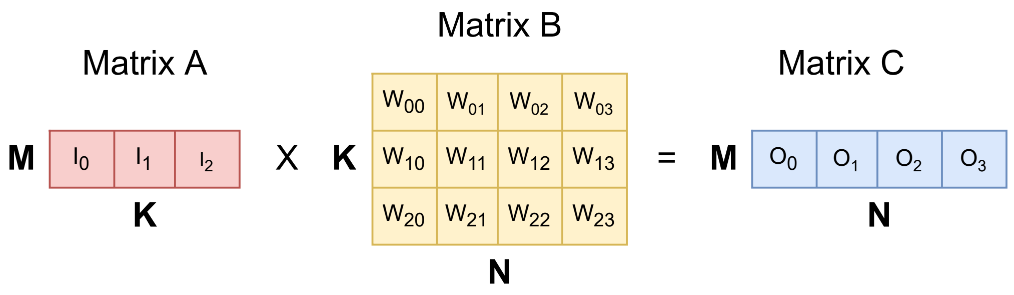


Figure 1.10: Matrix Multiplication operation mapping the FC layer presented in Figure 1.9.

CPUs, GPUs and accelerators execute this layer by mapping this abstraction into a matrix multiplication (GEMM) operation $A_{MK} \times B_{KN} = C_{MN}$ where matrix `A` corresponds with the inputs, matrix `B` maps the weights, and matrix `C` represents the outputs. Figure 1.10 shows the computation involved in Figure 1.9 mapped into a GEMM operation. The M dimension is often utilized to map the number of input samples that are executed (i.e., batch size).

1.3.2.2 Convolutional Layer

One of the key features of the image and video processing domain is the massive number of pixels that the inputs contain. Assigning each of these pixels to an input neuron in a FC layer (see 1.3.2.1) would bring about an explosion of the number of required weights in a DNN, hence making the execution unfeasible.

Besides, this assignment is against the nature of the image processing domain. Basically, each pixel is treated as a completely independent input signal from every other. However, neighboring pixels typically contain signals that relate to the same object in the image. Hence, a set of nearby pixels becomes a region of similar information that can be captured and processed together, what if properly exploited, can lead to a reduced amount of computation and number of required parameters by a DNN model.

This is exactly the aim of a Convolutional Neural Network (CNN), which involves the computation of consecutive convolutional layers, and were first introduced by Yann LeCun's group at Bell labs and later NYU [71].

As described by Sze *et al.* [117], each convolution layer in a CNN involves computation of high dimensional convolutions (2D or 3D), where an input fmap (`ifmap`) consisting of a set of channels is convoluted with a different trained filter composed of weights, and the results of the convolution at each point are accumulated across all the channels. The result of this computation is one channel of the output feature map (`ofmap`). The computation is described as follows:

$$O_{n,k,x,y} = \sum_{c=0}^{C} \sum_{r=0}^{R} \sum_{s=0}^{S} I_{n,c,U\times x+r,U\times y+s} \times W_{k,c,r,s}$$

Here, O, I, and W are the matrices for the ofmaps, ifmaps and filters, respectively. Furthermore, N stands for the number of `ifmaps` and `ofmaps` (i.e., batch size), C is the number of ifmap channels and filter channels. X and Y are the ifmap plane height and width, respectively. R and S are the filter plane height and width, respectively. X′ and Y′ are the ofmap plane height and width, respectively. U is the stride of the window of input values given to the convolution operation which has typically a value of 1.

To help the reader understand the computation of a convolution layer, we depict it in Figure 1.11 assuming a 3D convolution. First, it receives a 3D array of values (activations), i.e. the ifmap, of size $C \times X \times Y$, where, as shown in the figure, C is the number of channels and $X \times Y$ refers to the size of each channel. Taking this input, the layer performs a convolution applying a $C \times R \times S$ filter to obtain as output a single ofmap channel. Notice that the number of channels in the convolution operation (C) is the same for both filter and ifmap sets as each filter channel is assigned to a different ifmap channel. This convolution is often implemented as a sliding window operation which means that the filter is shifted across the ifmap from left to right and from up to down, calculating a different output activation for every shift. Finally, in order to calculate the K ofmap channels, K filters are needed and applied to the very same fmap. Similar to the FC layer described in Section 1.3.2.1, the convolution layer is also typically mapped into a matrix multiplication operation by means of a well-known operation called *img2col* [117]. Typically, the result of every convolution layer is given as input to an activation layer such as ReLu or Sigmoid [93], which adds non-linearity, and hence, the ineffectual output activations can be filtered out.

Along with convolution layers, FC (Fully-Connected) layers are also part of a typical CNN. This layer generates the class probabilities from the class scores (a.k.a., logits). In the context of image classification problems, the highest score represents the most likely class of an object (e.g., a dog) in the input image given to the CNN.

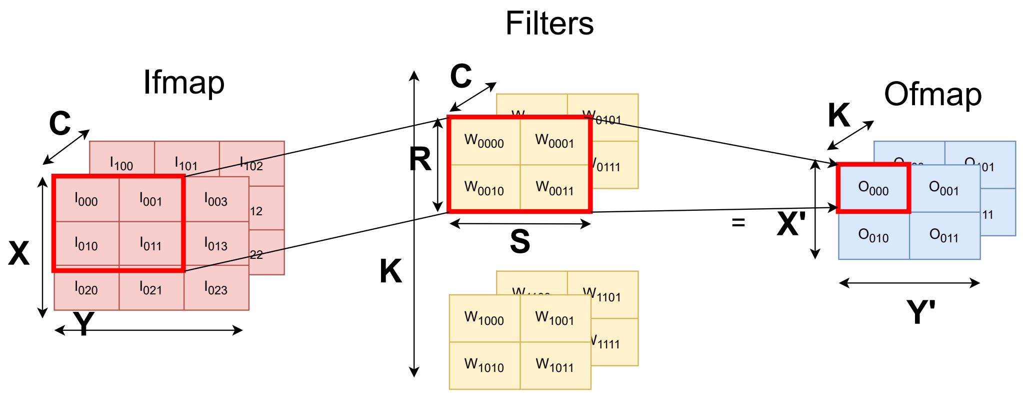


Figure 1.11: Example of a convolution layer.

1.3.2.3 Other Layers

Since convolution and fully-connected layers typically entail the 95% of the computation in most of the DNNs [117], this thesis mainly focuses on the acceleration of these layers[1]. In spite of this, other layers are also relevant for the community and deserve the attention. Next we describe the principal ones:

- Pooling layer: A pooling layer [43] is used typically in CNNs to reduce the spatial dimensions of the ofmaps through combining a set of values into a smaller number of values by using functions such as max-pooling or average-pooling. For example a 2×2 set of values can be represented by an unique average value.

- LSTM layers: A Long Short-Term Memory (LSTM) layer [52] is commonly used in Recurrent Neural Networks (RNNs). This popular variant of a DNN have internal memory to allow long-term dependencies that affect the output. The RNNs are used for sequence-based tasks that require time dependencies among inputs such as speech recognition or natural language processing. During this thesis we will not include any RNN in our evaluations because these networks have been massively replaced by the attention layers, explained as follows.

- Attention layers: Attention layers [118] are the basis of transformers which are emerging algorithms aiming to replace the aforementioned RNNs. This

[1]In Chapters 2 and 4 we will also use some attention layers as part of our evaluation, since they are gaining importance in some domains

layer consists of a sequence of operations which can be translated into a matrix-multiplication operation.

1.3.2.4 Computing DNN Layers

The large majority of the kernels described in Sections 1.3.2.1, 1.3.2.2 and 1.3.2.3 can be typically expressed as a massive number of multiply-and-accumulate (MAC) operations. To compute these MACs on general-purpose CPUs, these operations are normally mapped into a matrix-multiplication operation and executed using some of the popular high-performance libraries such as the Intel Math Kernel Library (MKL) [25]. However, as described in Section 1.1, this is no longer feasible and we need specialized architectures to run these kernels. Next, we present the state-of-the-art specific architectures for efficient DNN processing and the challenges we will address in this thesis.

1.4 Specific Architectures for DNNs

As we have explained previously, the deployment of a DNN model comprises two phases called *training* and *inference*. During the training phase, the DNN model adjusts the values of its set of weights. Subsequently, during inference, the trained DNN model is used to solve the problem it was designed for (e.g., image classification). Currently, training is mostly carried out using clusters of several GPUs [20], although some proposals for customized training platforms have also been developed by both industry (e.g. Google's Cloud TPU [61] and Microsoft's Project Brainwave [37]) and academia (e.g., [31, 34]). In contrast, the fact that the DNN inference phase must be primarily done *in-situ* has paved the way for the development of a plethora of accelerator architectures so as to maximize performance per watt while meeting their latency and energy-efficiency demands [22, 31, 33, 61, 70, 76, 78, 98]. All these recent architectures are known as spatial architectures and differently to traditional temporal architectures (i.e., general-purpose architectures such as CPUs or GPUs), they remove ineffectual hardware that is not needed to compute DNNs, leaving space for what is really needed, such as simpler computing elements. For example, traditional caches are typically used in general-purpose processors to capture irregular memory access patterns generated by general-purpose applications. Conversely, in the case of DNN processing, since the data movement and memory access patterns are well-known a priori (see Sections 1.3.2.2, 1.3.2.1 and 1.3.2.3), a simpler SRAM scratchpad is sufficient to keep track of the location of the data.

The key idea behind the design of all these recent specific architectures has been the capture of the different patterns of data reuse in what is known as a *dataflow* [23,68]. According to the dataflow, optimizations implemented and its flexibility, we can distinguish between three different types of architectures for DNN inference accelerators: rigid architectures, flexible architectures and data-dependent architectures.

1.4.1 Rigid DNN Accelerator Architectures

First-generation *rigid DNN inference accelerators* [28,61,78,98] focused their designs on fixed-size clusters of multipliers-and-accumulate units interconnected by means of a fixed tightly-integrated on-chip network fabric specifically tailored to efficiently support a particular dataflow. This is the path that current industrial approaches have followed because of its simplicity in terms of design. For example, the Google TPU [61] is built by interconnecting 256×256 Multiply-Accumulate (MAC) units to a tightly-coupled 2D grid and supports a weight-stationary dataflow, while ShiDianNao [28] groups 8×8 MAC units supporting an output-stationary dataflow. Other commercial design include the Cerebras chips [108], which include a large number of configurable systolic-like arrays within a wafer-scale.

1.4.2 Flexible DNN Accelerator Architectures

Unfortunately, as DNN models evolve at a rapid pace, these fixed designs fail to adapt well to contemporary DNN models.

From the models observed in the MLPerf benchmark suite [107], we can highlight two main sources of inefficiency in terms of performance and energy that are inherent in these first-generation DNN accelerators:

- **Wide range of DNN types**: Practitioners adapt their DNN models to different application budgets. This creates a large variability in terms of DNN sizes and types of layers with diverse computing demands, which impedes a fixed on-chip interconnect fabric to support all of them efficiently, as rigid topologies cause low compute unit utilization [61]. Besides, this makes imperative the need for an architecture able to scale appropriately. However, inflexible on-chip interconnects, such as 2D systolic arrays, cause low compute unit utilization [61] and, due to its rigid 2D shape, scale quadratically, while it could be logarithmically using a tree-based shape, as [31,70] demonstrate. DNN models are continuously evolving featuring different sizes and types of layers, hence leading to varying computing demands:

- **Shape and type diversity**: The high variability in terms of filter and input shapes even within the same DNN impedes a fixed on-chip interconnect fabric to support all of them efficiently, as rigid topologies cause low compute unit utilization [61].
- **Sparsity**: Modern DNN workloads exhibit different degrees of weight and input sparsity due to both network pruning and the use of the non-linear activation functions such as ReLU, respectively.

Exploiting this large diversity in computing demands makes rigid DNN accelerators, which are made up of fixed on-chip topologies, highly ineffective leading to poor scalability, under-utilization of the computing resources, and low energy efficiency [31,61,70].

To overcome these limitations, recent proposals such as FlexFlow [76], MAERI [70] and SIGMA [31] advocate using flexible DNN accelerator fabrics, which can be reconfigured to efficiently map different dataflows and dot product partitions through the creation of dynamic-size clusters (i.e., a set of multipliers computing the same output) in the same hardware substrate. These designs advocate for physically separating the components into three different networks: A Distribution Network (DN) that delivers the data from the SRAM structure (e.g., what is called Global Buffer or GB in the MAERI design) to the multipliers. A Multiplier Network (MN) that integrates a set of multipliers that perform the multiplication operations and a Reduction Network (RN) that consists of a tree-based topology of configurable adders and whose purpose is to accumulate the partial sums generated by the MN. This RN is the key ingredients that enables to configure dynamic-size clusters of multipliers in the same hardware substrate.

1.4.3 Data-Dependent DNN Accelerator Architectures

Other type of accelerators aim to optimize the processing of the DNNs by exploiting properties inherent in the data that is computed [31,33,50,97,114,122,124]. Among them, the accelerators that exploit the sparsity inherent in the DNNs have become one of the main sources of improving the processing of these workloads.

These accelerators have support for sparse execution via supporting an efficient processing of the sparse matrix multiplication. To do so, they utilize compression of one or both operands into formats like CSR, CSC, bitmap, CSF etc., which reduces the memory footprint and the number of multiplications. Some prior works like Eyeriss [22] focus on exploiting sparsity via zero-gating of the multiplier where multiplication by zero is skipped. This saves the number of multiplications but does not reduce the on-chip memory footprint. Eyeriss compresses data between the Global Buffer and the DRAM via run-length coding. Other accelerators such as ExTensor [50], SIGMA [31], GAMMA [122], MatRaptor [114], Outer-Space [97], SpArch [124], among others employ specific hardware structures to avoid not only the multiplication by zero, but also the transfer of the zero values. These accelerators typically map the sparse matrix multiplication operation into the accelerator via a specific dataflow. For example, ExTensor [50] or SIGMA [31] implement an Inner Product dataflow. Others such as Outer-Space [97] or SpArch [124] employ what is called an Outer Product dataflow while other accelerators such as GAMMA [122] or MatRaptor [114] employ a Gustavson's dataflow. The particular dataflow implemented by the accelerator defines the order loop of the sparse matrix multiplication operation and in consequence, defines how the operands have to be processed during the execution. In Chapter 4, we will analyze these three dataflows, with their pros and cons.

1.5 Thesis Motivations and Contributions

We observe three challenges in the state-of-the-art described above.

1. Nowadays, the dominant strategy that researchers adopt for the implementation of almost any new DNN inference accelerator follows a shallow design-space exploration, by which a single (or few) architectural design decisions are analytically investigated before building the particular ASIC-based or FPGA-based prototype. This strategy is acceptable as long as we deal with simple-to-moderate architectural design complexity, as it is the case of first-generation rigid DNN accelerators. In contrast, the higher complexity of flexible DNN accelerators urges for exhaustive design-space exploration to be able to identify the best architectural design decisions to synthesize. Architectural simulators have been extensively used during the design process of CPU and GPU architectures ([12–15,55,115] are just a few examples). Nonetheless, and quite surprisingly, the same has not taken

place until now for DNN accelerator architectures, mostly due to the broad variety of accelerator designs, which makes difficult the elaboration of a tool capable of covering all of them.

2. The reduction networks currently implemented in flexible accelerators do not support efficiently some common cases. For example, the case of folding, in which more partial sums than multipliers are generated, is not managed in the correct way. This leads to inefficiencies that decrease significantly the performance of the accelerators.

3. Current accelerators for sparsity processing only implement a single dataflow, while the diversity of the DNN layers, sizes and sparsity patterns reveal that there is not a single dataflow that may work well for all the layers. We are still missing an accelerator that supports the three dataflows and therefore can be able to adapt its hardware substrate to the characteristics of the DNN layer.

This thesis addresses the three challenges. In this way the main contributions of this thesis are summarized as follows:

1. We present STONNE, a cycle-level microarchitectural simulation framework that can plug into any high-level DNN framework as an accelerator device and perform full-model evaluation (i.e. we are able to simulate real, complete, unmodified DNN models) of state-of-the-art rigid and flexible DNN accelerators, both with and without sparsity support. As a proof of concept, we use STONNE in three use cases: *i)* a direct comparison of three dominant inference accelerators using real DNN models; *ii)* back-end extensions and *iii)* front-end extensions of the simulator to showcase the capability of STONNE to rapidly and precisely evaluate data-dependent optimizations.

2. We demonstrate that the strategy employed by the reduction network to support folding in state-of-the-art flexible accelerators is of paramount importance to ensure high performance. The simplest strategy, which is the one implemented by the current flexible accelerators, consists in sending the different results though the Global Buffer. As we demonstrate in this work, this breaks the computation pipeline, thus degrading performance. On the other hand, the naive strategy of using an accumulation buffer as done in the Google's TPU, solves this problem, but entails significant area and energy overheads. As the second contribution of this thesis, we present

a novel reduction network fabric call STIFT that integrates spatio-temporal accumulation on the same hardware substrate. This way, STIFT enables similar performance as the accumulation buffer strategy, but keeps the area and power in an affordable range.

3. We design and present Flexagon, the first SpMSpM reconfigurable accelerator that is capable of performing SpMSpM computation by using the particular dataflow (i.e., Inner Product, Outer Product or Gustavson's) that best matches each case. Flexagon accelerator is based on a novel Merger-Reduction Network (MRN) that unifies the concept of reducing and merging in the same substrate, increasing efficiency. Additionally, Flexagon also includes a 3-tier memory hierarchy, specifically tailored to the different access characteristics of the input and output compressed matrices. Using detailed cycle-level simulation of contemporary DNN models from a variety of application domains, we show that Flexagon achieves average performance benefits of 4.59×, 1.71×, and 1.35× with respect to the state-of-the-art SIGMA-like, SpArch-like and GAMMA-like accelerators (265% , 67% and 18%, respectively, in terms of average performance/area efficiency).

1.5.1 Publications Derived from this Thesis

All the contributions of this thesis have been presented in relevant international peer reviewed conferences, workshops and journals.

In particular, as for the first contribution (i.e., STONNE simulator), the following are direct results (in terms of presentations and publications) that have been derived from this thesis:

- A preliminar description of STONNE simulator published and presented at the AccML workshop co-located with ISCA 2020 [80].
- A short handy description of STONNE simulator published in the CAL journal [81].
- A detailed description of STONNE simulator published and presented at the prestigious conference IISWC 2021 [82].
- A 4-hour tutorial of the STONNE simulator co-located with ASPLOS 2021 conference [85].

- A description of a framework built on the top of STONNE called SST-STONNE (see Section 2.7.2) published and presented at the ModSim 2022 conference [84].
- A description of OMEGA, a framework that builds on the top of STONNE to model GNNs (see Section 2.7.1) published and presented at IPDPDS 2022 conference [40] and was awarded with a best paper nomination.

As for the second contribution (i.e., STIFT), two are the direct results:

- A detailed description of STIFT published and presented at the prestigious conference NoCs 2021 [89], and that was awarded with a best paper nomination.
- An extended description of STIFT published in JETC journal [83].

Finally, as for the third contribution (i.e., Flexagon), the paper has been submitted to the prestigious conference ASPLOS 2023 [86], and at the moment of writing this document, is under review.

1.6 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 presents, describes and evaluates the first contribution of this thesis (i.e., the STONNE simulator), which will be used as the reference tool for the rest of the thesis.
- Chapter 3 presents and evaluates STIFT proposal for efficient folding in flexible state-of-the-art accelerators.
- Chapter 4 presents and evaluates Flexagon accelerator for SpMSpM computation with respect to state-of-the-art sparse accelerators.
- Chapter 5 summarizes the main conclusions of the thesis and points out future lines of work.

Chapter 2

STONNE: Enabling Cycle-Level Microarchitectural Simulation for DNN Inference Accelerators

2.1 Introduction

Deep Neural Networks (DNNs) constitute nowadays a promising breakthrough for a large number of artificial intelligence (AI) applications [117].

The fact that their inference phase must be primarily done *in-situ* has paved the way for the development of a plethora of accelerator architectures so as to maximize performance per watt while meeting latency and energy-efficiency demands ([22, 31, 33, 61, 70, 76, 78, 98] are a few examples). The key behind all of these recent architectures has been the capture of the different patterns of data reuse in what is known as a dataflow [23, 68] and the use of data-dependent optimizations to reduce computation and memory footprint [33].

First-generation *rigid* DNN inference accelerators ([28, 61, 78, 98]) focused their designs on fixed-size clusters of multipliers-and-accumulate units interconnected by means of a fixed tightly-integrated on-chip network fabric specifically tailored to efficiently support a particular dataflow. For example, the Google TPUv1 [61] is built by interconnecting 256×256 Multiply-Accumulate (MAC) units to a tightly-coupled 2D grid and supports a weight-stationary dataflow, while ShiDianNao [28] groups 8×8 MAC units supporting an output-stationary dataflow.

Unfortunately, as DNN models evolve at a rapid pace, these fixed designs fail to adapt well to the great diversity of layer types and dimensions in contemporary proposals. Table 2.1 shows seven popular DNN models considered in this chapter. These models fall into three different application domains that mostly cover the diversity of machine learning models in the MLPerf benchmark suite [106], and represent different design tradeoffs for accuracy, memory requirements and computational complexity. In particular, we consider Mobilenets-V1 (M) [53], Squeezenet (S) [56], Alexnet (A) [67], Resnets-50 (R) [49], VGG-16 (V) [113], SSD-Mobilenets (S-M) [75] and BERT (B) [27]. From the data in this table, we can highlight two main sources of inefficiency in terms of performance and energy that are inherent to these first-generation DNN accelerators:

1. *Wide range of DNN types*: Usually, AI practitioners adapt their DNN models to different application budgets. This creates a large variability in terms of DNN sizes and types of layers (see the forth column in Table 2.1) with diverse computing demands, which impedes a fixed on-chip interconnect fabric to support all of them efficiently, as rigid topologies cause low compute unit utilization [61]. Besides, this makes imperative the need for an architecture able to scale appropriately. However, inflexible on-chip interconnects such as 2D systolic arrays cause low compute unit utilization [61] and, due to its rigid 2D shape, scale quadratically, while it could be logarithmically using a tree-based shape, as [31,70] demonstrate.

2. *Sparsity*: Modern DNN workloads exhibit different degrees of weight and input sparsity due to both network pruning and the use of the non-linear activation functions such as ReLU, respectively. Table 2.1 shows the significant state-of-the-art average weight sparsity ratio (from 60% to 90%) after applying an unstructured weight pruning approach similar to that described by Zhu et al. [126].

Exploiting this large diversity in computing demands makes rigid DNN accelerators, which are based on fixed on-chip topologies, highly ineffective, leading to poor scalability, under-utilization of the computing resources, and low energy efficiency [31,70].

To overcome these limitations, recent proposals such as FlexFlow [76], MAERI [70] and SIGMA [31] advocate using *flexible* DNN accelerator fabrics, which can be reconfigured to efficiently map different dataflows and dot product partitions through the creation of dynamic-size clusters (i.e., a set of multipliers computing the same output) in the same hardware substrate. Of course, this

Domain	DNN Model	Sparsity	Dominant Layer Types
Image Classification	Mobilenets-V1 (M)	75%	Factorized Convolution (FC)
			Linear (L)
	Squeezenet (S)	70%	Squeeze Convolution (SC)
			Expand Convolution (EC)
	Alexnet (A)	78%	Convolution (C)
			Linear (L)
	Resnets-50 (R)	89%	Residual Function (RF)
			Convolution (C)
	VGG-16 (V)	90%	Convolution (C)
			Linear (L)
Object Detection	SSD-Mobilenets (S-M)	75%	Factorized Convolution (FC)
			Linear (L)
Language Processing	BERT (B)	60%	Transformer (TR)
			Linear (L)

Table 2.1: Contemporary DNN models explored in this chapter. ImageNet [110], COCO [74] and squad-1.1 [104] dataset have been used to train the image classification, object detection and language processing models, respectively.

flexibility comes at the cost of increased architectural complexity that urges for a more exhaustive design-space exploration for fine tuning before building the particular ASIC-based or FPGA-based DNN accelerator.

Additionally, other works are exploring *data-dependent* optimizations in DNN accelerators that try to reduce computation and memory footprint by exploiting hardware optimizations based on the input data.

For example, SnaPEA [33] implements a data-dependent optimization that leverages the fact that there are no negative values in the input values of a Convolutional Neural Network (CNN). This approach statically re-orders at compile time the weights according to their signs, and periodically performs in hardware a single-bit sign check on the partial sum during the execution. Once the partial sum drops below zero, the rest of the computations are cut off, since the output value will inevitably be zero after applying the typical ReLU activation function in CNNs. In these cases, it is crucial to get access to the precise data values that will be used during the inference procedure.

Microarchitectural simulators have been extensively used during the design process of CPU and GPU architectures ([12–15,55,115] are just a few examples), albeit as we explain in Section 2.2, most recent efforts have focused on using analytical models to describe an accelerator design by means of simple yet insightful formulas. However, as we also demonstrate in Section 2.2, contemporary analytical models, while very useful for exploring Pareto-optimal accelerator parameters [29,68] lag far behind in timing accuracy when modeling more complex flexible architectures, and when running non-trivial computation (e.g., sparse computation or DNN layers that do not map well onto the accelerator substrate and lead to compute under-utilization) or data-dependent optimizations. In these cases, analytical models are not able to capture performance bottlenecks or unexpected behaviors that may occur during a real DNN full-model execution.

To the best of our knowledge, there is still no detailed, cycle-level, open-source microarchitectural simulator for extensive and accurate design-space exploration of DNN inference accelerators (further details are given in Section 2.2 and are summarized in Table 2.2). To bridge this gap, in this chapter we present STONNE (which stands for *Simulation TOol of Neural Network Engines*), the first attempt to derive a cycle-level, highly-modular and highly-extensible simulator for DNN inference accelerator microarchitectural exploration[1]. STONNE builds on the observation that most current DNN accelerator architectures can be logically organized as three configurable on-chip network fabrics (distribution network, multiplier network, and reduction network) and the corresponding memory controller and buffers, and provides an easily expandable and configurable set of microarchitecture modules (e.g., a tree-based distribution network or a sparse memory controller) that, conveniently selected and combined, can faithfully simulate both rigid DNN accelerators (e.g., the Google TPU [61]) and flexible DNN accelerators (e.g., MAERI [70] or FlexFlow [76]), including those exploiting sparsity (e.g., SIGMA [31]). Additionally, and unlike prior tools, STONNE is directly integrated with the widely used PyTorch DL framework [5] as an accelerator device, which enables cycle-level simulation of a wide variety of accelerator microarchitectures running complete DNN models and precise evaluation of data-dependent optimizations used in a plethora of DNN accelerators (e.g. SnaPEA [33]).

We see the following contributions in this chapter:

- We demonstrate the value of cycle-level simulation for accurate design-space exploration of DNN accelerators (Section 2.2). Particularly, we show that a state-of-the-art analytical model can underestimate the number of clock cycles for the execution of certain DNN layers by more than 400%.

- We present (Sections 2.4 and 2.4) and validate (Section 2.5) STONNE, the first simulator, to the best of our knowledge, that is connected as an accelerator device with a contemporary DL framework (PyTorch [5]), and enables cycle-level microarchitectural simulation of inference accelerators

[1]Support of training procedures in STONNE is left as future work.

(with both dense and sparse computation support) running complete DNN models.

- We demonstrate the usefulness, versatility and capability of STONNE via three diverse use cases. In the first one, we perform a direct comparison between TPU, MAERI and SIGMA type inference architectures running the seven DNN models shown in Table 2.1. In the other two use cases, we demonstrate the capacity of STONNE to be extended and to model data-dependent optimizations. Particularly, the second use case considers the modification of the back-end of the simulator by implementing SnaPEA [33], whereas the third use case analyzes the potential of static filter scheduling in DNN sparse accelerators, which entails modifications to STONNE's front-end.

We demonstrate the extensibility and usability of STONNE by presenting two external tools called OMEGA (Section 2.7.1) and SST-STONNE (Section 2.7.2) that have been developed to give birth to the modelling of accelerators for more complex applications such as Graph Neural Networks (GNNs) or heterogeneous multi-accelerator systems.

The rest of the chapter is organized as follows. First, in Section 2.2, we review the state-of-the-art in simulation tools for DNN accelerators. Then, Section 2.4 explains the organization of the STONNE framework and Section 2.4 describes the details of the flexible accelerator microarchitectures that STONNE simulates. Subsequently, Section 2.6.1 and Section 2.6.2 present and evaluate the two case studies used to demonstrate the new horizons opened by STONNE. Section 2.7 shows the description of two additional tools that are based on STONNE. Finally, Section 2.8 outlines the main conclusions of this chapter.

2.2 Motivation and Related Work

Table 2.2 shows a qualitative comparison of STONNE with respect to contemporary publicly available tools for design-space exploration of DNN inference accelerators. For the comparison, we consider five desirable features that a DNN inference simulator should meet: 1) *cycle-level simulation*; 2) *support for both rigid and flexible DNN accelerator architectures*; 3) *support for sparse executions*; 4) *ability to perform complete evaluations of deep learning models*; and 5) *ability to implement and evaluate data-dependent optimizations*. The rest of this section compares these tools in detail.

	Cycle Level	Architecture Type	Sparsity Support	FullModel Eval	DataDep Opt
MAGNet DNNBuilder	✗	None	✗	✗	✗
GEMMINI	✓	Rigid	✗	✗	✗
MAERI BSV	✓	Flexible	✗	✗	✗
SIGMA RTL	✓	Flexible	✓	✗	✗
SCALE-Sim	✗	Rigid	✗	✗	✗
MAESTRO TimeLoop	✗	Both	✗	✗	✗
SMAUG	✓	Rigid	✗	✓	✗
STONNE	✓	Both	✓	✓	✓

Table 2.2: State-of-the-art Simulators for DNN Accelerators.

2.2.1 Analytical Modeling

SCALE-Sim [30], MAESTRO [68] and TimeLoop [29] have recently been proposed as frameworks that enable the analysis of different dataflows in DNN architectures. These tools are very powerful for fast exploration of high-level architectural details, as they are based on analytical models that calculate the degree of data reuse and computations using simple equations. These types of simulators work accurately when it comes to rigid architectures as they are simple enough to be represented by a set of formulas. However, when the complexity of the accelerator grows and/or the computation does not follow regular patterns, these models fail to faithful capture the exact behavior of the architecture.

Figures 2.1, 2.2 and 2.3 show this fact quantitatively. First, SCALE-Sim only models simple rigid architectures (e.g., TPU-like systolic arrays) and do not have support to handle sparsity. Figure 2.1 shows the number of cycles obtained with this analytical model and with the cycle-level execution model implemented in STONNE after running eight different representative layers (Squeeze, Expand, Factorized and Regular Convolutions – *SC*, *EC*, *FC*, *C*; Linear – *L*; and Transformers – *TR*) extracted from Squeezenet (S), Resnets-50 (R), Mobilenets (M) and BERT (B). We have configured both models to simulate an Output-Stationary systolic array varying the size of the array of processing elements (PEs) from 16×16 to 64×64. As we can see, we obtain for the three configurations almost the same

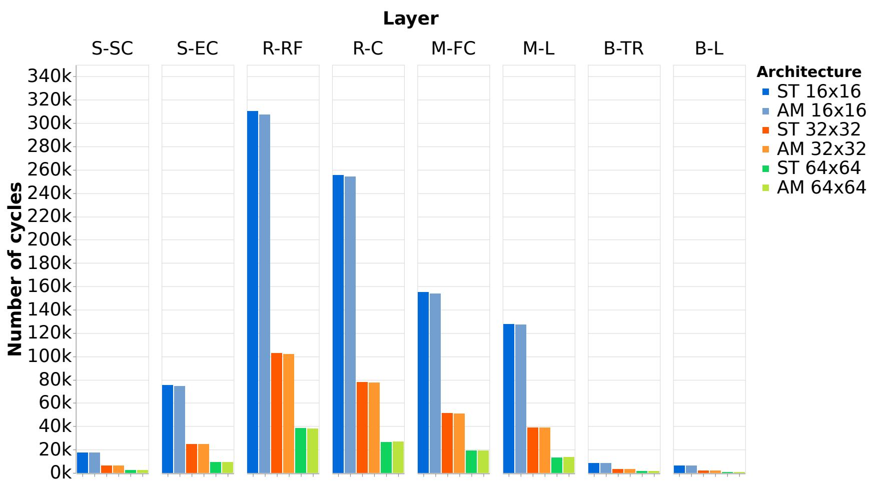


Figure 2.1: Runtime for 8 DNN layers run on a rigid TPU-like inference accelerator modeled using STONNE (ST) and an analytical model (AM). We model different systolic 2D grid sizes (16×16, 32×32 and 128×128). We use the following notation when plotting the results: `X-Y`, where `X` is the DNN model and `Y` is the layer type.

number of cycles for both alternatives, demonstrating that analytical models are valuable tools when it comes to rigid DNN accelerator architectures.

Contrarily, we have observed that analytical models fail to faithfully capture microarchitectural details of flexible DNN accelerator architectures, and therefore, are not appropriate to identify many of the bottlenecks or unexpected behaviors that may occur during a real DNN full-model execution. To demonstrate this claim, we perform a set of experiments for a 128-multiplier flexible dense accelerator simulating MAERI [70], using the detailed analytical model provided by the authors of the MAERI paper [70].

Figure 2.2 plots the number of cycles reported by both STONNE and the analytical model for different global buffer bandwidth (i.e., number of elements that the global buffer can deliver per cycle to the processing elements) configurations. In both cases we modify the parameter that controls the bandwidth to consider 128 (full bandwidth), 64 and 32 elements/cycle after running on MAERI a *dense* execution with the same layers extracted from the DNN models used before. As we can see, the analytical model perfectly matches the performance obtained with STONNE when there is full bandwidth (average difference of 1.03%), as this ideal case can be easily represented by a set of mathematical formulas. However, as the bandwidth decreases, STONNE begins to report a much higher number

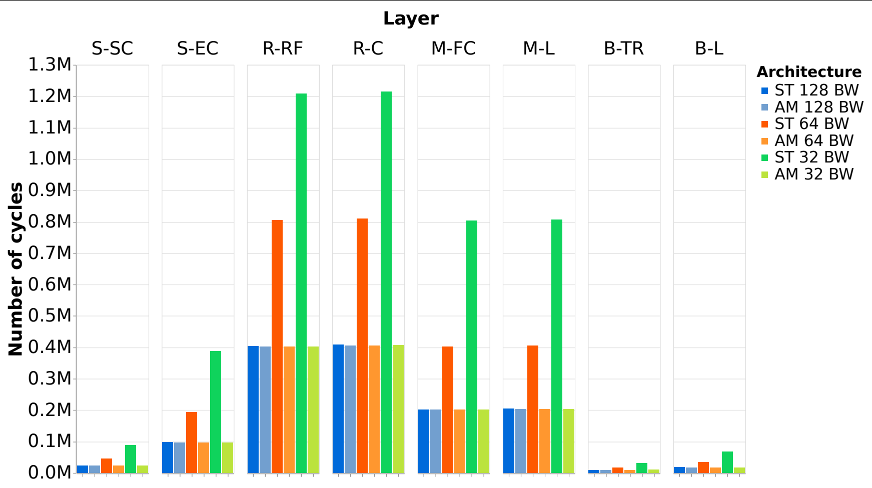


Figure 2.2: Runtime for 8 DNN layers run on a rigid MAERI-like inference accelerator modeled using STONNE (ST) and an analytical model (AM). We model a flexible 128-multiplier MAERI-like architecture with different Global Buffer I/O Bandwidth (32, 64 and 128 elems/cycle). We use the following notation when plotting the results: `X-Y`, where `X` is the DNN model and `Y` is the layer type.

of cycles. This is due to the ability of a cycle-level simulator like STONNE to faithfully capture the stalls produced in the architectural pipeline and that arise as a result of the increasing number of conflicts in the MAERI's distribution and reduction networks. The difference between the results reported by STONNE and the analytical model for 32 elements/cycle increases up to 400% (see M-FC in Figure 2.2), alerting about the important limitations of the analytical models.

Furthermore, we also observe that an analytical model is not capable of accurately representing DNN *sparse* executions. Figure 2.3 shows the same executions as before, but this time we have configured STONNE to model a sparse accelerator like SIGMA [31]. Again, we compare the results against the analytical model provided by the authors of SIGMA [31]. This time, we assume full bandwidth and variable sparsity ratio of the matrices between 0% and 90%. In this case, we also observe a perfect match between both STONNE and the analytical model when the sparsity ratio is 0%, but this similarity begins to diverge as the sparsity ratio increases (diverging up to 92% for a sparsity ratio of 90%).

The reason for this difference is that the actual distribution of zeros in the matrices, which affects the cluster sizes, and in the end, the performance obtained

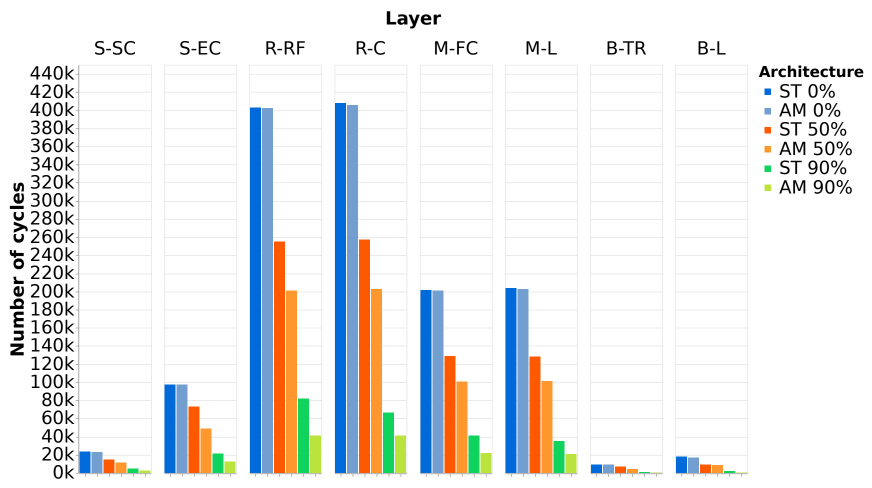
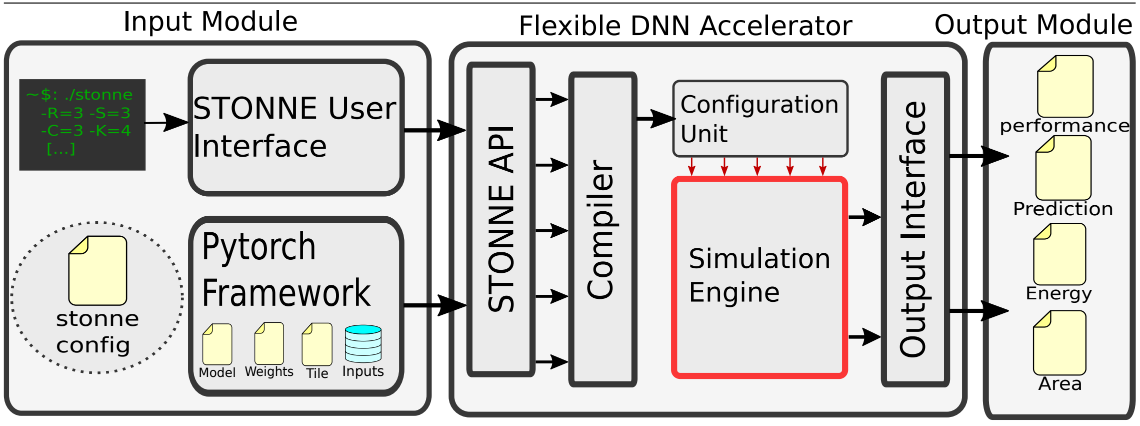


Figure 2.3: Runtime for 8 DNN layers run on a rigid SIGMA-like inference accelerator modeled using STONNE (ST) and an analytical model (AM). We model a flexible 128-multiplier SIGMA-like architecture with different sparsity ratios (0%, 50% and 90%). We use the following notation when plotting the results: `X-Y`, where `X` is the DNN model and `Y` is the layer type.

by the architecture, cannot be modeled analytically. Instead, cycle-level, full-model evaluations with real weight values are needed to capture it.

2.2.2 Cycle-level Simulation

Cycle-accurate simulation of DNN accelerators have been adopted by just a few simulators. First, GEMMINI [48] is a systolic-array accelerator generator based on matrix multiplication for the investigation of SoC integration of such accelerators. However, since the aim of this tool is the generation of RTL code for rigid accelerators, it is not suitable for fast exploration of the design space. Among all the alternatives, only the MAERI BSV [3], SIGMA RTL [6] implementations and SMAUG [120] claim to model flexible accelerator architectures with cycle-level precision. However, none of them really allows for efficient design-space exploration and rapid prototyping. MAERI BSV and SIGMA RTL are just two limited hardware implementations in Bluespec HDL [1] and Verilog, respectively, written to demonstrate the effectiveness of these two flexible architectures. Hence, they are not prototypes adapted to be extended or to carry out the inference procedure of a complete DNN model or perform design-space exploration. Although SMAUG is aimed to efficiently support full-model simulation of flexible



```
~$: ./stonne
  -R=3 -S=3
  -C=3 -K=4
  [...]
```

Figure 2.4: High level overview of the STONNE framework.

architectures, actually this flexibility only means that it is able to execute any layer with any tile configuration mapping. However, the architectures currently being supported are a systolic array and the NVIDIA Deep Learning Accelerator (NVDLA), which cannot be considered flexible accelerators. Besides, since SMAUG is a trace-based simulator, it is unable to run whole modern DNN models, and therefore, has to resort to a sampling approach, which impedes its use for real full-model evaluations or examine data-dependent architectural optimizations. To address all the above shortcomings, STONNE is able to fully simulate all operations in a DNN in a reasonable timeframe. Besides, STONNE is endowed with sparsity support, is capable of performing full-model evaluation of any contemporary DNN model and as the simulator works with real data, it is suitable to evaluate data-dependent optimizations.

2.3 STONNE Framework

STONNE is a cycle-level microarchitectural simulator for DNN inference accelerators. STONNE is open-sourced under the terms of the MIT license. To allow for full-model evaluations, STONNE is connected with a Deep Learning (DL) framework (PyTorch [5] and Caffe [2] DL frameworks in the current version). Therefore, STONNE can fully execute any dense and sparse DNN model supported by the DL framework that uses as its front-end. The simulator has been written entirely in C++, following the well-known GRASP and SOLID programming principles of object-oriented design [79]. This has simplified its development and makes it easier the implementation of any kind of DNN inference accelerator microarchitecture, tile configuration mapping and dataflow.

Figure 2.4 shows a high-level view of STONNE with its three major modules for full-model simulation flows. These three components work in conjunction as follows. First, the **Input Module** (see Section 2.3.2) is used to define the DNN to be run and to load the parameters of the layer and the initial inputs and weights onto the simulated accelerator. Once the accelerator has been configured, the **Simulation Platform** module (further details in Section 2.3.1) carries out a cycle-by-cycle microarchitectural simulation of the accelerator during the execution of the feed-forward computation of the layer (i.e., the inference procedure), collecting statistics during the process. After this, the results collected during the execution of the layer are sent back to the CPU, and Finally, once the simulation of each simulated layer is completed, the **Output Module** (further details in Section 2.3.3) takes in the values of the counters collected by the simulated architecture and produces several useful statistics of the execution, such as performance and energy consumption. Next, we describe every module in detail:

2.3.1 Simulation Platform

The simulation platform constitutes the principal block (see the central block in Figure 2.4), since it includes the implementation of the simulated DNN accelerators (i.e., `Simulation Engine`) whose different internal microarchitecture modules allow to compose and cycle-by-cycle simulate both rigid and flexible DNN accelerators. These modules are further described in Section 2.4. The composition of each accelerator (i.e., the selection of the microarchitecture modules) is defined by the user through a hardware configuration file given by the input module. These modules are configured through the `Configuration Unit` at runtime according to a set of signals generated by the `Mapper` based on the configured microarchitectural modules and the DNN layer type and shape to be executed.

The simulation platform is interfaced by means of a set of coarse-grained instructions called the `STONNE API` (Table 2.3). This API is the manner in which the input module (i.e., the DL framework) can interact with the simulated accelerator, configuring its simulation engine according to the user configuration file, loading layer and tile parameters, and configuring the weights and the inputs addresses in the main memory. The STONNE API can be easily extended to support new instructions. Furthermore, this API design allows to build tools upon the `Simulation engine` module. Representative examples of this characteristic are

Instruction	Description
CreateInstance	Creates an instance of STONNE.
ConfigureCONV	Configures the accelerator to run a convolution operation.
ConfigureLinear	Configures the accelerator to run a fully-connected layer.
ConfigureDMM	Configures the accelerator to run a matrix multiplication.
ConfigureSpMM	Configures the accelerator to run a sparse matrix multiplication.
ConfigureMaxPool	Configures the accelerator to run a max pooling layer.
ConfigureData	Configure weights, inputs and outputs addresses from the CPU to the accelerator memory.
RunOperation	Launches the simulation according to the current configuration of the architecture.

Table 2.3: STONNE API Instruction Set.

the OMEGA tool (see Section 2.7.1) and SST-STONNE (see Section 2.7.2 which implement its own input and input modules but utilize the STONNE API to connect to the `Simulation engine` and create cycle-level tools able to model more complex applications such as GNNs or heterogeneous systems.

2.3.2 Input Module

Due to the flexibility that the `STONNE API` provides, the simulator can be fed easily using a standard DL framework. To this end, we have modified the PyTorch (Caffe is also supported) DL framework[2] (see the left block in Figure 2.4) to connect it to the simulator and to make it able to run an instance of the `Simulation Engine` transparently to the user. This way, a PyTorch user just needs to select the typical *.pb* file with the weights, choose the inputs (e.g., a set of images or sentences) and briefly modify each DNN model to include the path of the hardware configuration file with the parameters of the accelerator to simulate, and the tile configuration for every layer.

Furthermore, since PyTorch requires a more complicated installation and use, apart from this mode of execution, we have also enabled the `STONNE User Interface`. This is basically a tool inside STONNE in which the user is presented with a prompt and a set of well-defined commands to load any layer and tile parameters onto a selected instance of the simulator, and run it with random

[2]Other DL frameworks, such as Tensorflow, can be easily integrated with STONNE using the same STONNE API philosophy.

weights and input values. This allows for faster executions, facilitating rapid prototyping and debugging.

2.3.3 Output Module

Once a simulation for a certain layer has been completed, this module is used for reporting simulation statistics such as performance, compute unit utilization, and activity counts of different components such as wires, FIFOs or SRAM usage (i.e., number of accesses). In particular, STONNE reports two different output files: First, a general file in *json* format that includes a summary of the statistics and facilitates their processing through user-created scripts; Second, a *counter file* written in a customized format which contains the activity counts for each component of the architecture (e.g., multiplier, wire, adder, etc). From these activity counts, the output module is able to report the amount of energy consumed by the simulated architecture. To do so, STONNE includes a script that given the counter file and a table-based energy model similar to Accelergy [119], computes the total consumed energy taking into account the cycle-level activity stats for each module and the corresponding energy costs. Similarly, the area numbers are obtained by employing a table-based model, calculating the final area based on the architectural parameters and the area cost of each one. Obviously, these statistics depend, for example, on the particular data format (e.g., FP16 or INT8) utilized to represent the parameters of the DNN model. So, STONNE includes different energy and area tables that can be used. To derive these tables, we ran synthesis using Synopsys Design-Compiler, and place-and-route using Cadence Innovus on each module of the simulated accelerator (further details in Section 2.5).

2.3.4 Walk-Through Example

This section clarifies the interaction between the Input Module (i.e., PyTorch) and the Simulation Platform with a walk-through example illustrating the execution of a simple DNN model composed of 5 typical DNN operations: *Conv2d*, *MaxPool*, *Linear*, *sparse_mm* and *log_softmax*. First, Figure 2.5 graphically shows this interaction over time (x-axis) when running these operations. As we can see, the execution is driven layer-by-layer by the DL framework (PyTorch in this case) that offloads compute-intensive layers (e.g., a convolution layer) to the simulated accelerator, and runs natively in the real CPU those layers that are not compute-intensive enough for acceleration (e.g., a SoftMax layer).

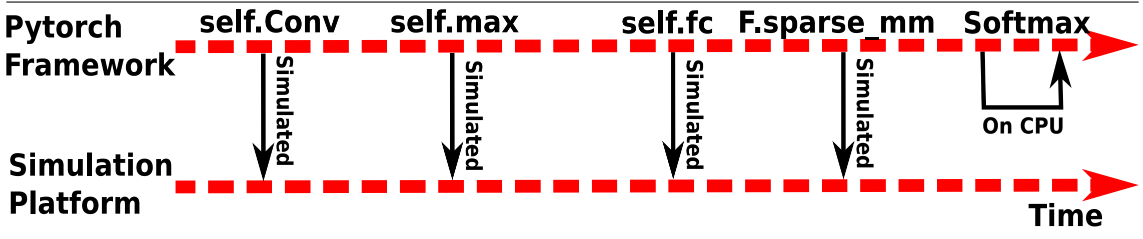
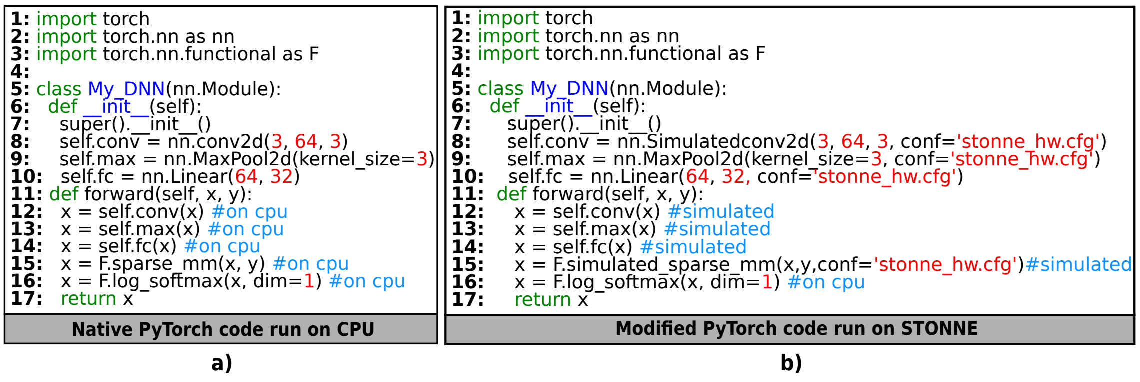


Figure 2.5: DNN simulation example.



```
1: import torch
2: import torch.nn as nn
3: import torch.nn.functional as F
4:
5: class My_DNN(nn.Module):
6:   def __init__(self):
7:     super().__init__()
8:     self.conv = nn.conv2d(3, 64, 3)
9:     self.max = nn.MaxPool2d(kernel_size=3)
10:    self.fc = nn.Linear(64, 32)
11: def forward(self, x, y):
12:    x = self.conv(x) #on cpu
13:    x = self.max(x) #on cpu
14:    x = self.fc(x) #on cpu
15:    x = F.sparse_mm(x, y) #on cpu
16:    x = F.log_softmax(x, dim=1) #on cpu
17:    return x
```

Native PyTorch code run on CPU

a)

```
1: import torch
2: import torch.nn as nn
3: import torch.nn.functional as F
4:
5: class My_DNN(nn.Module):
6:   def __init__(self):
7:     super().__init__()
8:     self.conv = nn.Simulatedconv2d(3, 64, 3, conf='stonne_hw.cfg')
9:     self.max = nn.MaxPool2d(kernel_size=3, conf='stonne_hw.cfg')
10:    self.fc = nn.Linear(64, 32, conf='stonne_hw.cfg')
11:  def forward(self, x, y):
12:    x = self.conv(x) #simulated
13:    x = self.max(x) #simulated
14:    x = self.fc(x) #simulated
15:    x = F.simulated_sparse_mm(x,y,conf='stonne_hw.cfg')#simulated
16:    x = F.log_softmax(x, dim=1) #on cpu
17:    return x
```

Modified PyTorch code run on STONNE

b)

Figure 2.6: a) Native PyTorch code for CPU. b) Modified PyTorch code for STONNE.

More specifically, for each intensive computational operation such as a convolution (*nn.Conv2d)*, the DL framework configures the corresponding memory addresses onto the simulator (using the *ConfigureData* instruction) and configures the layer to be run (*ConfigureCONV* instruction). Then, the simulator takes control and runs the operation on a cycle basis on the simulated accelerator. Once the simulator finishes, it reports the statistics through the Output Module, notifies back the DL framework and returns it control to continue with the next operations (*nn.MaxPool*, *nn.Linear* and *F.sparse_mm*). As shown, those operations that are not worth for acceleration (e.g., softmax operation) are executed directly by the DL framework (as it would be done in a real scenario), so correctness of the entire execution is ensured.

Figure 2.6(a) shows the native PyTorch code that would be used to run these operations on a CPU (or GPU), while Figure 2.6(b) shows the required modifications to off-load the aforementioned intensive computational operations onto STONNE (lines 8, 9, 10 and 15). As we can see, each operation instance to be off-loaded is replaced by a similar operation that adds the prefix *Simulated* to its name. This allows PyTorch to distinguish when the operation has to be run on STONNE rather than natively on the CPU (or GPU). Furthermore, the arguments of the operations have to be extended to include the hardware configuration file (i.e., *stonne_hw.cfg*) that will be used by the simulation platform to create the instance of the simulated accelerator. As it may be appreciated, those lines that do not change will run normally on PyTorch, maintaining the correctness of the execution.

Note that mapping of non matrix multiplication layers on STONNE, such as Pooling (e.g., *nn.MaxPool*) and batch normalization, is not a problem, as they can be easily supported in flexible accelerator architectures without additional specific SIMD modules (as required in some other architectures) [70]. Even crossing layers (i.e., kernel fusion) operations could be mapped onto the processing units of a flexible architecture. As we illustrate with use cases 2 and 3 in Section 2.5, and in Chapters 3 and 4, the design of STONNE allows to be easily extended to incorporate other hardware and software optimizations and even entire DNN hardware accelerators that support other characteristics and dataflows.

2.4 STONNE Simulation Engine

STONNE builds on the observation that most current DNN accelerator architectures can be logically organized as three configurable network fabrics (distribution network, multiplier network, and reduction network) and the corresponding memory controller and buffers [65], and provides an easily expandable and configurable set of microarchitecture modules (for buffers, on-chip data delivery and memory controllers) that, conveniently selected and combined, can model both rigid and flexible DNN accelerators (see Figure 2.7).

2.4.1 On-Chip Networks

All the on-chip components are interconnected by using a general three-tier network composed of a Distribution Network (DN), a Multiplier Network (MN), and a Reduce Network (RN), inspired by the taxonomy of on-chip communication flows within DNN accelerators [70]. These networks can be configured to support any topology to model state-of-the-art accelerators such as the TPU [61], Eyeriss-v2 [24], ShDianNao, SCNN, MAERI [70] and SIGMA [31], among others. First, to compute all the MAC operations of a certain DNN layer, the DN distributes the required weights, activations or partial sums from the GB towards the MN.

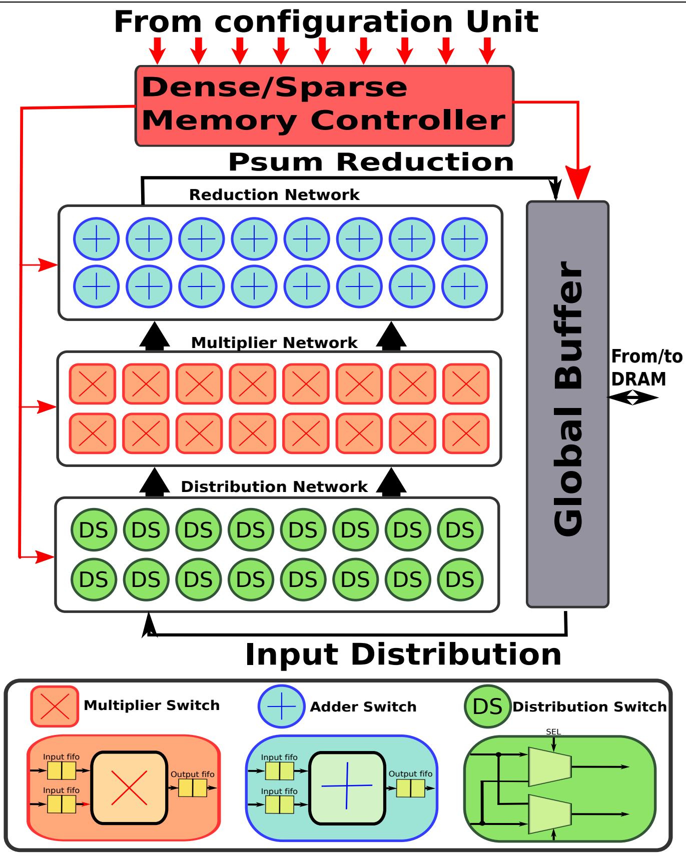


Figure 2.7: Overview of the general flexible DNN inference accelerator considered in STONNE.

To enable all types of dataflows, the DN must provide support for unicast, multicast and broadcast data delivering. As explained in Section 2.4.1.1, this is accomplished through different possible configurations of the Distribution Switches (DS) shown in the figure. After the distribution, the multipliers at the MN carry out the multiplication operations, generating the operands of the partial sums to be accumulated. Finally, the RN network is equipped with adders that implement the required accumulations. As we have seen, the secret sauce of STONNE is that the code of the simulator is properly designed to completely change or modify the networks at user preferences very easily. Next, we describe the different topologies of the three networks (DN, MN and RN) currently supported in STONNE that are basic building blocks of state-of-the-art

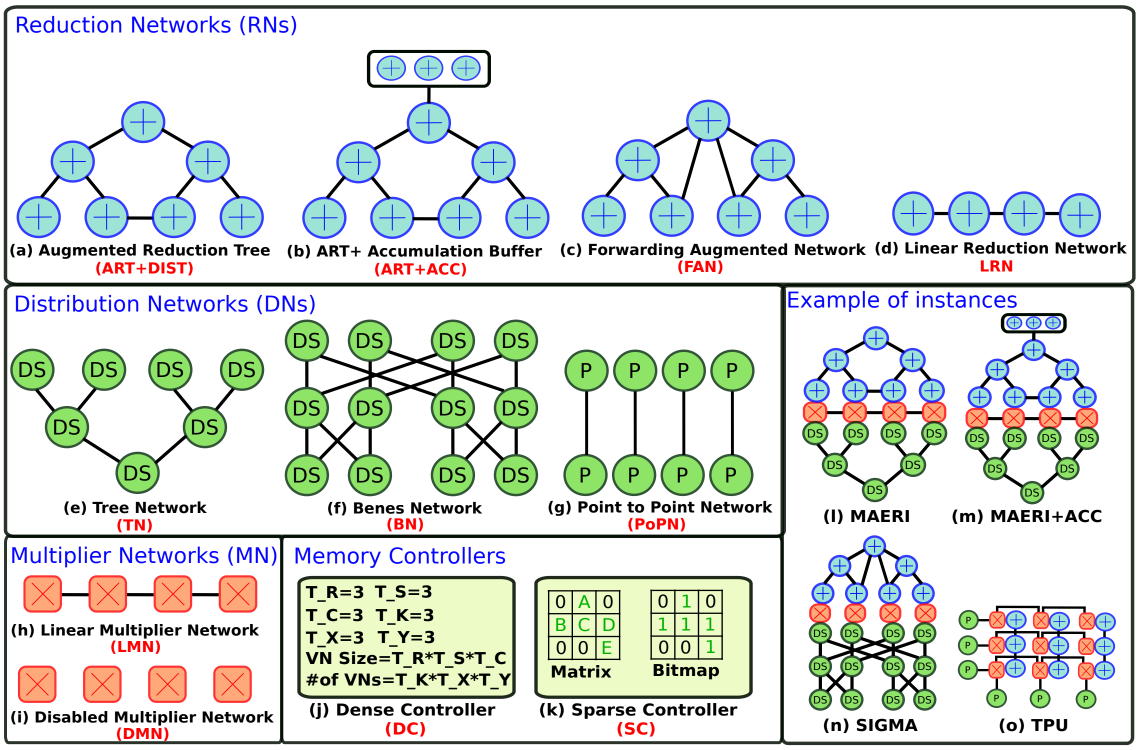


Figure 2.8: Basic building blocks of STONNE. We illustrate how the building blocks can be easily combined to configure four DNN inference accelerators.

accelerators such as the Google's TPU, Eyeriss-v2, ShDianNao, SCNN, MAERI and SIGMA, among others.

2.4.1.1 Distribution Networks (DNs)

In order to deliver the data from the Global Buffer (GB) to the MN, we implement the next DNs:

- **Tree Network (TN)**: A TN (illustrated in Figure 2.8(e)) is a binary-tree-based network topology inspired by the MAERI distribution network that is replicated as many times as the number of read ports available in the GB, and that provides single-cycle unicast, multicast and broadcast data delivery from the GB to the multipliers [70]. Each node of the TN is just a low-cost bufferless Distribution Switch (DS) that selects whether to send the input to one or both outputs using a bit vector that is set by the input source. Due to the simplicity of the DSs, the DN can provide single-cycle traversals from the GB to the MN for every piece of data.

- **Benes Network (BN)**: A BN (illustrated in Figure 2.8(f)) is an N-input N-output non-blocking topology with $2 \times log(N) + 1$ levels, each with N tiny 2×2 switches. This DN is implemented in SIGMA [31] and ensures efficient single-cycle unicast, multicast and broadcast data delivery from the GB to the MN. Differently from the TN, it is necessary just one BN to connect the number of read ports available in the GB to the MN. To do so, in this case, every DS requires two control bits, one for selecting a vertical output and one for diagonal output. As for the DSs of the TN, these switches are also capable of providing single-cycle traversals from the GB to the MN for every piece of data.

- **Point to Point Network (PoPN)**: Unlike the two DNs described above, the PoPN (illustrated in Figure 2.8(g)) provides only unicast data delivery from one source point (typically the GB) to a destination (typically a multiplier). This is the basic component to build an interconnect for a systolic array such as the TPU.

2.4.1.2 Multiplier Networks (MNs)

These networks are made up of a set of Multiplier Switches (MSs) that can be configured to act as either forwarders or multipliers. The forwarding configuration is used to forward psums from the GB to the RN so that folding[3] can be supported, whereas the multiplier configuration mode is utilized to compute a multiplication between a weight and an input value. In case folding is needed (further details in Section 2.4.2) and the accumulation buffer is disabled, the architecture would need to allocate one extra MS for each cluster to perform the forwarding of the psums calculated in the previous iterations of the same cluster. Currently, we support two MN topologies:

- **Linear Multiplier Network (LMN)** (Figure 2.8(h)): This RN is capable of leveraging the spatio-temporal data reuse (e.g., when processing the sliding window operation of convolution DNN layers) by using forwarding links between each pair of multipliers. This reduces the bandwidth pressure on

[3]Folding is utilized when a dot product needs more multiplication operations than the number of multiplier units available in hardware. Then, the dot product is "folded" to be processed in several sequential steps and partial results should be accumulated and taken at inter-steps boundaries.

the memory and on the DN by reusing data across different multipliers. The LMN is utilized in several DNN accelerators (such as MAERI and TPU).

- **Disabled Multiplier Network (DMN)** (Figure 2.8(i)): Removes completely the forwarding links between the multipliers, disabling their communication, and is aimed at performing basic GEMMs. This MN is presented in DNN accelerators such as SIGMA [31] and SpArch [124] whose basic primitive is the GEMM operation, and therefore, the sliding window operation has no longer effect.

2.4.1.3 Reduction Networks (RNs)

These networks are composed of adders whose purpose is to accumulate the different clusters of partial sums that are generated by the MN. Currently, we support the following RNs:

- **Reduction Tree (RT)** and **Augmented Reduction Tree (ART+DIST)** (Figure 2.8(a)): An ART integrates a tree-based topology built upon a reduction tree but augmented with one 3:1 adder unit per node for efficiently executing reduction operations. The tree structure is augmented with links between the nodes of the same level (horizontal links) that do not share the same parent. This augmented tree enables flexible support of multiple and non-blocking virtual reduction trees over a single physical tree hardware substrate [70]. More specifically, each node is a configurable Adder Switch (AS) that can be statically configured as either *2:1 ADD*, *3:1 ADD*, *1:1 ADD plus 1:1 forward*, or *2:2 forward*. This configurable capability within each ART node along with the augmented links are key aspects to enable high flexibility in MAERI.

- **ART + Accumulation Buffer (ART+ACC)** (Figure 2.8(b)): This RN is similar to ART, but allocates a set of accumulators at the output of the reduction network, allowing partial sums from consecutive iterations to be temporarily accumulated in the accumulators, and enabling them to be pipelined.

- **Forwarding Augmented Network (FAN)** (Figure 2.8(c)): As it is demonstrated in SIGMA [31], the ART topology is inefficient in terms of area and power due to the 3:1 adders. SIGMA proposed a more sophisticated RN called FAN, which equivalently to ART, allows to create any arbitrary number of dynamic-size clusters, but replaces the inefficient 3:1 adder switches by simpler 2:1 adders t the cost of more wires.

- **Linear Reduction Network (LRN)** (Figure 2.8(d)): In order to support all types of accelerators, in STONNE we also implement a linear reduction network which is typically used in rigid accelerators such as the TPU [61], Eyeriss [22], Eyeriss-v2 [24] or ShDianNao [28], to perform the cluster reductions.

Additionally, and as a part of this thesis, STONNE also implements **STIFT** and **MRN** reduction networks which are described as thesis proposals in Chapters 3 and 4, respectively.

2.4.2 Memory Hierarchy and Memory Controllers

STONNE implements the typical configurable memory hierarchy found in most DNN accelerators composed of local storage, some on-chip global storage (i.e., the Global Buffer, GB), and the off-chip DRAM memory. These three levels of the hierarchy are configurable by the user through the STONNE configuration file, which defines parameters such as bandwidth, different FIFO sizes, GB size or DRAM size and technology (e.g., HBM). Data orchestration between the GB and the distribution and reduction networks is performed by a memory controller (i.e., control unit) which is also selected by the user based on their preferences. As data movement differs depending on both the dataflow and whether the execution is dense or sparse, STONNE implements different types of memory controllers which are configurable and interact with DRAM memory assuming double-buffering prefetching at the Global Buffer. These memory controllers use internal counters to calculate the next addresses that the accelerator will read or write and their implementation is inspired by Buffets [100]. We described these memory controllers as follows:

- **Dense controller (DC)**: it takes inspiration from mRNA [125] and hence, orchestrates the data based on a fixed tile partition that cannot change during the execution of the layer (see Figure 2.8(j)). First, a DNN layer is defined with 7 parameters as *Layer(R, S, C, K, N, X′, Y′)* where R and S are the number of rows and columns in a filter respectively, C is the number of channels, K is the number of filters, G is the number of groups (i.e., to give support for factorized convolutions), N is the batch size, and X′ and Y′ are the number of rows and columns in the output respectively. We define a tile as *Tile(T_R, T_S, T_C, T_G, T_K, T_N, T_X′, T_Y′)*, where $T\_R \times T\_S \times T\_C$ parameters are a subset of the filter dimensions, and therefore, what defines the size of the dot product. Similarly, $T\_G \times T\_K \times T\_N \times T\_X' \times T\_Y'$ parameters represent the subset of number of groups, filters per group, input fmaps, and output dimensions, respectively, thus defining the number of clusters that are mapped onto the architecture. Note that, if the size of the cluster is smaller than the filter size (i.e., $(T\_R/R \times T\_S/S \times T\_C/C) > 1$), then the architecture will have to enable folding as it will be necessary to iterate over the same cluster to process the entire filter.

- **Sparse Controller (SC)**: The use of the sparse controller (Figure 2.8(k)) changes drastically the way in which the data flows throughout the elements of the architecture as when sparsity is enabled, the size of the dot products varies according to the sparsity of the data. The sparse controller implemented in STONNE runs GEMM operations (any CONV operation can be mapped to GEMM using the *img2col* function) and supports both bitmap and CSR formats to represent the sparsity of the MK and KN matrices.

Obviously, the configured memory controller must always be compatible with the hardware substrate selected to be modelled. In terms of dataflows, STONNE implements the dense weight-stationary, output-stationary and input-stationary dataflows. DRAM is modeled using DRAMsimv3 [32].

Other alternatives could also be easily implemented from the existing memory controllers. For example, in Chapter 4 we describe an additional memory controller incorporated in STONNE that supports the three combination of *Inner-Product*, *Outer-Product* and *Gustavson's* dataflows for sparse GEMM computation.

2.4.3 Modeling DNN Accelerators in STONNE

2.4.3.1 Cycle-level Simulation

Figure 2.9 shows the class diagram used in the STONNE Simulation Engine to model each component. As can be observed, all the components contain a `cycle()` method which implements their behaviour during a clock cycle. To enable the abstraction and allow the user to configure its own accelerator, we employ a hierarchical abstract class implementation whose specific instances are selected at runtime by the main `Accelerator` class. This top class iterates over every configured component in the accelerator and runs its `cycle()` method, emulating a cycle-by-cycle microarchitectural behaviour.

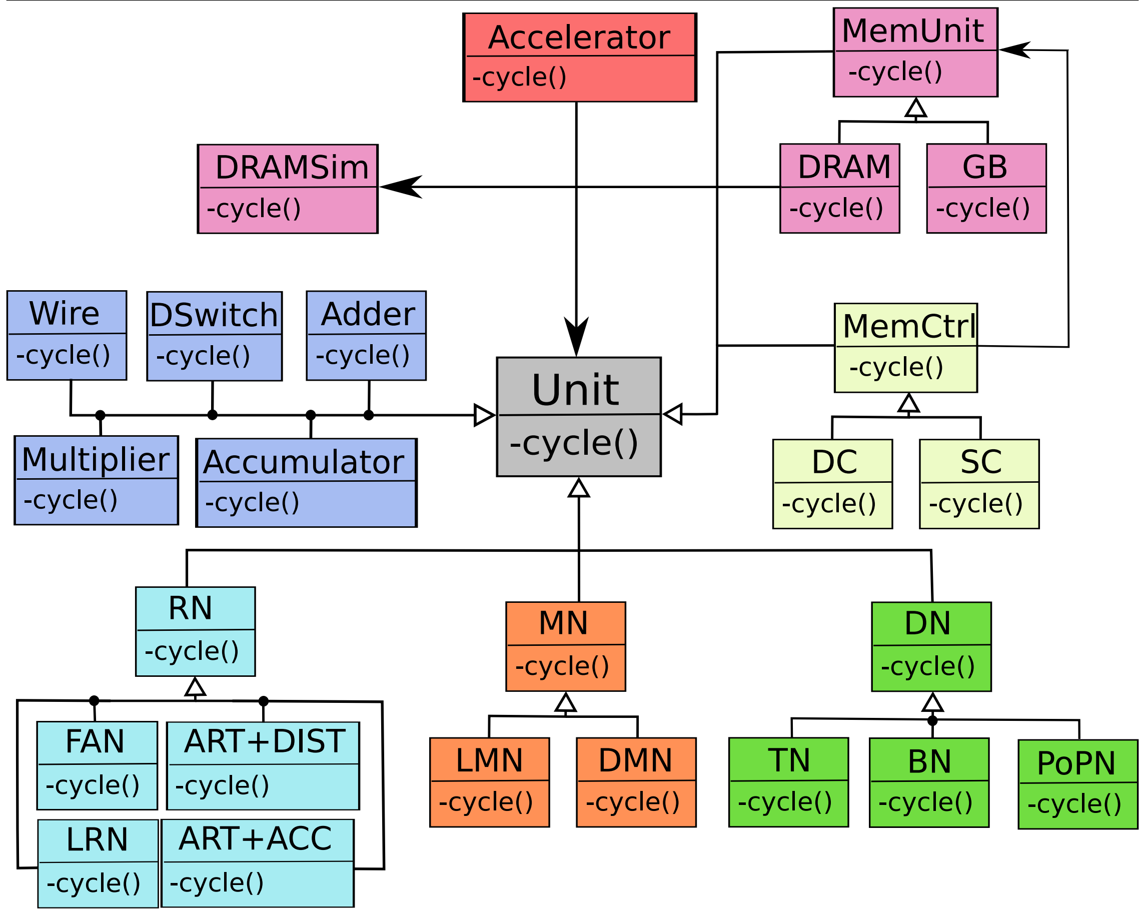


Figure 2.9: Simulation Engine class diagram. Acronyms from Figure 2.8.

2.4.3.2 Variability

Using the building blocks shown in Figure 2.8(a-k), STONNE is able to model a variety of DNN accelerator architectures. Examples of particular architectures directly supported in STONNE and the basic building blocks used in each case are given in Table 2.4 (also drawn in Figure 2.8(l-o)). Moreover, STONNE can be easily extended with additional models of DNs, MNs, RNs and memory controllers, giving rise to new accelerator architectures, as we demonstrate in Chapters 3 and 4.

Data-dependent optimizations: Since STONNE connects with DL frameworks, the aforementioned building blocks can be extended to precisely evaluate

	TPU-like	MAERI-like	SIGMA-like
Memory Controller	Dense	Dense	Sparse
Distribution Network	PoPN	TN	BN
Multiplier Network	LMN	LMN	DMN
Reduce Network	LRN	ART	FAN

Table 2.4: Modeling DNN Accelerators in STONNE.

data-dependent architectural optimizations. The last two use cases presented in Section 2.5 showcase this.

Limitations of STONNE: In essence, STONNE is aimed to model MAC-based accelerators. Modelling other types of accelerators (e.g., bit-wise or analog ones) could require major changes to the Simulation Platform component of STONNE.

2.5 Validation

This section validates STONNE by approaching three different angles: First, we perform a timing validation in which we compare the timing results (i.e., the number of cycles) obtained with STONNE against those obtained with the real hardware. After that, we ensure that the results obtained with STONNE are correct. Finally, we measure the accuracy of STONNE in terms of energy and area results.

2.5.1 Timing Validation

To validate the timing accuracy of STONNE against real hardware, we focus on three open-source implementations of DNN accelerators: the MAERI BSV code, the SIGMA Verilog code, and the TPU RTL implementation used to validate SCALE-Sim [30] implemented in Verilog. This helps us also validate the experimental results performed in our three use cases in Section 2.6.

For the timing validation process, we configure three instances of a MAERI-like, a SIGMA-like and an output stationary TPU-like architecture using their corresponding building blocks (see Figures 2.8(l), 2.8(n) and 2.8(o)). Recall that this execution mode allows for easy configuration of the Simulation Engine (to model a MAERI architecture in this case), DNN layer configuration and memory/compute partition tiles.

Since these RTL versions do not provide the large flexibility of our cycle-level architectural simulator–which can model any combination of the parameters

Design	Layer	M	N	K	RTL # cycles	STONNE # cycles	Error %
MAERI	MAERI-1	6	25	54	1338	1381	3.10%
	MAERI-2	20	25	180	16120	16081	0.24%
	MAERI-3	6	400	54	26178	26581	1.51%
SIGMA	SIGMA-1	64	128	32	2321	2304	0.73%
	SIGMA-2	256	64	64	8594	8448	1.72%
	SIGMA-3	256	128	64	17192	16896	1.75%
	SIGMA-4	128	1	64	139	138	0.72%
TPU	TPU-1	16	16	32	66	67	1.50%
	TPU-2	16	16	16	50	51	2.00%
	TPU-3	32	32	16	200	204	2.00%
	TPU-4	64	64	32	1056	1072	1.50%

Table 2.5: Timing accuracy of STONNE using RTL versions of MAERI, SIGMA and an OS-dataflow TPU.

of the accelerator (e.g., number of MSs, number of trees in the DN, number of input/output ports in the Global Buffer)–we are heavily constrained in the number of validation experiments that we can carry out. This way, for the MAERI-like architecture, we have configured both STONNE and BSV versions with 32 MSs and 4 DN/RN elements/cycle bandwidth parameters. In addition, the MAERI BSV version can only execute the three different types of layers listed in Table 2.5, with the tile shape: *Tile(T_R=3, T_S=3, T_C=1, T_G=1, T_K=1, T_N=1, T_X'=3, T_Y'=1)*. For the SIGMA-like version, we have configured both the RTL and STONNE versions with 128 MSs and 128 DN/RN elements/cycle bandwidth parameters running 4 layers. For the TPU-like, we have configured both the RTL model and STONNE using a 16×16 PE-array and full bandwidth. Given this set of microbenchmarks targeting specific layer types, we run STONNE using its direct user interface (the `STONNE User Interface` in Figure 2.4).

To evaluate the accuracy of timing simulation, Table 2.5 shows a comparison of the total number of executed cycles reported by the RTL versions and STONNE after running the eleven layers supported by the RTL versions. As we can see, the differences in the total number of executed cycles obtained with STONNE and the RTL versions range from 0.14% to 3.10% (1.53% on average), demonstrating that STONNE closely mimics the characteristics of the hardware versions.

2.5.2 Functional Validation

Since STONNE simulator is a back-end compute platform of PyTorch, it also outputs the result of the inference (the prediction) when running a particular DNN model for certain input data. To validate the functionality of STONNE, we have configured and run the three DNN accelerators presented in Table 2.4 with 256 processing elements and full bandwidth (i.e., 256 elements/cycle). We have executed the seven DNN models listed in Table 2.1 with a test set of 50 samples (e.g., an image or a sentence) from their respective datasets, and for every sample, we have compared the output of the last DNN layer (e.g., the score digits of a fully-connected layer) reported by PyTorch when running natively on the CPU, with the obtained for the executions with STONNE. They perfectly match for all cases.

2.5.3 Accuracy of Energy and Area Estimates

We ran synthesis using Synopsys Design-Compiler and place-and-route using Cadence Innovus on each module inside the TPU, MAERI and SIGMA RTL to obtain the real energy and area numbers. We used those numbers to derive the energy and area models implemented in STONNE. In this way we ensure the energy and area numbers reported by STONNE perfectly mimics the real hardware.

2.6 Examples of Use Cases of STONNE

Through three use cases, we demonstrate how STONNE can be used to conduct comprehensive evaluations of several DNN accelerator architectures running complete DNN models. For the three use cases we assume the next system parameters: 28-nm technology node, 1 GHz clock, FP8 datatype, 108-KB Global Buffer (GB) size and two 256 GB/s 512-MB HBM2 DRAM modules. Other relevant parameters that are specific to each use case are given below.

2.6.1 Evaluation of DNN Inference in TPU, MAERI and SIGMA

The aim of the first use case is to directly compare three different accelerator architectures (namely, TPU, MAERI and SIGMA) considering their achievable performance, energy consumption and required area. All the simulations were

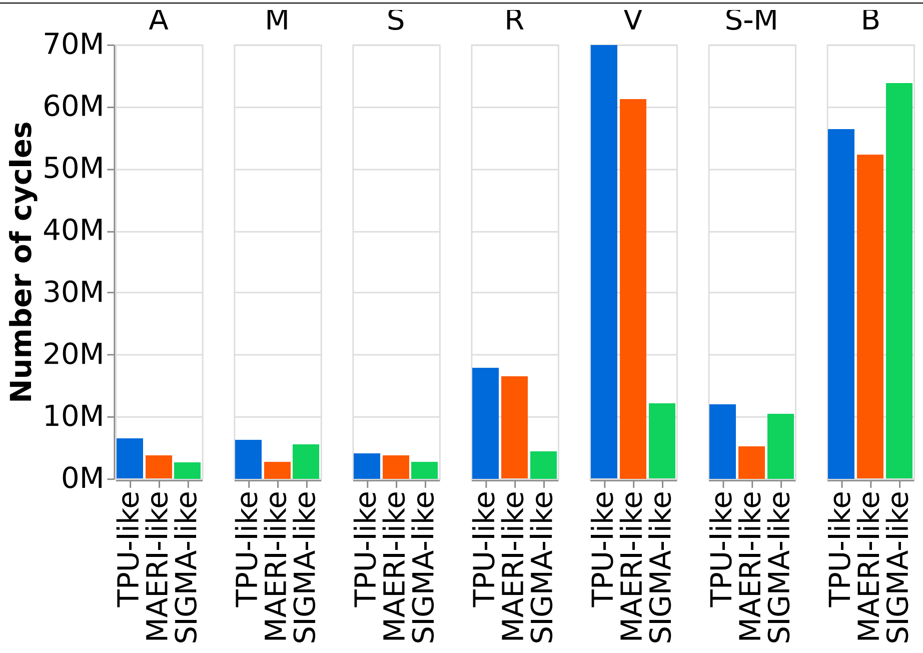


Figure 2.10: Number of cycles reported by STONNE after running the inference procedure of the DNN models listed in Table 2.1 on MAERI, SIGMA and TPU.

performed considering the complete inference processing of the 7 DNN models presented in Table 2.1.

2.6.1.1 Methodology and Configuration Parameters

We assume the next system parameters for the three architectures: For both MAERI and SIGMA, we assume 256 multipliers and adders, and 128 elements / cycle GB read/write bandwidth. For the TPU, we have configured 256 processing elements and full bandwidth (as this architecture requires). Note that, configuring these three architectures in STONNE does not require any modifications in the simulation framework, as STONNE directly supports the required hardware modules and dataflows for all of them (see Table 2.4). To configure STONNE, we have used the hardware input file of STONNE to simulate the native building blocks shown in the Table 2.4.

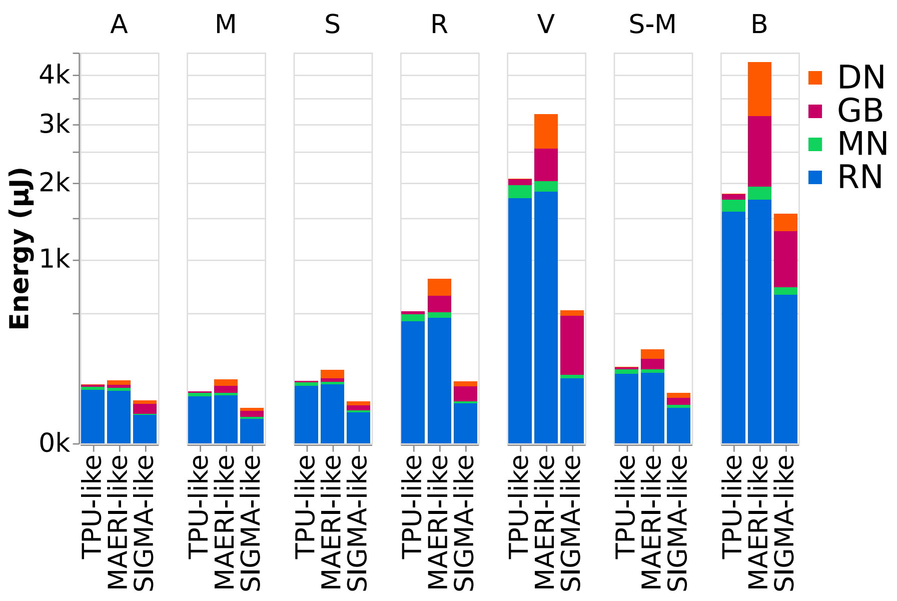


Figure 2.11: Energy consumption (we use an *sqrt* log y-axis) in μj (b) reported by STONNE after running the inference procedure of the DNN models listed in Table 2.1 on MAERI, SIGMA and TPU.

2.6.1.2 Results

Figure 2.10 shows the number of cycles obtained for the three simulated architectures. We observe that a MAERI-like architecture reaches average performance improvement of 20% over the TPU-like architecture for the execution of the seven DNN models, with a maximum of 231% for Mobilenets and a minimum of 9% for Resnets-50. Besides, we found that a SIGMA-like architecture is 91% faster on average than a MAERI-like one thanks to the sparsity support.

Figure 2.11 shows a breakdown of the total amount of energy consumed (μJ) in each case, distinguishing the main architectural components: Global Buffer (GB), Multiplier Network (MN), Distribution Network (DN) and Reduction Network (RN). As we can appreciate, the energy consumption is dominated by the RN as it reaches 84%, 58% and 43% of the total energy on average across the DNN models for the TPU-like, MAERI-like and SIGMA-like architectures, respectively. In general, STONNE finds that the SIGMA-like architecture is 70% and 54% more energy efficient than the MAERI-like and TPU-like architectures, respectively. This is due to the capacity of SIGMA to exploit sparsity, which reduces the number of operations by 77%, thus bringing significant dynamic energy savings.

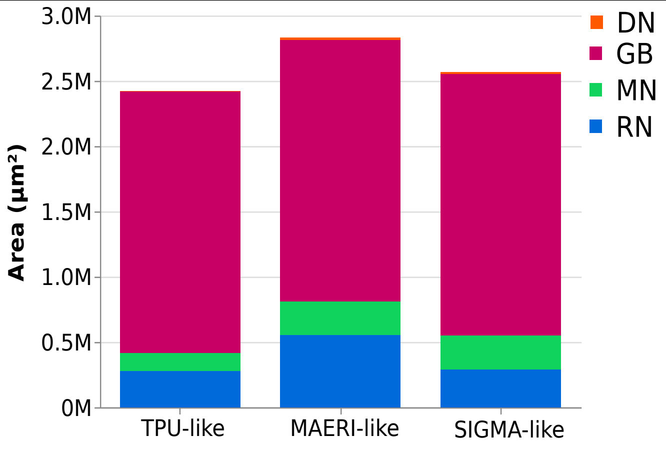


Figure 2.12: Area estimations in μm^2 for MAERI, SIGMA and TPU obtained with STONNE.

Finally, in terms of area (see Figure 2.12), the SIGMA-like architecture is 13% more efficient than the MAERI-like one, while the TPU-like architecture is 17% and 6% more efficient with respect to the MAERI-like and SIGMA-like architectures, respectively. As we observe, the differences in area are not as noticeable as those in the energy and runtime metrics. This is due to the area required in the three cases is mainly dominated by the SRAM structure of the GB, which is the same for the three architectures, and that represents 70%, 77% and 82% of the total area of the MAERI-like, SIGMA-like and TPU-like architectures, respectively.

These results are consistent with the trends pointed in prior works [31,70] and validate that flexible architectures can adapt much better to the current diversity of DNN layers.

2.6.2 Back-End Extension for Data-Dependent HW Optimizations

Data-dependent optimizations aim to reduce computation and memory footprint during DNN inference by exploiting the characteristics of the data values being used, such as exploiting input data repetitions or zero skipping. These techniques, yet powerful, are very challenging to evaluate, as it is necessary to have the exact data values that the hardware accelerator processes at all times during the execution of a real DNN model. This second use case demonstrates how STONNE makes this possible by acting as an accelerator device for a real DNN framework (like Pytorch) and by enabling cycle-level modeling of the accelerator architecture activity under consideration. Furthermore, this use case also illustrates how STONNE can be easily extended to model architectures that are not originally included such as the TPU, MAERI and SIGMA.

To do so, we will use STONNE to model the data-dependent accelerator SnaPEA [33] by extending its back-end (i.e., the simulation platform described in Section 2.4). This architecture that aims to optimized CNN processing, exploits a property in which all the activation values in the convolution operations are either zero or positive. Any negative value calculated during the convolution is directly converted into zero by the subsequent ReLU operation. This means that the weights can be statically reordered based on their signs so that the architecture can perform at runtime a single-bit sign check on the partial sum. Once the partial sum drops to zero, the rest of computations and memory accesses can be avoided, since the output value will unfailingly be zero.

2.6.2.1 Implementation

The changes that have been introduced in STONNE to model this architecture mainly affect its back-end, and are as follows:

1. Inclusion of a prior-simulation function in the input module (see Figure 2.4a) to reorder the weights as explained in SnaPEA [33] and that creates a table of indexes to locate the inputs. This table is passed to the memory controller (i.e., control unit) which will use it to match every sorted weight with its activation.

2. A new memory controller (i.e., Control Unit) in the Simulation Engine (see Figure 2.4a) that utilizes this table of indexes to correctly deliver the weights and inputs to the multipliers. This unit is just an extension of the previous dense memory controller already provided in STONNE and explained in Section 2.4.2.

3. We use the current linear multiplier network (see MNs Section 2.4.1.2) configured to use the output-stationary dataflow.

4. We have extended the accumulation logic in the processing units to detect when the results are negative. As soon as this event is triggered, the data is sent out to the Global Buffer, cutting out the computation earlier, and thus, saving energy and time. We implement the *exact mode* explained in SnaPEA.

5. To estimate energy consumption, we have included in the Output Module a new table with the energy model of SnaPEA based on the published energy numbers provided in the SnaPEA paper.

2.6.2.2 Methodology and Configuration Parameters

Similar to the SnaPEA work [33], for this use case we model 64 multipliers and adders, and 64 elements/cycle GB read/write bandwidth. We have configured two different versions of our SnaPEA implementation: the Baseline, which models the SnaPEA architecture but that excludes the negative detection logic, and therefore, runs the entire execution; and the full SnaPEA architecture (we call it SnaPEA-like) which adds this logic, cutting out the computation earlier whenever possible. We have configured and run these two versions of the accelerator with the aforementioned parameters and we have executed the four purely CNN models of those listed in Table 2.1 (i.e., Alexnet, Squeezenet, VGG16, and Resnets-50) with a set of 20 input images extracted from ILSVRC-2012 validation dataset. For each input image, we have compared the output of the last DNN layer (e.g., the score digits of a fully-connected layer) reported by PyTorch when running only on a CPU, with the one obtained for the executions with STONNE simulating the two SnaPEA implementations to corroborate that they perfectly match.

2.6.2.3 Results

Figure 2.13 plots the speedups achieved by the SnaPEA-like architecture against the baseline for the considered four CNN models. STONNE shows that SnaPEA can bring average speedups of 35%, closely approaching the 30% originally reported in [33]. On the other hand, Figure 2.14 shows the energy consumed when running the benchmarks on the two architectures. The results are normalized to the baseline. Similarly, these numbers demonstrate that the speedups mentioned previously translate into significant energy savings (21% on average).

These results can be explained by observing Figures 2.15 and 2.16, which show the number of operations and memory accesses performed during the execution of the CNN models on both the SnaPEA-like architecture and the baseline. In particular, we can observe that on average the technique exploited by SnaPEA is able to reduce the number of computations and memory accesses by 30% and 16%, respectively, being Squeezenet (S) the CNN model with the highest reductions (30% in operations and 22% in memory accesses) which correlates with the highest improvements in energy consumption. As it can be

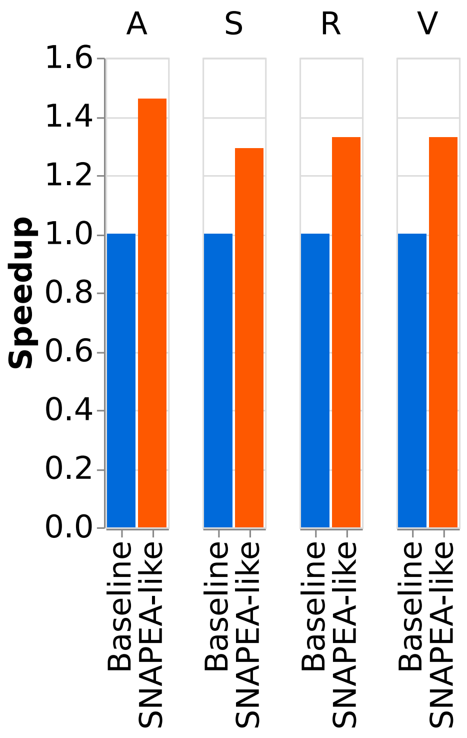


Figure 2.13: Speedups of SnaPEA against the baseline after running four CNN models using the STONNE simulator.

appreciated, these results closely follow the trend reported in [33], confirming that SnaPEA is a promising optimization to be applied to CNN accelerators. Obviously, we found timing and energy differences between the original paper and the results obtained with STONNE. These differences mainly stem from slight variances in the methodologies used in each case: older CNN models are used in [33], potential differences in the weights of the CNN models are possible (the SnaPEA paper does not specify how the weights of the CNN models have been obtained), and presumably, different images are used as inputs in both cases. More importantly, however, is that we see the same order of magnitude in the gains that SnaPEA is able to achieve, demonstrating STONNE's ability to quickly and faithfully quantify the benefits of applying data-dependent optimizations to DNN accelerators.

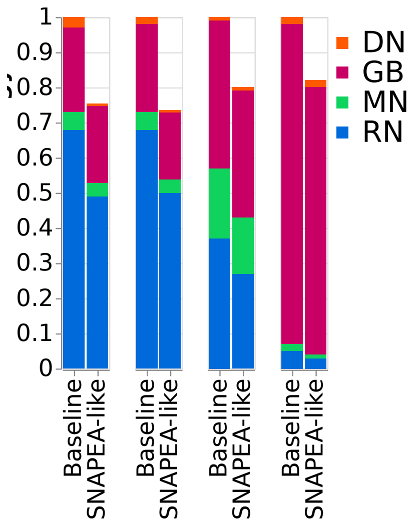


Figure 2.14: Normalized energy of SnaPEA against the baseline after running four CNN models using the STONNE simulator.

2.6.3 Front-End Extension for Filter Scheduling in Sparse Accelerators

Through the third use case, we demonstrate that precise, full-model evaluation is required to expose the particular values used during inference. This is needed for some optimization techniques such as filter scheduling in flexible sparse DNN accelerators that we present here. The modifications now focus on the front-end of the simulator.

2.6.3.1 Motivation and Idea

In the Chapter 3 we will confirm the importance of the folding strategy implemented in a dense architecture (such as MAERI) due to the large size of the filters that are required to compute a certain output (i.e., large dot products). When we consider the large amount of sparsity in the filters of contemporary trained DNN models (from 60% to 90%, as shown in Table 2.1), the amount of computation involving a certain filter can be largely reduced by only mapping the non-zero weights onto the accelerator's processing elements. As a result, even more than one entire filter could be mapped at once onto different clusters of MSs in the MN of a flexible sparse DNN accelerator. In addition to this well-known optimization, in this use case, we demonstrate for the first time that

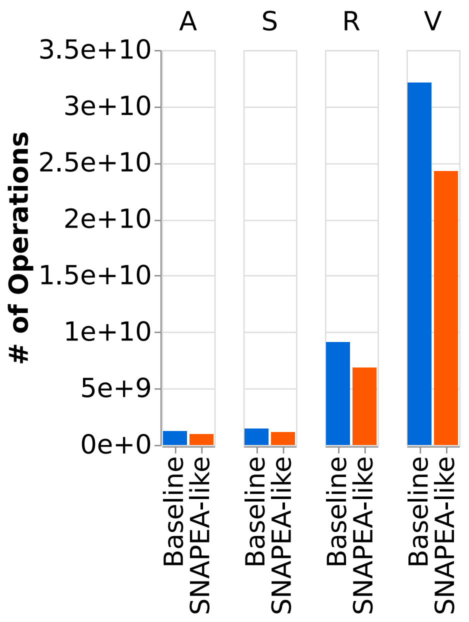


Figure 2.15: Number of computed operations during the execution of the four CNN models using the STONNE simulator for SnaPEA and the baseline.

the way in which the filters of a sparse DNN model are scheduled onto a MN network of a flexible DNN inference accelerator might have significant impact on performance. A prior work [88] examined this idea, but with a focus on GPUs as the research community lacked a simulation tool for DNN accelerators. Here, we prove that this reordering has also significant impact on DNN accelerators. To do so, we use STONNE to simulate a flexible and sparse 256-MS SIGMA-like architecture [31].

First, to analyze scheduling opportunities of variable size filters onto the SIGMA's MN fabric, we pay attention to the diversity of filter sizes for the seven DNN models under study when sparsity is exploited. Particularly, Figure 2.17a shows the average number of entire filters that could be mapped simultaneously onto a 256-MS flexible architecture for every single DNN layer depending on each DNN model. As can observed, between 4 and 8 filters can be entirely mapped simultaneously in most cases. The only exceptions are Alexnet and BERT, that features larger filter dimensions by design (e.g., up to 4.3× larger filters compared to Mobilenets-V1, which comes next in terms of filter size). Moreover, we have further looked into the size of the filters required to compute each of the DNN layers, finding out huge variability between them. As an illustrative example,

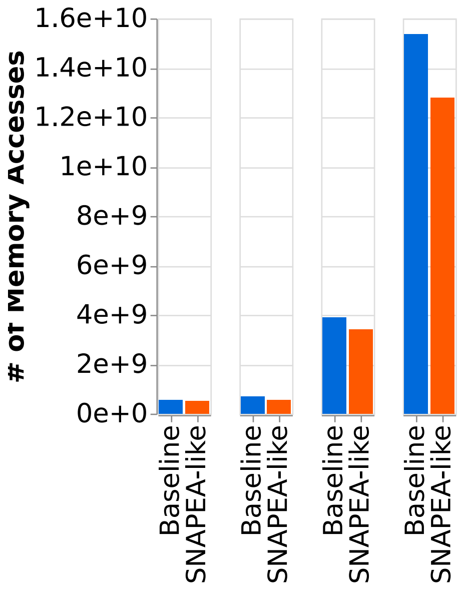


Figure 2.16: Number of performed memory accesses during the execution of the four CNN models using the STONNE simulator for SnaPEA and the baseline.

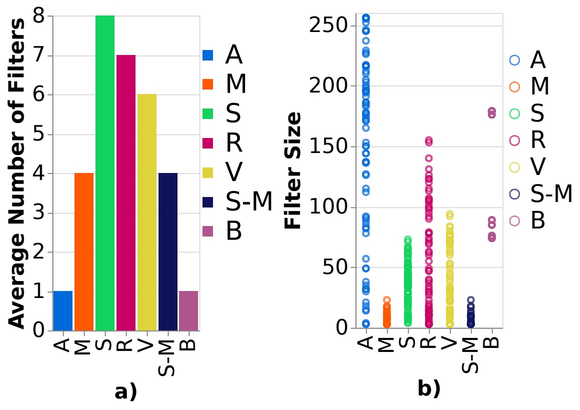


Figure 2.17: (a) Average number of entire filters that can be mapped simultaneously in a 256-MS flexible sparse architecture. (b) Filter sizes for the first layer of the DNN models.

Figure 2.17b shows the size (*y-axis*) for every mapped filter onto the 256-MS architecture for the first layer of each DNN model[4] (*x-axis*).

[4]The maximum mapping size is 256 because of the 256-MS SIGMA architecture.

Next, we show that the large filter size variability found in contemporary DNN models can be exploited to optimize the DNN inference procedure. In particular, we observe that the specific order in which the filters are scheduled to be mapped onto the DNN accelerator can impact the overall compute utilization, and in consequence, overall performance. Figure 2.18 illustrates this optimization opportunity for an example 8-MS SIGMA-like architecture. At the top of the figure, we consider an example layer composed of a 1×5 vector of inputs and four 1×5 sparse filters (F_0 and F_2 have an effective size of 4, while it is 2 in the case of F_1 and F_3) utilized to compute four dot products (producing the outputs from O_0 to O_3). Figure 2.18a shows that computing this layer in SIGMA completely ignoring the opportunities of scheduling the filters can lead to an unbalanced scenario, requiring a total of 4 cycles. More specifically, in the first step, the sparse controller maps both F_0 and F_1 filters onto the MN. Then, since F_2 does not fit entirely within the MN, the sparse controller cannot allocate it and F_2 has to wait for execution until the next iteration. Therefore, in the first iteration, the dot product to calculate O_1 can be completed in just 1 cycle, while the one to calculate O_0 needs 2 cycles, clearly unbalancing the computation. This happens again in the second iteration, where the next two remaining filters are computed after 2 additional clock cycles (i.e., 4 cycles overall). As we can see, we show how changing the order of the filters (which could be done either statically by the compiler or dynamically by the accelerator's memory controller) yields a variable number of cycles to complete the computation of the four dot products in the example.

Figure 2.18a shows that computing this layer completely ignoring the opportunities of scheduling the filters can lead to an unbalanced scenario, requiring a total of 4 cycles. On the other hand, Figure 2.18b illustrates how faster layer processing can be achieved if the computation of the four filters is scheduled differently. For instance, we can consider a simple scheduling heuristic to achieve perfect load-balancing in this example for computing the dot products. In particular, we can rearrange the filters to be mapped at every iteration depending on filter size following a Largest Filter First policy as follows (more details next). In the first iteration, F_0 and F_2 with 4 non-zero values can be mapped together, thus taking 2 cycles to compute their two outputs. Then, in the next iteration, the 2-size filters (F_1 and F_3) can also be mapped together, but in this case the computation of their associated outputs would be completed in just 1 cycle. Therefore, after applying this simple reordering of the filters, we can balance the computation of the outputs and make better use of the accelerator resources, which in the end results in fewer clock cycles required (25% less in this simple example).

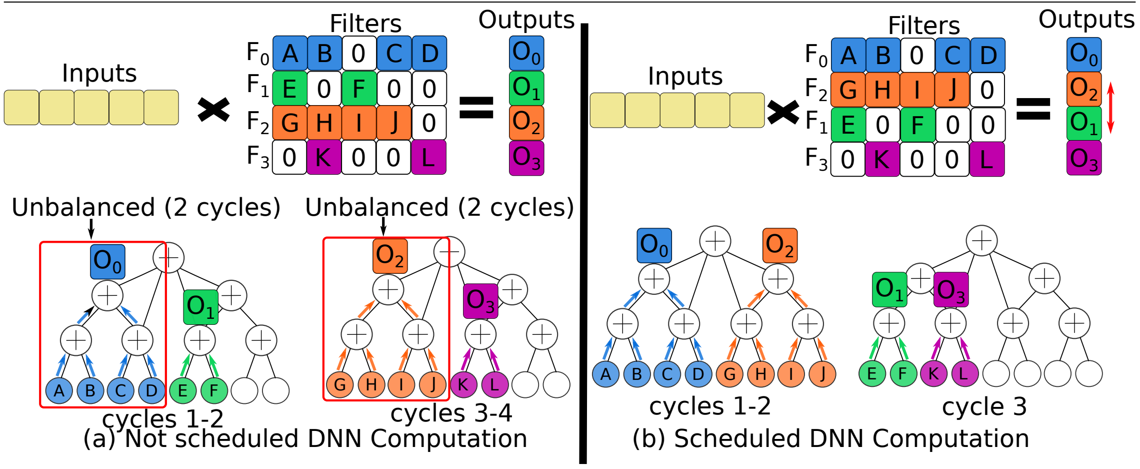


Figure 2.18: An example filter scheduling heuristic to optimize four dot products in a SIGMA-like accelerator architecture.

This observation opens up a new avenue to optimize the inference procedure of sparse DNN models in flexible sparse DNN accelerators. In particular, the exploration of novel scheduling strategies for the sparse filters to better balance the mapping of the dot products onto the DNN accelerator, so that compute unit utilization is maximized and processing time reduced. We focus on exploring static scheduling heuristics, where the sparse memory controller issues the filters for execution in the same order that is determined by a certain static scheduling strategy directly applied layer by layer to each of the DNN models. To guarantee the correctness of the executions, a final reordering step is carried out after the last fully-connected layer of each DNN model.

2.6.3.2 Implementation

Implementing new scheduling approaches in STONNE just requires modifications in the front-end (i.e., Input Module) of the simulator. To do so, we have incorporated a prior-simulation function that reorders the filters based on its size and on the scheduling technique.

2.6.3.3 Methodology and Configuration Parameters

We model and simulate a single Flexible Dot Product Engine (Flex-DPE) in a SIGMA-like flexible architecture that supports sparse and irregular matrix formats based on weights and/or activation sparsity. The on-chip network choices are

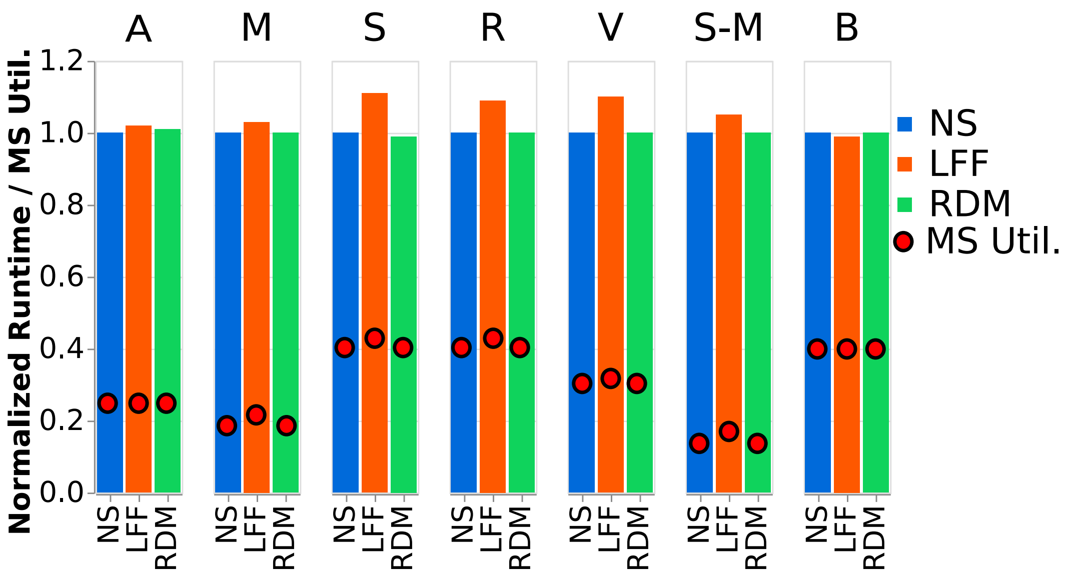


Figure 2.19: Normalized runtime for the LFF static scheduling strategy with respect to a non-scheduled execution.

shown in Table 2.4. We have run the seven DNN models whose sparsity levels are shown in Table 2.1. We model these system parameters: 256 multipliers and adders and 128 elements/cycle Global Buffer (GB) read/write bandwidth.

In this use case, we consider a simple static heuristic: *Largest Filter First (LFF)*. In LFF, the filters are reordered so that the sparse controller always selects the largest available filter (i.e., of those not yet used for computation) that can be mapped onto the Multiplier Network (256 MSs in this case). To cover the rest of the available MSs, the scheduler selects as many available filters as possible in descending size order. For the sake of completeness, we also present the results obtained for a *Random (RDM)* ordering.

2.6.3.4 Results

We compare the results for LFF and RDM against those obtained when running the sparse filters in their natural order (i.e., without performing any kind of reordering). We call this approach the *No Scheduling (NS)* ordering. As observed in Figure 2.19, using the random scheduling strategy does not yield any performance improvement as the MS utilization does not increase at all. This demonstrates that a naive strategy is not good enough to better balance the prcoessing of the clusters. Alternatively, LFF is capable of both balancing the

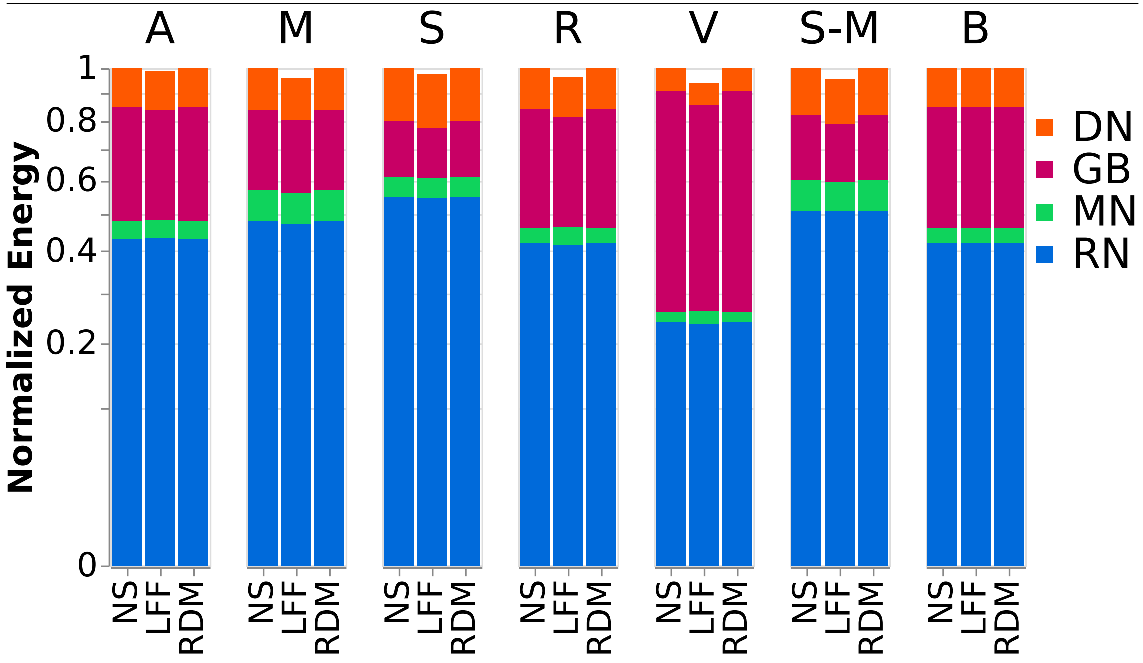


Figure 2.20: Normalized energy for the LFF static scheduling strategy with respect to a non-scheduled execution.

processing of the clusters during most of the execution and selecting a smaller filter when another one does not fit. This leads to increased MS utilization (2.5% on average), which translates into performance advantages ranging between 11% for the most sensitive DNN models (Squeezenet, VGG-16, Resnets-50 and ssd-Mobilenets) and 1% in models such as BERT, whose large filter sizes and low sparsity ratio (60%) often prevent multiple clusters from being processed simultaneously (see Figure 2.17a). On average, we observe a performance gain of 7% across the seven DNN models.

Figure 2.20 plots a breakdown of the total amount of energy consumed in each case, distinguishing between the main components of the architecture: Global Buffer, Multiplier Network, Distribution Network and Reduction Network. As we can see, energy reductions are not very significant, ranging between 1% and 6% (4% on average). The energy consumption in this case is mainly dominated by the number of operations carried out during the execution, which is the same regardless of how the filters are rearranged. In this case, most of the observed energy gains come from both the reduction of static energy due to the decrease in execution time, and the reduction in the number of messages to be sent through

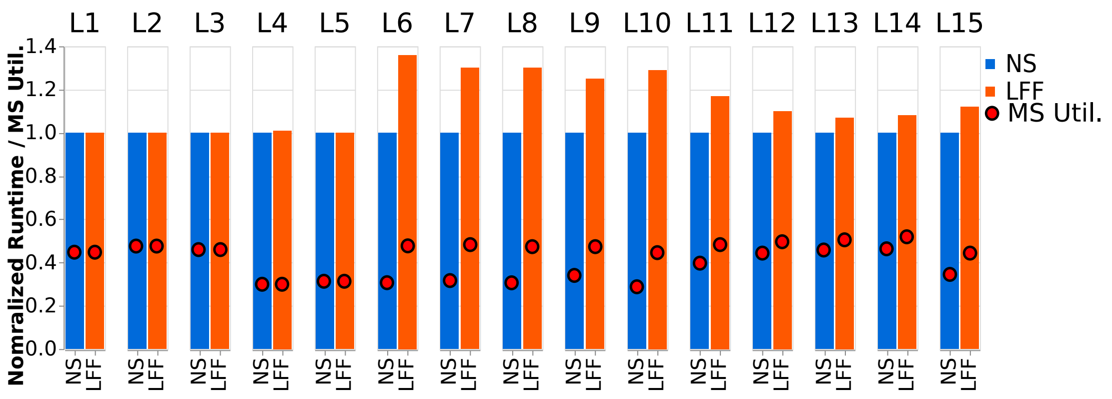


Figure 2.21: Normalized energy for the LFF static scheduling strategy with respect to a non-scheduled execution.

the DN (i.e., running more clusters simultaneously increases the ability to exploit the DN's multicast package delivery).

Another important observation that we would like to make is that we have found out a huge difference in terms of layers sensitivity when it comes to LFF. In particular, there are some layers that experience a large impact in terms of energy and performance when LFF is applied, but this benefit is subsequently hidden by other low-sensitive layers.

As an illustrative example, Figure 2.21 shows the performance and energy gains for 14 representative layers (in terms of LFF sensitivity) of Resnets-50 when using the LFF static filter scheduling heuristic. The results are normalized to the obtained for the non-scheduled execution. Here, we can see how the layers can be divided into three different groups according to their sensitivity to the LFF scheduling heuristic. The first five layers show no benefit at all as LFF is not capable to leverage the MSs better (gains of 0.01% on MS utilization), so they fall into the *low-sensitive* layer category. Contrarily, for the next five, significant improvements (up to 36% and 16% performance and energy gains, respectively, in layer *L6*) are observed, and would therefore be those that make up the *high-sensitive* layer category. These significant gains are explained by increased MS utilization, which ranges from 9% to 13% (11% on average). Finally, the *medium-sensitive* layer category would be comprised of the last five layers, for which MS utilization benefits varying from 8% to 4% (5% on average) are obtained, leading to lower performance advantages between 17% and 8% (energy benefits range between 5% and 1% in this case).

The obtained results point out that more intelligent heuristics capable of adapting the filter scheduling strategy to the specific features of each layer could bring large benefits in terms of performance and energy savings when running sparse DNN models. This observation paves the way for the development of much more sophisticated strategies aimed to improve the energy efficiency of next-generation DNN accelerators.

2.7 External Tools based on STONNE

Although other domains such as GNNs or multi-heterogeneous systems are out of the scope of this thesis, it is important to put emphasis on the impact the STONNE simulator has had on the development of other tools, as well as the impact that it might have in the future. For that reason, this section describes OMEGA and SST-STONNE, two additional tools developed upon the STONNE framework, which illustrates how STONNE enriches the research of not only the DNN application domain, but also multitude of other areas of domain-specific acceleration.

2.7.1 OMEGA

GNNs are becoming increasingly popular because of their ability to accurately learn representations from graph structured data. GNN inference runtime is dominated by two phases [121]: (1) *Aggregation* which is an SpMM (i.e., sparse-dense matrix multiplication) computation with irregular, workload dependent data accesses, and (2) *Combination* which involves computations that can be cast as GEMMs. Prior works on DNN dataflow studies have described the data orchestration and data movement in DNN accelerators (see Chapter 1). However, these works only model dense computations and model one GEMM or convolution operation at a time. GNNs offer an additional knob of pipelining between the two phases, which also leads to interdependence of the two dataflows.

The arrival of STONNE, has allowed to further study these dataflows. To do so, OMEGA framework [40] (**O**bserving **M**apping **E**fficiency over **G**NN **A**ccelerators) is built on top of STONNE [82] which enables to model the cost of the pipelined GNN dataflows. OMEGA instantiates two instances of the STONNE's simulation platform (see Section 2.3.1) and loads both SpMM and GEMM operations using the STONNE API operations *ConfigureSpMM* and *ConfigureDMM*, respetively (further details of these operations may be found in Section 2.4). Then, the STONNE's simulation platform feeds the statistics to an inter-phase cost model

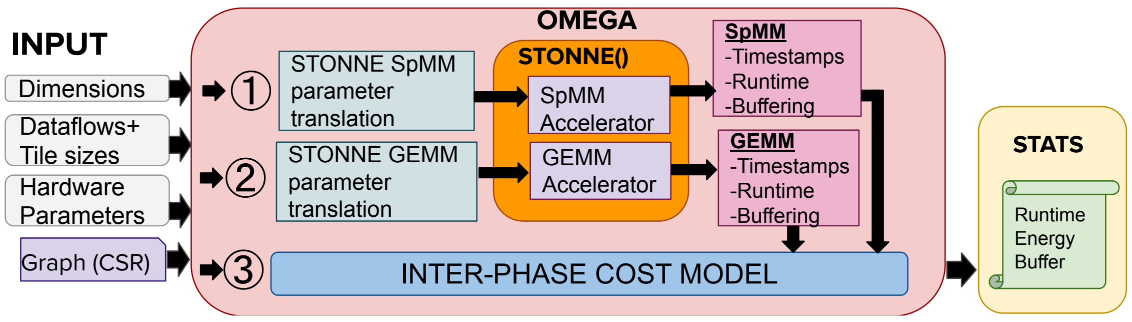


Figure 2.22: OMEGA framework toolflow.

(i.e., a specific output module for OMEGA) that returns the metrics of a pipelined inter-phase dataflow as shown in Figure 2.22.

OMEGA framework aims to provide analysis of the design-space of GNN dataflows over flexible accelerator [68] which captures both individual phase dataflows (Intra-phase dataflows) and dataflows between the two phases (Inter-phase dataflows).

OMEGA framework has gained a lot of interest by the research community and it is publicly available under the terms of the MIT License on [4]. Currently, OMEGA has been integrated within the STONNE ecosystem.

2.7.2 SST-STONNE

STONNE and OMEGA are tools that operate in standalone manner, which impedes them from simulating interactions with other simulated architectures such as CPUs, GPUs or other type of accelerators. To overcome this, we have also integrated STONNE into the Structural Simulation Toolkit (SST) infrastructure [9] (referred to as *SST-STONNE*). Figure 2.23 shows a high level overview of this integration. As we can see, SST-STONNE enables the modeling of both DNN and GNN accelerators within a larger HPC computing cluster, expanding the edge of the exploration to borders such as multi-tenant applications running on heterogeneous systems, scheduling techniques across different workloads or tasks within the same workload, or even host-device interaction. Furthermore, the SST-STONNE replaces the native STONNE memory controllers (explained in Section 2.4.2) by custom controllers that connects STONNE to a cycle-level memory hierarchy simulator (*SST memHierarchy*) able to accurately model configurable memory hierarchies. In this way, the STONNE simulator sends the memory requests to the SST memHierarchy simulator, which processes them

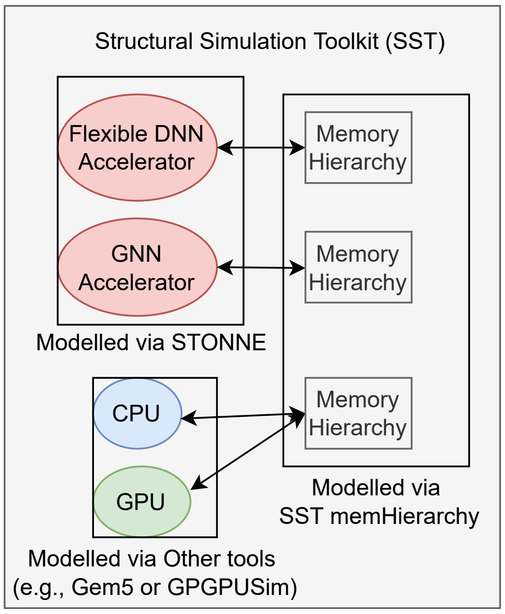

Structural Simulation Toolkit (SST)
Flexible DNN Accelerator
GNN Accelerator
Modelled via STONNE
Memory Hierarchy
Memory Hierarchy
Memory Hierarchy
Modelled via SST memHierarchy
CPU
GPU
Modelled via Other tools (e.g., Gem5 or GPGPUSim)

Figure 2.23: High level overview of SST-STONNE.

and sends back the responses. SST-STONNE supports four key tensor operations used in the DL domain: Convolution, GEMM, SpMSpM, and SpMM. In both SpMSpM and SpMM operations, we support the compression formats CSR, CSC and bitmap.

SST-STONNE has become the first simulator to integrate the three flavours of kernel computations: CPU, GPU and domain-specific accelerators. The source code is available on Github [7] and the tool has even been announced as an official external component of the SST framework [9]. In Chapter 4 we will utilize SST-STONNE to accurately model our sparse accelerator, demonstrating the impact that this tool may have in the future of the domain-specific acceleration.

2.8 Conclusions

STONNE paves the way towards rapid and accurate prototyping of next-generation DNN accelerator architectures. Through three use cases, we demonstrate the huge potential of STONNE to assist the research community in the pursuit of better DNN accelerator architectures. In the first use case we demonstrate how the STONNE simulator can be utilized to model both rigid and flexible accelerators by modeling and comparing state-of-the-art accelerators such as the Google's TPU, MAERI, or SIGMA. In the second use case, we model SnaPEA accelerator to show the importance of having a cycle-level simulator in order to model data-dependent architectures. Finally, in the third use case, we propose a novel filter scheduling technique that verifies the importance of computing real values when modeling. Furthermore, the proposed technique can be leveraged to improve the energy efficiency of current state-of-the-art sparse accelerators.

Finally, we also demonstrate the extensibility of STONNE by presenting two external tools called OMEGA and SST-STONNE. Both tools build on top of STONNE to support more complex application domains, such as GNN hardware acceleration, or even, heterogeneous systems. In this chapter, we demonstrate that as the complexity of the microarchitecture of DNN accelerators grows, the analytical models typically used to estimate their performance and energy figures are not able to capture many important subtleties that simulation at cycle level does.

Finally, STONNE simulator has become in two years one of the most popular and utilized tools on the research of DNN accelerators. The interest can be observed on [8] as well as among the researchers around the world that are leveraging STONNE as the heart of their research projects.

Chapter 3

STIFT: A novel network fabric for efficient spatio-temporal reduction in flexible DNN accelerators

3.1 Introduction and Motivation

As we have discussed in chapter 1, the computation of the dot products entailed by the DNN inference phase requires the execution of simple Multiply-and-Accumulation operations (i.e. MACs), albeit in a very large quantity (e.g. 15.5 billion MACs approximately for the VGG16 DNN model). In order to fulfill the performance and energy consumption requirements of the inference procedure, the research community has embarked on the mass development of specialized DNN accelerator architectures [22,24,28,31,61,70,78]. These proposals are usually built using a large number of connected multiplier and adder units. The adder units massively reduce the large number of partial sums (psums) generated by the multiplication operations.

Prior to execution of the inference procedure on a DNN accelerator, a mapper [23,125] (see Figure 3.1) selects, based on the hardware capabilities and the specific characteristics of the workload (e.g., type and dimensions of the DNN layers), the number of dot products that will be calculated in parallel, and the group of multipliers and adders in charge of computing each dot product. In this chapter, we will also use the term *cluster* to refer to a group of multiplier units in charge of a certain dot product. Usually, the number of multipliers in the cluster

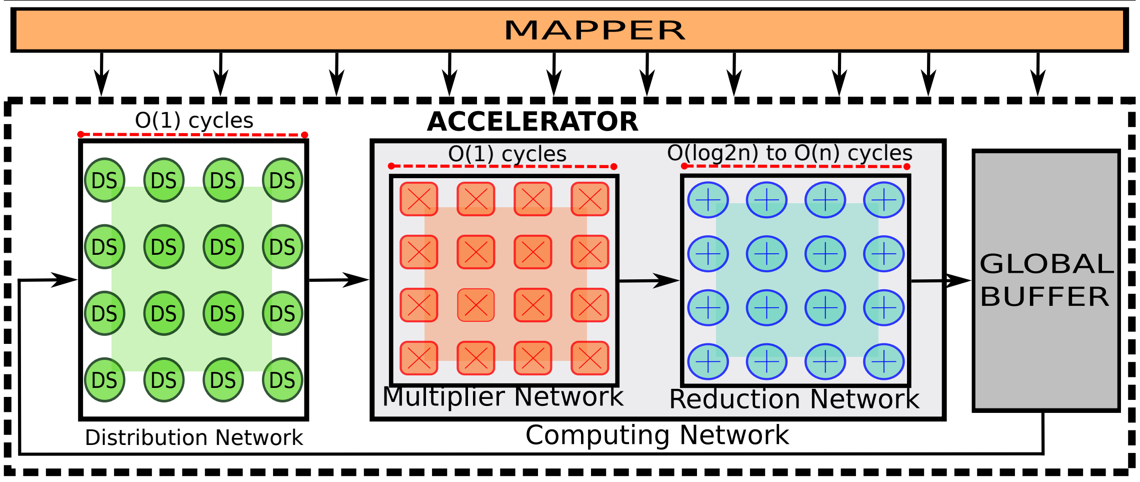


Figure 3.1: Conceptual view of a typical DL accelerator.

is smaller than the number of products required for the assigned dot product, so that several iterations are required. The result of each iteration is accumulated with that of the subsequent one, and so on, until the full dot product is completed. This process is known as *folding* [23], as a large dot product operation *folds* over a smaller cluster of multipliers [23].

Once the hardware configuration is determined by the mapper, the processing of each of the clusters in most recent proposals usually relies on a hardware implementing a 3-stage pipeline based on three network fabrics [66] (see Figure 3.1): the Distribution Network (DN), a global network that connects the global scratchpad SRAM (i.e., the Global Buffer) to the multipliers, is in charge of distributing all the operands for each cluster; the Multiplier Network (MN), a local network between the multipliers that enables reuse of weights (or inputs), performs the multiplication operations; and the Reduction Network (RN), a global network connecting the MN and the Global Buffer, carries out all the accumulations needed for the reduction phase. The MN and RN contain all the compute units (multipliers and adders, respectively) and along with the DN, define all the permitted mappings, and thus, determine the *dataflow* [68, 117] and the energy efficiency of the execution. These NoC-based designs have been shown capable of reducing the expensive off-chip memory accesses, and thus, to improve performance and energy consumption [18].

The separation between the DN, MN and RN can be physical (e.g. MAERI [70], SIGMA [31]) or logical (e.g., Eyerissv2 [24], Google's TPU [61]). In this chapter, we focus on the first type of accelerator architectures (*flexible* DNN accelerators), as they have been shown to provide increased flexibility and better performance than the second type (*rigid* accelerators) [31,70].

The DN, MN and RN in these flexible accelerator designs are typically implemented using lightweight (energy- and area-efficient) microswitches [19,69], which enable independent reconfiguration of each on-chip network in order to efficiently map different dot product partitions through the creation of dynamic-size clusters enabling single-cycle traversals between the transmitter (i.e., the on-chip global buffers) and the receiver (i.e., the multipliers). Different kinds of network topologies have been proposed for them. For example, tree-based [24,70], Benes-based [11,17,31,73] or linear-based [22,28,61] topologies for the DN, enabling single-cycle traversals between the transmitter (i.e., the Global Buffer) and the receiver (i.e., the multipliers), 1D-linear [31,70] or 2D-linear [22,24,61] topology for the MN, and customized tree-based topologies for the RN [31,70]. The DN, MN and the RN can be independently reconfigured to efficiently map different dot product partitions through the creation of dynamic-size clusters.

The reduction stage through the RN is a critical part of the accelerator (it has a computation complexity higher than $O(1)$–see Figure 3.1) since it determines: (i) the flexibility of the architecture to map simultaneous variable-size cluster reductions (unfeasible in rigid accelerators); and (ii) the reuse pattern that will determine how the elements flow during the reduction, and therefore, the efficiency of the dot product reduction. These two features constitute what in this chapter we refer to as the *reduction dataflow*. To provide high energy-efficiency and low latency when processing the reduction dataflow, flexible DNN accelerators often dedicate a large part of the chip resources to implement the RN. For instance, MAERI [70], which clearly distinguishes between the 3 stages mentioned above, dedicates up to 25% and 38% of the chip area and power budget, respectively, just for this network.

Recent proposed accelerators implement one out of the three following reuse patterns which are directly linked to the way they manage folding situations: i) *Temporal (T) reduction*, in which each unit keeps the dot product to be reduced stationary in a register and performs a temporal accumulation of the different psums over that register (represented in Figure 3.2a); ii) *Spatial (S) reduction*, which coordinates a network of adder units, usually with a tree topology for efficiency, to perform a spatial reduction (illustrated in Figure 3.2b); and iii) *Spatio-Temporal (ST) reduction*, that combines both approaches usually performing a spatial reduction first, followed by a final temporal one to accumulate different folding iterations (Figure 3.2c). Among the three approaches, the latter has

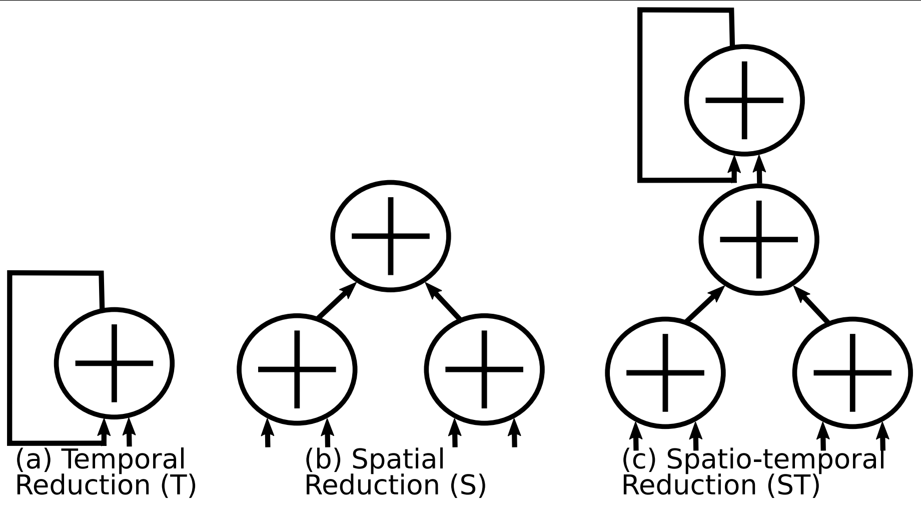

(a) Temporal Reduction (T)
(b) Spatial Reduction (S)
(c) Spatio-temporal Reduction (ST)

Figure 3.2: Reuse patterns implemented in DL accelerators for partial-sum reduction.

been shown to reach better performance [68]. However, as we will demonstrate, adding Accumulators (Ac) to the RN for the T reduction as used in ST-based rigid accelerators, entails significant area and power when it comes to flexible accelerators.

In this chapter, we present a new strategy for Spatio-Temporal reduction in flexible DNN accelerator architectures that dodges the need of adding extra accumulators while keeping performance. In particular, our proposal, that we call STIFT (*Spatio-Temporal Integrated Folding Tree*), binds both spatial and temporal reductions together in a new design of the RN. STIFT builds on the observation that when several clusters are configured, there are some adder units in a tree-based RN topology that go unused. By adding a second root to the tree and additional links between the adder units, we create a new RN topology that is able to perform both spatial and temporal accumulations for all possible cluster configurations. This way, STIFT enables efficient and flexible dot product reductions while reducing the area and power dissipation up to 32% and 31%, respectively, with respect to the state-of-the-art (a RN augmented with accumulators).

We see the following contributions in this chapter:

- We quantify, for the first time, the significant impact that different implementations of folding have on the performance, area and power of flexible DNN accelerator architectures.

- We propose STIFT, a new Spatio-Temporal RN fabric for flexible accelerators that significantly improves performance over current Spatial Tree-based RNs, while avoiding the large area and power expenses of a Spatio-Temporal RN with accumulators.

- We present an exhaustive evaluation that includes RTL implementations and cycle-level simulation of STIFT with DNN models.

The rest of the chapter is organized as follows. Background about contemporary strategies to implement reduction dataflows and their associated RN fabrics in both rigid and flexible accelerators are described in Section 3.2. We detail the architecture, operation with several mapping examples, and properties of STIFT in Section 3.3. Section 3.4 explains the evaluation methodology that involves RTL implementations and cycle-level microarchitectural simulation with STONNE (see Chapter 2). The experimental results showing the benefits obtained in terms of performance, on-chip area and power consumption are exposed in Section 3.5. Finally, Section 3.6 summarizes the main conclusions.

3.2 Background and Related Work

Reduction dataflows may be materialized in different RNs according to the flexibility of the accelerator design to create a variable number of clusters of arbitrary size to be reduced in parallel. In this section, we do a comprehensive literature review, classifying how existing DNN accelerators give hardware support for reduction dataflows. Thus, we distinguish between the RNs in rigid accelerators, which limit the number and size of the clusters, and the RNs in flexible accelerators, which allow to create arbitrary number of clusters of varying size.

Along this section, we will utilize the example depicted in Figure 3.3a, wherein two groups of 8 psums (that have been previously generated by the MN to compute two different dot products) must be accumulated to obtain two output values. For each RN solution, we will use Table 3.1 to discuss different aspects such as the reuse pattern employed, the configured topology, the performance exhibited, its flexibility to map different number of dot products of arbitrary size, and the hardware overhead required to implement the folding support.

Acronym	Reuse Pattern	Topology	Time	Flexibility	Hardware overhead
PT	T	N/A	$O(n)$	Rigid	Low
ST-Linear	ST	Linear	$O(n)$	Rigid	Low
S-Tree	S	Tree	$O[(i+l) \times \log_2(n/i)]$	Flexible	Low
ST-Tree$_{ac}$	ST	Tree	$O(i \times \log_2(n/i))$	Flexible	High
STIFT	ST	Tree	$O(i \times \log_2(n/i))$	Flexible	Low

Table 3.1: RNs implemented in DNN accelerators. *n=Elements in a dot product; i=Folding iterations.*

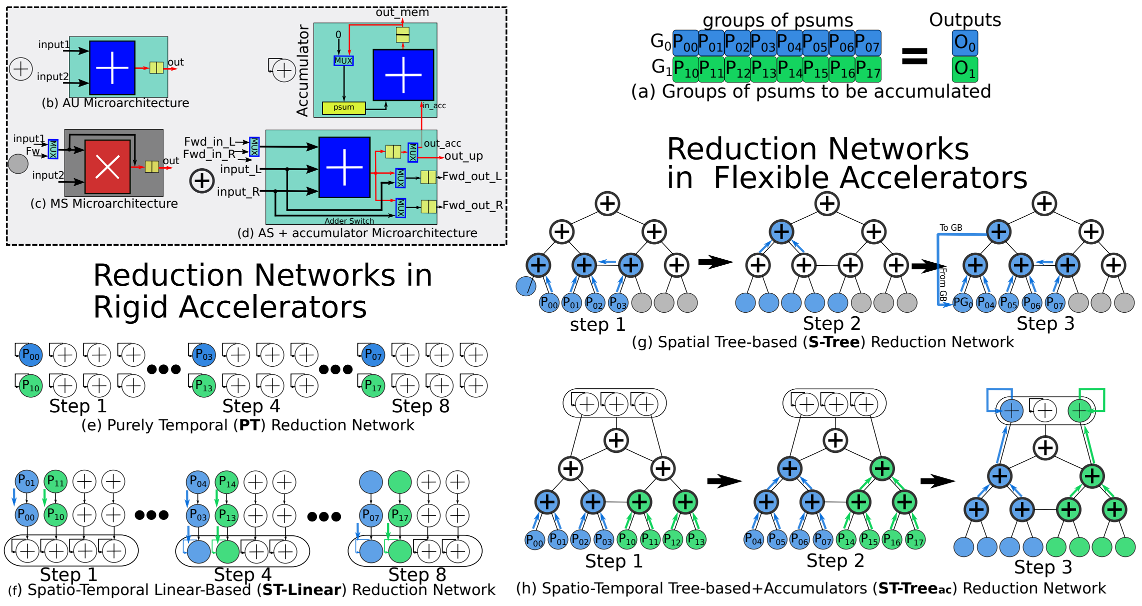

(b) AU Microarchitecture
(c) MS Microarchitecture
(d) AS + accumulator Microarchitecture
Accumulator
out_mem
Reduction Networks in Rigid Accelerators
Step 1
Step 4
Step 8
(e) Purely Temporal (PT) Reduction Network
(f) Spatio-Temporal Linear-Based (ST-Linear) Reduction Network
groups of psums
Outputs
(a) Groups of psums to be accumulated
Reduction Networks in Flexible Accelerators
step 1
Step 2
Step 3
(g) Spatial Tree-based (S-Tree) Reduction Network
(h) Spatio-Temporal Tree-based+Accumulators (ST-Tree$_{ac}$) Reduction Network

Figure 3.3: Different types of reduction networks employed by state-of-the-art accelerators.

3.2.1 Reduction Networks in Rigid Accelerators

First-generation rigid DNN accelerators build on fixed-size clusters of MAC units interconnected by means of a fixed tightly-integrated on-chip network fabric. This means that the distinction between the MN and RN is purely logical, but for the sake of the simplicity, in this chapter we will focus on the accumulation process, which is the one that dictates the behaviour of the execution, and in the end, what defines its energy efficiency. Two RNs are typically employed in this type of accelerators: what we call a Purely Temporal (*PT*)[1] RN and a Spatio-Temporal Linear-based RN (*ST-Linear*)[2].

A PT RN is shaped by a set of non-connected accumulators which are Adder Units or AUs (see Figure 3.3b) augmented with a register for implementing a temporal reuse pattern in charge of temporarily reducing a particular dot product (see Accumulator in Figure 3.3d). This way, each accumulator has to fold $O(n)$ times to compute in $O(n)$ cycles a whole dot product that requires n multiplications, and the maximum number of dot products that may run in parallel is dictated by the number of accumulators in the RN. Figure 3.3e shows a 3-step walk-through example. As we may appreciate, the two dot products to be computed are mapped onto two accumulators. In step 1, the two units are initialized with the first psums. Then, the following steps accumulate the next psums, performing one sum per adder every cycle. Step 4 corresponds to the accumulation done in cycle 4, while step 8 finalizes the calculation after 8 cycles. This approach has been used by several prior accelerators such as [28,78].

Contrarily, some accelerators such as [16,61], exploit spatial reduction utilizing a Spatio-Temporal Linear-based RN (ST-Linear). This RN relies on several columns of simple AUs (see Figure 3.3b) connected vertically through a 1-D linear array (see Figure 3.3f). Each column of the topology corresponds to a cluster of AUs able to map and spatially reduce a single output vertically. Since it is very common to find that the number of AUs located in a column is smaller than the number of psums needed to be accumulated for a certain output (i.e., folding is needed), these networks rely on a Spatio-Temporal accumulation scheme to manage the folding accumulation process. With this aim, each column of AUs is extended with an extra accumulator that is in charge of performing temporal accumulation of the spatially accumulated iterations[3]. This way, similar to the PT RN, the total amount of time for a cluster of adders to calculate a dot product is $O(n)$. Differently, the fact of employing a spatial reduction may increase unit utilization in several occasions. Figure 3.3f depicts an example. Here, the two outputs to be reduced are mapped onto the two first columns of the 2×4 topology and the reductions flow vertically during the three steps until the entire operation is reduced.

Both PT and ST-Linear RNs exhibit two main drawbacks: First, they are composed of a rigid interconnection network that makes them unable to gracefully

[1]Observe that this corresponds to an Output Stationary (OS) dataflow in [117].

[2]This could link, for example, with the Weight Stationary (WS) dataflow in [117].

[3]Note: accelerators such as Eyeriss [22] do not have such accumulators for folding support and would be purely Spatial Linear.

adapt to different dot product dimensions and sizes. This has already been shown to lead to under-utilization of the computing units, which can significantly impact energy efficiency [23,31,68,70]. The second drawback is that n-size reductions take $O(n)$ cycles to complete, which can be further improved by naturally performing the reductions in a tree-based manner

As it may be appreciated, both PT and ST-Linear RNs are composed of a rigid interconnection network that makes it difficult adaptation to different dot product dimensions. This leads, in general, to under-utilization of the units, which in turn translates into significant energy inefficiency of the architecture when running a particular DNN model [31]. To solve this problem, flexible architectures has recently emerged which incorporate configurable RNs able to accommodate a variable number of arbitrary-size clusters.

3.2.2 Reduction Networks in Flexible Accelerators

In order to overcome the limitations presented in rigid accelerators, recent *flexible* accelerators advocate having physically separated and reconfigurable DN, MN and RN fabrics. The RN is built from more configurable AUs called Adder Switches (ASs). Each AS is an AU augmented with a tiny switch to enable arbitrary cluster reductions of variable size over the same physical RN (see the AS block in Figure 3.3d), thereby significantly increasing unit utilization [31,70].

The type of RN that materializes this flexibility is a Spatial Reduction Tree (S-Tree) used in both the ART and the FAN RNs proposed for MAERI [70] and SIGMA [31] accelerators, respectively. This type of RN enables efficient reduction by employing a binary tree-based accumulation so that an ideal whole reduction operation should take $O(\log_2 n)$ to be completed (see next for further details). Besides, to enable the parallel execution of any number of arbitrary-size clusters, they utilize augmented links (see the horizontal extra link between the two central ASs in the example of Figure 3.3g) which avoid conflicts when multiple reductions are mapped in the tree. The challenge to address, which is not analyzed with detail in any of these works, is how to manage the common situation of folding, in which the number of multiplications in a dot product is larger than the number of multiplier units implemented in hardware. In this specific purely spatial approach, which is the one implemented by these RNs [3,6], the psum obtained in each cluster iteration is spatially sent to the Global Buffer (GB), and subsequently redistributed from it to a dedicated Multiplication Switch (MS) (see Figure 3.3c to observe the microarchitecture details of an MS), responsible only for forwarding it back to the RN. This way, the entire accumulation process is performed spatially. This is illustrated in Figure 3.3g. The leaves of the trees are MSs that just generated the psums. The values for the first iteration are initially sent and computed in steps 1 and 2. Then, for computing the second iteration, the value of the psum calculated in step 2 is passed through the GB in step 3, from where it is injected back into the RN by using a dedicated MS (the leftmost one) that just acts as a forwarder. The process is repeated until the whole dot product is spatially computed.

We observe in our evaluation (see Section 3.5) that this strategy, yet simple to implement in hardware, has two major drawbacks:
i. Low theoretical utilization of the MSs, as the required extra MS impedes to map efficiently some number of dot products. Note that, in our example, the extra MS needed to forward the psum impedes using the rest of the multipliers to calculate the other dot product, which also would need 5 multipliers.
ii. Low effective utilization of the mapped MSs as this implementation impedes to iterate over the same dot product in a *pipelined manner*. In the example, the MSs would not be able to operate in step 2, as the psum required for the second iteration has not been calculated yet. In the general case, the time needed to compute an entire operation is $O[(i + l) \times \log_2(n/i)]$ as each of the iterations (i) will require to wait as many cycles as levels in the tree (l).

We would like to emphasize that these two drawbacks had not been observed yet in the previous works [70], and reveal the importance of proper folding management in flexible accelerators.

In order to overlap multiplications and sums of consecutive iterations of the same dot product, and thus, be able to attain a seamless pipelined execution for folding, it is necessary to break the dependency between two consecutive iterations by composing a Spatio-Temporal Tree (*ST-Tree*). To this end, one particular extension over the previous S-Tree, leveraging the design discussed for some ST-based rigid accelerators (Figure 3.3f), could be to add a set of accumulators, connected with the ASs, in charge of temporarily accumulating the different psums being calculated for each cluster. We call this approach an ST-Tree+Accumulators *(ST-Tree$_{ac}$)* RN. In this way, the time needed to complete an n-size operation in this case is $O(i \times \log_2(n/i))$. Besides, having these accumulators would eliminate the need to allocate the additional MS in each cluster just to forward the psum, thus making much better use of the MSs.

Figure 3.3d illustrates the microarchitectural details of an AS connected to its corresponding accumulator, able to perform temporal accumulation for a cluster that collapses on that particular AS. The AS just needs a control signal to diverge the resulting psum to the accumulator rather than to the top node or the Global

Buffer. This way, the different psums reach the accumulator, which temporarily accumulates them by using a simple accumulation register and an extra adder unit.

As a representative example, Figure 3.3h shows how the accumulators would be employed with two configured clusters. In this case, there are 3 units, each of them connected to a different AS. As it may be appreciated, the first two steps are used to calculate the two psums of the first iteration (one per cluster). Observe that the calculation of the products required in the second iteration starts while the tree is computing the psums corresponding to the first one (step 2). Then, in the third step, these two psums are stored in the two accumulators, and at the same time, the tree of ASs computes the psums corresponding to the second iteration. In a fourth step (not shown in the figures for the sake of brevity), the result of the psums of the second iteration would be sent to the corresponding accumulators, where the final outputs would be obtained.

This approach has two major inconveniences. First, and most importantly, it entails significant area and power overhead as it requires to significantly increase the number of adders in the RN (i.e., the extra accumulators). The second inconvenient has to do with the resulting increased complexity of the design by requiring the addition of this extra component.

3.3 STIFT: A Spatio-Temporal Integrated Folding Tree

The discussion presented in the previous section reveals that Spatio-Temporal RNs are the best match for both rigid and flexible DNN accelerators by endowing the RN with accumulators. To avoid incurring into high on-chip area and energy overhead due to these extra components, the number of accumulators and the connections to the ASs should be done according to the size and number of the clusters that are supported by the architectural design.

In the case of rigid accelerators, for example, in an R×C ST-Linear RN like the one used in the Google's TPU, the number of the accumulators should be equal to the number of columns (i.e., C) of the design (see Figure 3.3f), as a whole column is typically used to vertically reduce a cluster. This way, in this type of architectures the accumulation units are affordable and can be implemented without incurring significant area and energy overhead.

On the other hand, in the case of the higher-performance flexible accelerators, since the number, size and exact location of the clusters to be mapped onto the architecture is undetermined, it is not known *a priori* the number of accumulators that are needed to support all the configurations. Therefore, in order to support the worst case in which every multiplier was an independent cluster, the architecture would need as many accumulators as ASs, doubling the area required for the RN. Reducing the number of accumulators could lower this overhead, but it would limit the number of clusters that can be accumulated in parallel, constraining the flexibility.

This way, this approach has two major inconveniences when it comes to flexible architectures. The first, and most important, is that this approach usually entails significant area overheads to be able to guarantee the level of flexibility that allows these flexible accelerator architectures maintain the promised high efficiency and configurability. The second inconvenient has to do with the resulting increased complexity of the design by requiring this extra component.

To ensure efficient folding support in flexible accelerator architectures and to avoid the addition of such extra accumulators, we propose in this chapter a novel Reduction Network fabric, specifically suited for flexible accelerator architectures. We call our proposal STIFT which stands for *Spatio-Temporal Integrated Folding Tree*. Similarly to ST-Tree$_{ac}$ RN using accumulators, STIFT is capable of running any number of dynamic-size clusters in a non-blocking manner, but unlike this one, it enables efficient and flexible support to ensure full non-blocking processing of folding.

3.3.1 STIFT Topology

The observation behind STIFT is that for the S-Tree RN employed in recent accelerator proposals [31,70], there are free ASs when two or more clusters are configured (for the example in Figure 3.3h, the AS at the root of the tree is not used as two clusters have been defined). The more (and in consequence smaller) clusters that are formed, the more ASs go unused. So, the immediate question is: *why not dedicate these free ASs to accumulating the psums for each cluster?*

To do so, we obviously need that the RN can guarantee, at least, one free AS per each cluster. Taking a look at Figure 3.3h, we can clearly see that the number of ASs in the tree is insufficient for our purpose (note that when the two clusters are formed, just one AS becomes free). We can also see that we can easily solve the problem by adding a second root to the tree, and also several links to guarantee that every cluster, independent of its configuration (size and/or location), can always have access to one of the free ASs that will be used to accumulate the corresponding psums. In broad terms, this is STIFT.

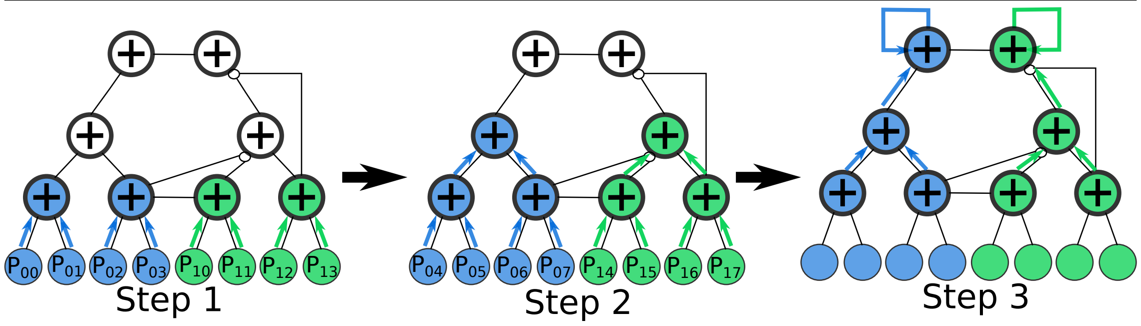


Figure 3.4: STIFT running the example shown in Figure 3.3a.

Before delving into the details of how the extra links required by STIFT are added, Figure 3.4 shows an example of how folding would be carried out with STIFT. In this case, the ASs have to be extended in order to be able to act either as accumulators or ASs (further details in Section 3.3.2). We call these units Extended Adder Switches (eASs). As we can see, the top eASs are employed to accumulate the two dot products being calculated. This way, in this case, after step 2, the eAS on top of each cluster sends its psum to the corresponding root of the tree, which will be configured to accumulate the psums calculated in the two iterations. Once the calculation of the dot products is finished (all the psums have been accumulated in each case), these nodes will send the accumulated values to the Global Buffer, and then, they will begin to accumulate the psums of the next dot product assigned to each of the clusters. Note that, as with ST-Tree$_{ac}$, STIFT removes the dependency among consecutive iterations, and hence, enables fully pipelined execution in the MSs and eASs. However, unlike ST-Tree$_{ac}$, STIFT does not require the addition of extra accumulation units, but instead achieves the same goal by only requiring some lightweight extra logic to leverage the ASs to play the role of both ASs and accumulators, one extra root node, and some extra connections between the eASs.

A detailed view of a 16-wide STIFT topology is depicted in Figure 3.5b. All the links that are inherited from the S-Tree RN topology that we use as base (the ART network [70]) for building STIFT are shown either in pink or black colors. In red, we identify the new links that STIFT requires and that we call the *folding links*. We add the minimal number of links to ensure that every cluster can use an eAS to accumulate its psums and produce its final output value. This is achieved by using either a pink or a red link.

In Algorithm 3.1, we show how to add the extra links required by STIFT (the red links in Figure 3.5b) for an arbitrary number of adders (`numAdders`).

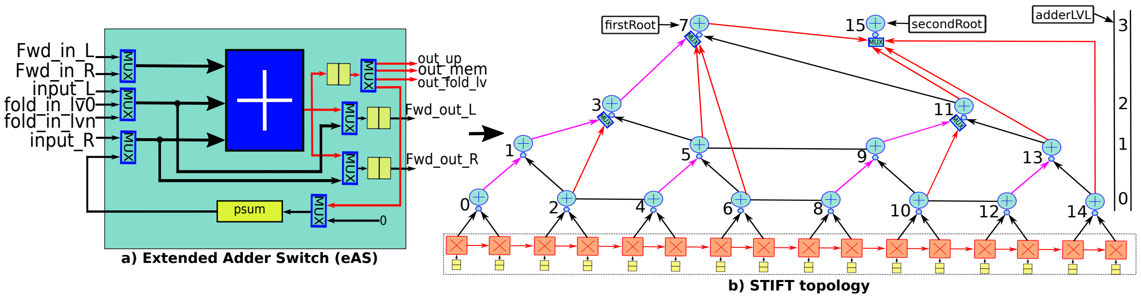


Figure 3.5: a) Extended adder switch microarchitecture of STIFT. b) STIFT topology. The black and pink links are shared with ART, but the pink ones are also used for folding purpose. The red links are additional folding links introduced in STIFT to remove all structural hazards during the folding process.

According to this algorithm, for each of the nodes in the topology (i.e., *i*), we iterate over the levels (i.e., *lvl*) that are below the node to connect its corresponding pair in that level. By establishing the new links in this way, we guarantee that every eAS in the RN can have a different associated parent eAS that would be used to perform the accumulation process for the cluster that converge in the first eAS.

3.3.2 Microarchitectural Implications of STIFT

Figure 3.5a shows the microarchitectural details of each of the eASs used in STIFT. As observed, it is quite similar to the ASs of a flexible RN such as ST-Tree$_{ac}$ (and that we explained in Section 3.2). Since the proposed STIFT topology enables integrated temporal accumulation, every of its eAS has an extra register

```
//Code to add STIFT links
//Inputs: numAdders, adderLVL
for(int i=0; i < numAdders-1; i++):
 for(int l=1; lvl<adderLVL[i]; lvl++):
  connect adder i with: i - 2^(l-1)
//Connecting first with second root
int firstRootID=numAdders/2 - 1
int secondRootID=numAdders / 2
Connect firstRootID with: secondRootID
```

Algorithm 3.1: Pseudocode illustrating how the additional links required by STIFT are added.

which is used to store the accumulated partial sum in case it is needed (the eAS acts as the accumulation point for a particular cluster). Besides, since the left input data (see Figure 3.5b) could be injected either from the left-child eAS or from any other lower-level eAS which sends the psum to be accumulated, it is necessary to incorporate an extra multiplexer unit selecting this input. The size of this multiplexer varies according to the level in which the eAS is located in the reduction tree. In general, given an eAS at a certain level *L*, the size of its multiplexer will be *L-1:1*, as every level in the tree will have a certain eAS connected that will provide the psums to accumulate (when the eAS is used for this purpose). In order to avoid the critical path delay degradation due to the long wires connecting the different levels in the topology, all of the inter-eAS/MUX wires are implemented in the semi-global metal layers using standard repeater wires.

Each eAS in STIFT can act either as an AS (psum spatial generator) or as an accumulator (psum temporal reducer). If the eAS is configured as an AS, then it will receive the inputs from the children eASs, will calculate the psum, and will send the output to its parent eAS (as done in S-Tree). On the contrary, if the eAS is configured as an accumulator, the left input will be obtained from the eAS that sends the psums, and the right input will be taken from the internal register. Once the final accumulation has been performed, the result will be sent directly to the Global Buffer. All the additional multiplexers, as well as the new operation mode are configured by the mapper (see Figure 3.1) based on the size and number of the clusters to be reduced. The benefit of this approach is that since STIFT topology allows the eASs to operate either in psum spatial generator or psum temporal reducer modes, they can reuse the adder unit already contained in the eAS, reducing significantly the area overhead and power dissipation in comparison to adding the extra accumulators required in the ST-Tree$_{ac}$ RN (see Section 3.5).

3.3.3 STIFT Mappings

3.3.3.1 Examples of Diverse Mapping Configurations and Reduction Dataflows with STIFT

Figure 3.6 depicts 5 possible mapping configurations in a 16-wide STIFT RN to perform the accumulation of 8 16-size groups of psums (that have been previously generated by the MN to compute 8 different dot products). As we can see, the flexibility of STIFT allows not only to map arbitrary size clusters, but also to perform temporal accumulation for all of them. Next, we present in detail each mapping configuration. For ease of explanation, we have labeled each node in the tree with a different number.

Mapping 1: This case is the simplest possible mapping configuration as an entire group of psums is reduced by mapping it onto all the available MSs in the MN fabric (16). As we can observe, the cluster collapses in the first root of the tree (i.e., node 7) and the psums are forwarded to the second root (i.e., node 15) to be reduced in a pipelined manner. Since in this case the cluster size equals the group size, there is just one iteration (i.e., temporal accumulation is not required), and the only psum generated will be directly forwarded to main memory. Subsequent groups of psums will be mapped to the MSs similarly, generating, one after other, the eight final reductions.

Mapping 2: In this particular example, two groups of psums are mapped for reduction onto STIFT. As mapping the entire two groups would require 32 MSs, this reduction has to be divided into two 8-size clusters. This means that each cluster requires 2 iterations which are temporarily accumulated by the two roots of the tree (i.e., nodes 7 and 15), which are configured as accumulators. In this figure, we can note the way in which eAS labeled with 11 requires the use of the folding link to forward the generated psum to node 15. Sending the psum to any other node would produce a structural hazard (the corresponding adder in the eAS has to carry out two different psums) which would break the computation pipeline.

Mapping 3: In this mapping, four groups of psums are allocated to the 16 MSs. To do so, the computation is divided into four 4-size clusters, each performing 4 iterations in total. As it may be appreciated, the 4 clusters collapse in the 4 eASs belonging to the first level of the tree, and therefore the use of folding links by the nodes 5 and 13 is required to avoid the kind of structural hazards in the upper levels already illustrated in the previous mapping.

Mapping 4: This is the extreme case of mapping in which eight 2-size clusters are configured to reduce 8 groups of psums, each requiring 8 iterations. Even in this case in which the 8 clusters are spatially collapsing in the 8 eASs belonging to the first level of the tree, there are always free wires and eASs to send the corresponding psums and carry out the temporal accumulations, respectively, in a pipelined manner.

Mapping 5: This case illustrates how STIFT can be utilized to map arbitrary size clusters. This enables to efficiently support scenarios in which sparsity is exploited. In these cases, the matrices are compressed (e.g., in CSR or bitmap formats) to reduce computation and memory footprint, and the resulting dot products can have variable size. In this example, four groups of psums are mapped to be reduced, but they present different sizes. As we can see, the groups 0, 1, 2, and 3, with cluster sizes of 3, 5, 2 and 6, respectively, are mapped, which require 5, 3, 7 and 2 iterations, respectively. Like the ART, STIFT leverages the intermediate links between eASs that do not share the same parent for mapping irregular cluster sizes. Even in this case, STIFT supports temporal accumulation by taking advantage of the novel folding links.

3.3.3.2 Generating Mapping Signals

Given a particular mapping, the control logic signals to be sent to STIFT RN to route the psums can be generated either offline or at runtime by following the steps given as follows. First, the ART routing decisions are made by applying the particular ART algorithm (described in [70]). This algorithm ensures that the clusters are able to reduce in a non-blocking manner for a single iteration. Once this process is completed, the eASs where the clusters are collapsing are configured. If the position of a particular eAS in the level of the tree is even, the psum is configured to be sent to the parent using the ART link. Otherwise, the eAS is configured to forward the psums to the folding link.

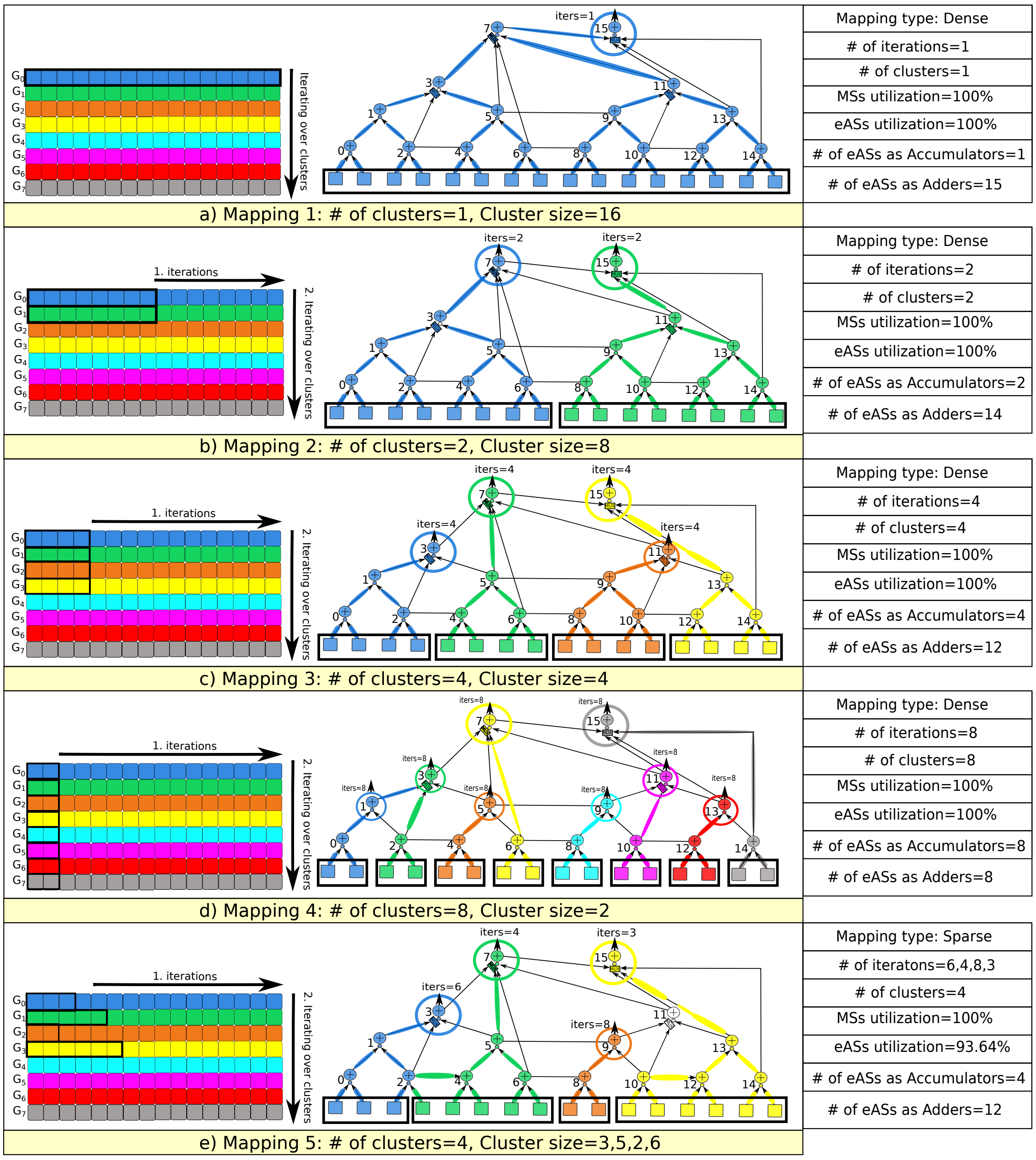
3.4 Experimental Methodology

To prove the benefits of STIFT as the RN of a flexible accelerator architecture, our evaluation methodology considers the following three different angles: 1) RTL implementation to faithfully obtain both energy and area numbers; 2) synthetic evaluation to describe the main benefits of STIFT; and 3) end-to-end evaluation to demonstrate how STIFT can be used to run real DNN models. Below, we describe each of these three angles in detail.

3.4.1 RTL Implementation

We analyze the power and on-chip area overheads that the different RN solutions entail. In particular, we have implemented both STIFT[4] and ST-Tree$_{ac}$ RNs in BSV (Bluespec System Verilog) [1], and use Synopsys Design Compiler and Cadence Innovus Implementation System for synthesis and place-and-route, respectively, using TSMC 28nm GP standard LVT library at 800 MHz. For comparison, we

[4]STIFT is available on https://github.com/maeri-project/stift-bsv.



a) Mapping 1: # of clusters=1, Cluster size=16

Mapping type: Dense
of iterations=1
of clusters=1
MSs utilization=100%
eASs utilization=100%
of eASs as Accumulators=1
of eASs as Adders=15

b) Mapping 2: # of clusters=2, Cluster size=8

Mapping type: Dense
of iterations=2
of clusters=2
MSs utilization=100%
eASs utilization=100%
of eASs as Accumulators=2
of eASs as Adders=14

c) Mapping 3: # of clusters=4, Cluster size=4

Mapping type: Dense
of iterations=4
of clusters=4
MSs utilization=100%
eASs utilization=100%
of eASs as Accumulators=4
of eASs as Adders=12

d) Mapping 4: # of clusters=8, Cluster size=2

Mapping type: Dense
of iterations=8
of clusters=8
MSs utilization=100%
eASs utilization=100%
of eASs as Accumulators=8
of eASs as Adders=8

e) Mapping 5: # of clusters=4, Cluster size=3,5,2,6

Mapping type: Sparse
of iteratons=6,4,8,3
of clusters=4
MSs utilization=100%
eASs utilization=93.64%
of eASs as Accumulators=4
of eASs as Adders=12

Figure 3.6: Different mapping configurations to perform 8 16-size cluster accumulations on a 16-wide STIFT.

also consider MAERI's ART design [3] as an example of S-Tree RN. We study how the different RN fabrics scale by exploring different RN widths (number of multiplier units) and data formats (i.e., INT16, FP16 and FP32).

3.4.2 Synthetic Evaluation

We conduct a comparison of the performance (runtime) that the three flavours of RNs (S-Tree, ST-Tree$_{ac}$ and STIFT) achieve for reduction operations. To do so, we have implemented these three RNs in our STONNE simulator [82]. As described in Chapter 2, STONNE is a cycle-level, highly-modular and highly-extensible microarchitectural simulation framework that can be potentially plugged into any high-level DL framework as an accelerator device. Therefore, all the activity found during the execution of the DNN inference procedure of real, unmodified DNN models processing real input data can be accurately simulated at cycle level. STONNE also allows us to easily configure and implement different reduction networks, thereby facilitating the task of evaluating and comparing our three target RNs.

For a fair comparison, we have configured the simulator with the same DN and MN networks in all cases. In particular, we consider a tree-based DN and a linear-based MN to build MAERI-like flexible accelerator architectures [70]. Besides, we assume the next hardware parameters: 256 MSs, FP16 arithmetic datatype, 108-KiB Global Buffer size, 128 elements/cycle I/O Global Buffer bandwidth, and two 256 GB/s, 512-MiB HBM2.0 DRAM modules. For the resulting three accelerator architectures, we run three different experiments:

1. First, to study how a single cluster behaves, we have executed seven synthetic workloads by mapping a single cluster ranging in size from 2 to 128. To study how each RN deals with folding, we iterate each cluster 512 times [5], which allows us to observe how the RN topology affects performance when running a certain representative reduction.

2. To observe the impact of mapping multiple same-size clusters in parallel when it comes to folding, we have executed other seven synthetic workloads. Differently, this time we have swept the number of clusters to be mapped and its size, but keeping the number of used multipliers as 128 in all the cases. This way, we have executed the next configurations: 64 clusters of size 2 (*64C-S2*), 32 clusters of size 4 (*32C-S4*), 16 clusters of size 8 (*16C-S8*), 8 clusters of size 16 (*8C-S16*), 4 clusters of size 32 (*4C-S32*), 2 clusters of size 64 (*2C-64S*) and 1 cluster of size 128 (*1C-128S*). Note that we always use the same terminology *xx*C-*yy*S

[5]We observe this is a representative folding iteration number when running real DNN workloads.

to refer to *xx* mapped clusters and its size (*yy*). Similar to the aforementioned experiment, we iterate each cluster 512 times.

3. Finally, we demonstrate that STIFT is able to map multiple variable-size clusters in parallel and still perform the folding efficiently. In this way, in this experiment we configure irregular-size clusters keeping always the utilization of 128 multipliers and iterate each cluster 512 times.

3.4.3 End-to-End Evaluation

We have determined the overall impact on performance and energy consumption of the different RNs when running complete DNN workloads. To do so, we have used STONNE together with its PyTorch interface [5] to execute the inference procedure of seven full DNN models on the three resulting accelerator configurations (S-Tree, ST-Tree$_{ac}$ and STIFT). We consider *Alexnet* [67] (*A*), *Mobilenets* [53] (*M*), *Squeezenet* [56] (*S*), *Resnets-50* [49] (*R*), *VGG16* [113] (*V*), *ssd-Mobilenets* [75] (*S-M*) and *BERT* [27] (*B*), taken from MLPerf [106] benchmark suite (further information in Table 2.1). In all the cases, we have obtained the best mapping configurations using the mRNA tool [125]. To calculate the energy consumed by each execution, we have fed the simulator with the energy numbers obtained during the RTL synthesis and presented in Section 3.5.

3.5 Results

3.5.1 RTL Evaluation

Figure 3.7 shows the area (μm^2) and power (mW) synthesis results for the three evaluated RN designs: *S-Tree*, *ST-Tree$_{ac}$* and *STIFT*. For both area (top) and power (bottom) figures, we plot three different design points varying the data type that the RN utilizes (INT16, FP16 and FP32). For each one of them, we also show in the *x-axis* the results for 5 RN sizes by scaling the number of MSs (i.e., the leaves) from 64 to 1024. Each bar is split into two components: the amount of area/power attributable to the Adder Units (*AUs*) and that corresponding to the rest of elements of the ASs and the topology, namely switching logic, registers, multiplexers and wires (*Rest*).

As it may be appreciated, for INT16 (see Figures 3.7a and 3.7d), which is a datatype typically used in some inference accelerators (e.g. TPUv1 [61] can operate on 8- and 16-bit data elements), we observe that STIFT saves up to 19% and 20% power and on-chip area, respectively, as compared with ST-Tree$_{ac}$. In

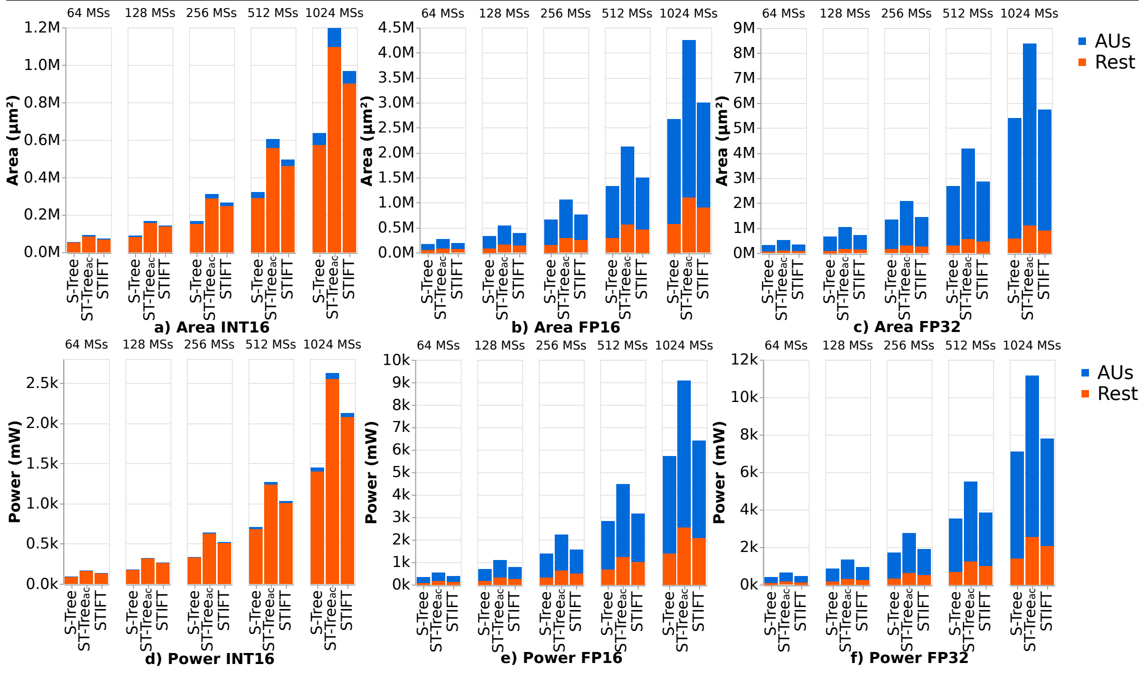


Figure 3.7: Area (μm^2) and power (mW) synthesis results for the three evaluated RN designs (*S-Tree*, *ST-Tree$_{ac}$* and *STIFT*) varying RN size (64, 128, 256, 512 and 2014 MSs) and data types (INT16, FP16 and FP32).

this case, the simple INT16 AUs do not represent a significant fraction of the logic of the RNs (see blue portion of the bars), and therefore, the elimination of the extra AUs of the accumulators with STIFT brings modest savings. Reductions in power and on-chip area in this case come mostly from the elimination of the extra wires and logic needed to connect the accumulators when these units are used.

The reduction in the number of elements can also be observed in Table 3.2, where we show the number of wires presented in each RN and the overhead added by *ST-Tree$_{ac}$* and STIFT with respect to *S-Tree*. As we can see, STIFT not only reduces the number of AUs, but also the wires (average reduction of 20%), making it more efficient even when the area and energy fraction represented by the AUs is not very significant. This also demonstrates that our STIFT design will introduce less overhead as technology shrinks. In these cases, the energy spent in the wires becomes dominant and STIFT ensures high flexibility while keeping the RN more efficient. Finally, we can also observe that the overhead introduced by the muxes in the STIFT design does not translate into a scalability

MSs	AUs S-Tree	AUs ST-Tree$_{ac}$	AUs STIFT	Wires S-Tree	Wires ST-Tree$_{ac}$	Wires STIFT	Muxes STIFT
64	63	126 (+100%)	64 (+2%)	152	215 (+41%)	184 (+21%)	63
128	127	254 (+100%)	128 (+1%)	311	438 (+40%)	375 (+20%)	127
256	255	510 (+100%)	256 (+1%)	630	885 (+40%)	758 (+20%)	255
512	511	1022 (+100%)	512 (+0.1%)	1269	1780 (+40%)	1525 (+20%)	511
1024	1023	2046 (+100%)	1024 (+0.1%)	2548	3571 (+40%)	3060 (+20%)	1023

Table 3.2: Number of MSs, AUs, wires and multiplexers (Muxes) required to implement *S-Tree*, *ST-Tree$_{ac}$* and STIFT RNs for different RN sizes. The percentage of extra elements added in *ST-Tree$_{ac}$* and in STIFT with respect to *S-Tree* is also shown.

issue as the orange bar sizes increase linearly with the RN size for both area and energy numbers.

Power and area reductions are more significant when it comes to the FP16 and FP32 designs, as the AUs are responsible for the most important fraction of the chip area attributable to the RN. For example, for FP16 (mostly used in some inference devices, such as Eyeriss [22], for better inference accuracy) the AUs consume between 70% and 78% of the total chip area devoted to the RN. This results in STIFT obtaining significant area (Figure 3.7b) and energy (Figure 3.7e) reductions over ST-Tree$_{ac}$, which range from 28% to 30% and from 29% to 30%, respectively. As shown in Figures 3.7c and 3.7f, these differences are even larger for FP32 (utilized in training accelerators). In this case, 87% on average of the chip area occupied by the RN is due to the AUs, and STIFT reduces area and energy demands of ST-Tree$_{ac}$ by 32% and 31%, respectively, on average.

Obviously, when compared to S-Tree, STIFT introduces area and power overheads, as it adds the accumulation logic needed to perform temporal reductions. On average across all the design points, the extra power and area overheads are 17% in both cases, much lower than the 39% and 40%, respectively, added by the accumulation buffer in ST-Tree$_{ac}$.

Nevertheless, as we next show, unlike S-Tree and similar to ST-Tree$_{ac}$, STIFT enables pipeline execution of consecutive folding iterations, resulting in much better performance results than S-Tree (and, also, less amount of total energy consumed).

3.5.2 Synthetic Evaluation

Figure 3.8a shows the number of cycles (*y-axis*) obtained for the three RN implementations to complete the reduction of a single cluster varying its size (as

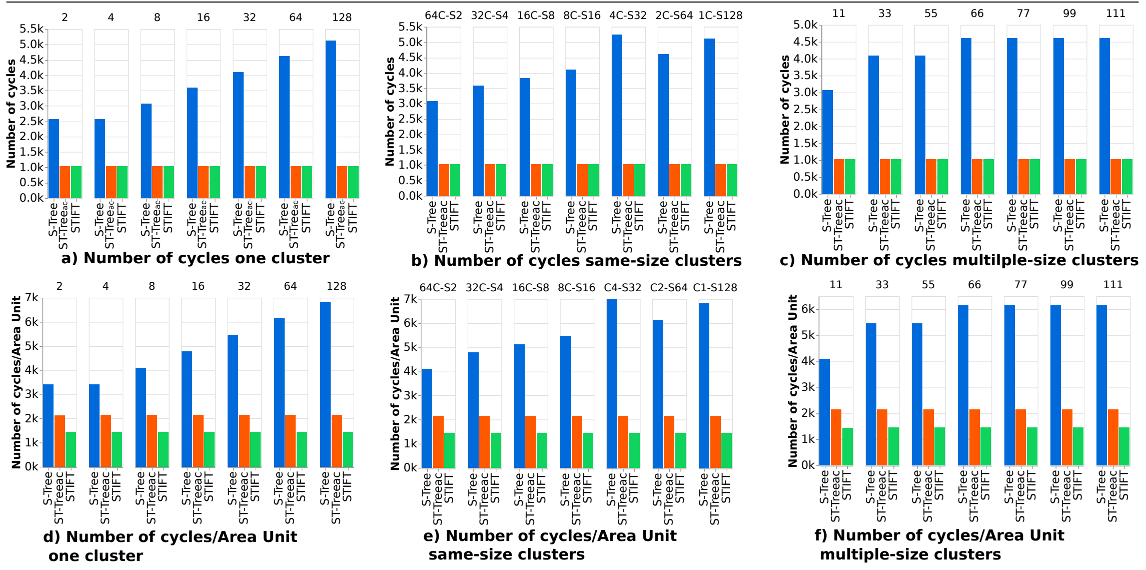


Figure 3.8: a) Number of cycles for the three RNs (*S-Tree*, *ST-Tree*$_{ac}$ and STIFT) to complete the reduction of one single cluster varying its size. b) Number of cycles for the three RNs (*S-Tree*, *ST-Tree*$_{ac}$ and STIFT) to complete the reduction of multiple same-size clusters varying both the number of clusters (C) and its sizes (S). c) Number of cycles for the three RNs (*S-Tree*, *ST-Tree*$_{ac}$ and STIFT) to complete the reduction of multiple variable-size clusters varying both the number of clusters and its sizes. d-f) Number of cycles per unit area for the experiments performed in *a-c*.

described in Section 3.4.2). Across the seven considered sizes, we observe that both ST-Tree$_{ac}$ and STIFT are on average 3.43× faster than S-Tree, demonstrating that when the reductions are performed purely spatially (and the psums are forwarded through the Global Buffer for supporting folding), performance degrades significantly due to the psum dependency between consecutive folding iterations.

The performance gap between ST-Tree$_{ac}$/STIFT and S-Tree is more significant as the cluster size increases. For the smallest cluster size (i.e., 2), we appreciate that ST-Tree$_{ac}$ and STIFT are 2.49× faster than S-Tree, while this difference reaches 4.95× for a cluster size of 128. The reason for this is that the higher the reduction tree, the longer it takes to produce the psum and return it to the MN *via* the Global Buffer, and therefore the longer the MN will have to wait until it can start the next folding iteration. Another important limitation to note is that when folding is used, S-Tree must dedicate one MS per cluster to forward the psums calculated in the previous iteration. This results in worse utilization of the MSs, which usually translates into a larger number of folding iterations required per cluster (due to smaller effective cluster sizes). On the contrary, spatio-temporal approaches like ST-Tree$_{ac}$ and STIFT eliminate this constraint as the accumulation of consecutive folding iterations is performed temporarily on the top of each tree, leaving the entire MN available for effectual computation, and thus allowing for more flexible cluster reduction mappings.

On the other hand, Figure 3.8 shows the results for the second experiment described in Section 3.4.2 in which we sweep the number of mapped clusters and its size (*x-axis*). In this case, we keep all the clusters with the same size. As we can observe, similar to the first experiment, there is a clear gap between the number of cycles obtained with S-Tree and the number of cycles obtained with both ST-Tree$_{ac}$ and STIFT, verifying that STIFT is suitable to run multiple clusters in parallel. However, in this case the results are even more impressive with both STIFT and ST-Tree$_{ac}$ being 4.02$\times$ faster than S-Tree. This difference is due to both spatio-temporal approaches keep the partial sums stationary and therefore reduce the number of writings to memory when multiple clusters are reducing in parallel. This alleviates the pressure on the memory hierarchy which translates even to further performance and improvements.

The results for the third experiment described in Section 3.4.2 are shown in Figure 3.8c. Here, we vary again the number of clusters and its size, but this time we map variable size of clusters in parallel. The *x-axis* in this case shows the largest cluster configured in each execution. This is so, because we have clearly observed that when variable-size clusters are running in parallel, the largest one dominates the execution time while the rest lag far behind it. In this experiment we have observed that both STIFT and ST-Tree$_{ac}$ are 4.07$\times$ faster than S-Tree. Similar to the second experiment, mapping variable-size clusters when STIFT is configured benefit from alleviating the pressure on the memory hierarchy with respect to S-Tree. Furthermore, this effect is even exacerbated as the clusters write at different times due to its irregular size.

Finally, it also may be appreciated that STIFT and ST-Tree$_{ac}$ reach virtually the same performance numbers across the three experiments. This comes as no surprise if we take into consideration that both approaches implement spatio-temporal reductions and behave similarly. The difference in this case is that by integrating temporal accumulations in the RN, STIFT can save the significant area and power overhead that the use of the accumulators add to ST-Tree$_{ac}$. This is further demonstrated in Figures 3.8d-f where we show the number of cycles per unit area obtained when running the three aforementioned experiments depicted in Figures 3.8a-c, respectively. In this case, we have divided each number of

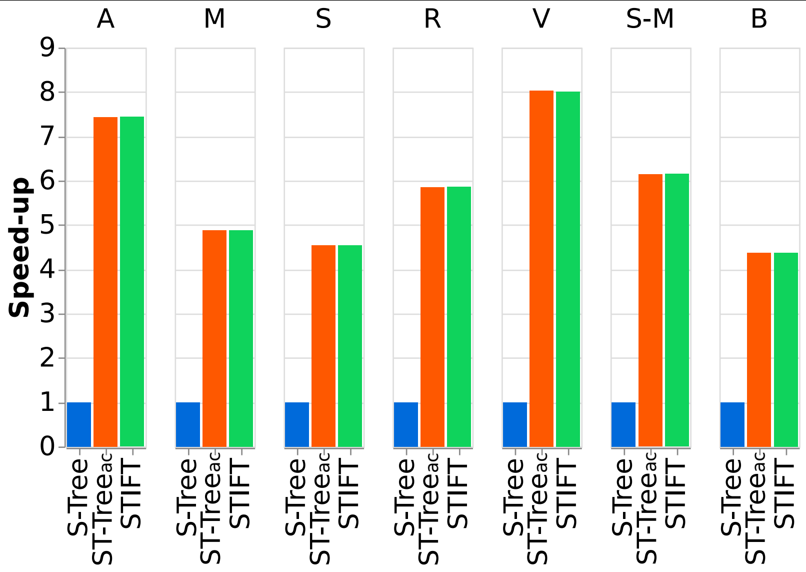


Figure 3.9: Speed-up of ST-Tree and STIFT against S-Tree after running 7 DNN models. *A=Alexnet; M=Mobilenets; S=Squeezenet; R=Resnets-50; V=VGG16; S-M=ssd-Mobilenets; B=Bert.*

cycles shown in the Figures 3.8a-c by the inverse of each RN area occupation given the results described in Section 3.5.1. We observe that on average across the executions of the three experiments, STIFT improves the number of cycles per unit area by 3.30× and 1.48× against S-Tree and ST-Tree$_{ac}$, respectively. Further experiments in next section with real DNN models validate this claim.

3.5.3 End-to-End Evaluation

As for the performance observed for the three configurations under study with complete DNN models, Figure 3.9 shows the speed-ups obtained by ST-Tree$_{ac}$ and STIFT with respect to S-Tree. Our results corroborate the trend observed with the synthetic workloads in that we clearly see that the folding strategy of redistributing the psums through the Global Buffer and the extra MS, results in very poor performance, mainly due to the dependency created between consecutive iterations.

By augmenting the S-Tree configuration with the accumulators, and therefore, implementing spatio-temporal reductions (ST-Tree$_{ac}$), performance speed-up of up to 8× for VGG-16 (5.9× on average across the 7 DNN models) are gained. It is also interesting to appreciate how the magnitude of the gains changes from one

DNN model to another. For instance, for the largest networks, namely Alexnet, Resnets-50, VGG-16 and ssd-Mobilenets, the obtained speed-ups range from 5.86× to 8.03×. On the contrary, for the smallest networks, namely Mobilenets-V1 and Squeezenet the speed-up values are 4.88× and 4.54×, respectively, as their layers are smaller, and therefore, some of them do not make use of folding. In particular, Mobilenets-V1 and Squeezenet require folding in 75% and 39% of their layers, respectively, while this percentage grows to 99% for Alexnet, Resnets-50 and VGG-16. The only exception is BERT, which despite being the deepest network, its runtime is dominated by layers of size 768 (20% of the layers) which barely require 3 folding iterations, and by attention layers, which represent 75% of the total number of layers and do not require folding as their size is small (64). The achieved speed-up in this case is 4.37×.

As we explained in Section 3.2, the reason for these significant performance improvements is that the use of the accumulators allows to break the aforementioned dependency between consecutive folding iterations, thus enabling that the different components of the flexible ST-Tree$_{ac}$ architecture can operate in a pipelined manner (the DN, MN, RN and accumulators). But, what is more important, we can observe that the more area- and power-efficient STIFT design very closely approaches the speed-up values of ST-Tree$_{ac}$.

In Figure 3.10, we consider both achieved speed-ups and area requirements of each design. The area requirements are normalized with respect to the S-Tree case, which is also the reference for the calculation of the speed-ups. Here, we can clearly see that in all cases, STIFT reaches the best compromise between performance and area consumption (the higher Speed-up/Area values). This is due to, compared with ST-Tree$_{ac}$, STIFT achieves virtually the same performance but it requires significantly less area (see Figure 3.7b). Overall, we can appreciate that on average across the seven DNN models, STIFT reaches a speed-up/area ratio of 5.13× while this value is reduced to 3.67× in the case of ST-Tree$_{ac}$.

Finally, Figure 3.11 plots a breakdown of the total amount of energy consumed (static and dynamic energy consumption) in each case, distinguishing between the main components of the architecture. All the results have been normalized with respect to the obtained for S-Tree. As we can see, the energy benefits are not as impressive as the performance numbers. The reason for this is that energy consumption in each case is mainly dominated by the dynamic component (dynamic energy is 30× higher on average), and the number of operations that must be carried out is the same regardless of the implemented folding strategy. Nevertheless, the fact that the total number of execution cycles is greatly reduced in STIFT and ST-Tree$_{ac}$ translates into significant reductions in the static

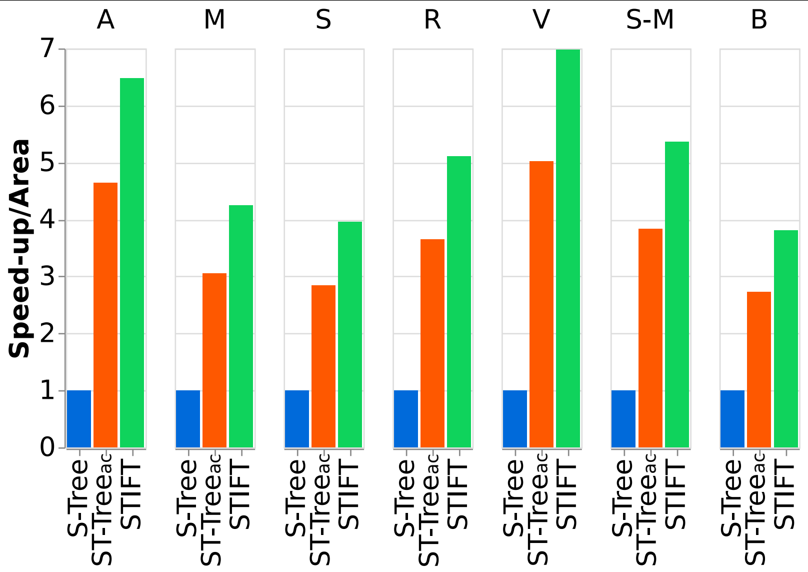


Figure 3.10: Speed-up/Area of ST-Tree and STIFT against S-Tree after running 7 DNN models. *A=Alexnet; M=Mobilenets; S=Squeezenet; R=Resnets-50; V=VGG16; S-M=ssd-Mobilenets; B=Bert.*

component, which in turn results in average reductions of 11% and 9% in total energy (up to 20%) for STIFT and ST-Tree$_{ac}$ RNs, respectively.

3.6 Conclusions

In this chapter, we have demonstrated that the way the Reduction Network (RN) of a flexible DNN accelerator architecture manages folding significantly impacts the achievable performance, when processing the inference procedure of DNN models. More specifically, a purely temporal approach is not able to perform the reductions in a tree-based manner, which limits its efficiency. On the contrary, a spatial reduction typically lacks of sufficient resources to reduce large clusters.

To overcome this, we have presented a new strategy for Spatio-Temporal reduction–STIFT–that dodges the need of adding the extra accumulators used by contemporary RN fabrics in flexible DNN accelerators. STIFT takes advantage of the unused adder units that are available in a tree-based RN topology when several clusters are configured, and adds the minimal elements required to guarantee that for all possible cluster configurations, each one can be associated with one adder unit for temporal accumulation of its generated psums.

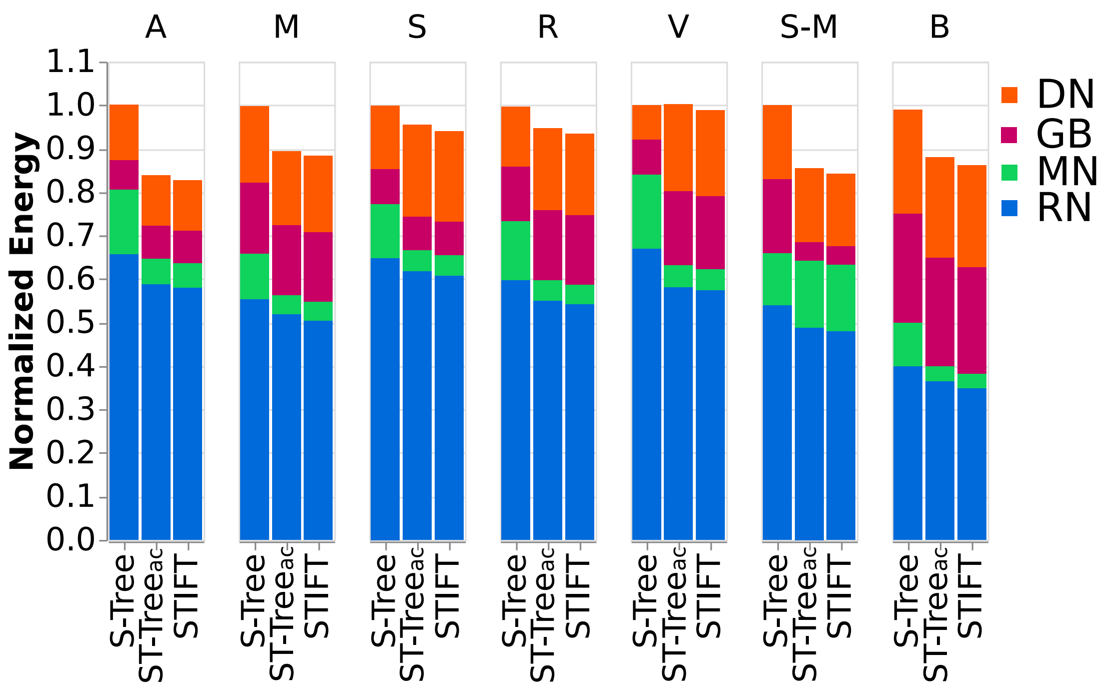


Figure 3.11: Normalized Energy of ST-Tree and STIFT against S-Tree after running 7 DNN models. *A=Alexnet; M=Mobilenets; S=Squeezenet; R=Resnets-50; V=VGG16; S-M=ssd-Mobilenets; B=Bert.*

Through a comprehensive evaluation that includes RTL implementation of different RN approaches able to operate with several data types, we have observed that STIFT obtains area and power benefits of up to 32% and 31% respectively. Overall, in the context of a specific flexible accelerator like MAERI, this translates into general area and energy savings of up to 8% and 12, respectively. Note that MAERI is an accelerator targeting inference which implements the INT16 datatype. Other accelerators targeting training and using a wider datatype may experience larger benefits. Besides, through cycle-level simulation of complete and synthetic DNN models, we have proven that STIFT is up to 3.67× more efficient in terms of the balance between the performance and area requirements for the considered alternatives.

Chapter 4

Flexagon: A Multi-Dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing

4.1 Introduction

Sparsity in tensors is an emerging trend in modern DNN workloads [87,91,107]. These workloads have diverse sparsity ratios, ranging from 0.04% to 90%, and are used in various applications, ranging from personalized recommendations [91] to Natural Language Processing [27]. Sparsity in weights stems from pruning [47] and sparsity inside activations stems from nonlinear functions such as ReLU. As a result, exploiting the benefits of sparsity by directly implementing sparse matrix-matrix multiplication (SpMSpM) has become an important target for customized DNN accelerators [31,50,97,98,114,122,124].

The most common way for these accelerators to exploit sparsity is using compressed formats like Bitmap, CSR and CSC to store and operate (multiply and accumulate) only the non-zero values. This allows to significantly reduce both the memory footprint and the number of operations, which in turn translates into significant energy savings. However, these accelerators vary widely in their hardware implementation and in the exploited dataflow. The dataflows used by these accelerators in terms of the loop order of computation have been broadly

Accelerator	Architectural Features	IP	OP	Gust
TPU	Dense Systolic Array	N/A	N/A	N/A
SIGMA	Configurable Reduce Tree	✓	✗	✗
ExTensor	Intersection Unit	✓	✗	✗
MatRaptor	Merger	✗	✗	✓
Gamma	Fiber Cache, Merger	✗	✗	✓
Outerspace	Merger	✗	✓	✗
SpArch	Matrix condenser, merger	✗	✓	✗
Flexagon (ours)	Flexible Merge/Reduce tree and memory controller	✓	✓	✓

Table 4.1: Comparison of Flexagon with prior Sparse DNN accelerators in terms of supported dataflows. IP=Inner Product, OP=Outer Product, Gust=Gustavson's (Row-wise Product).

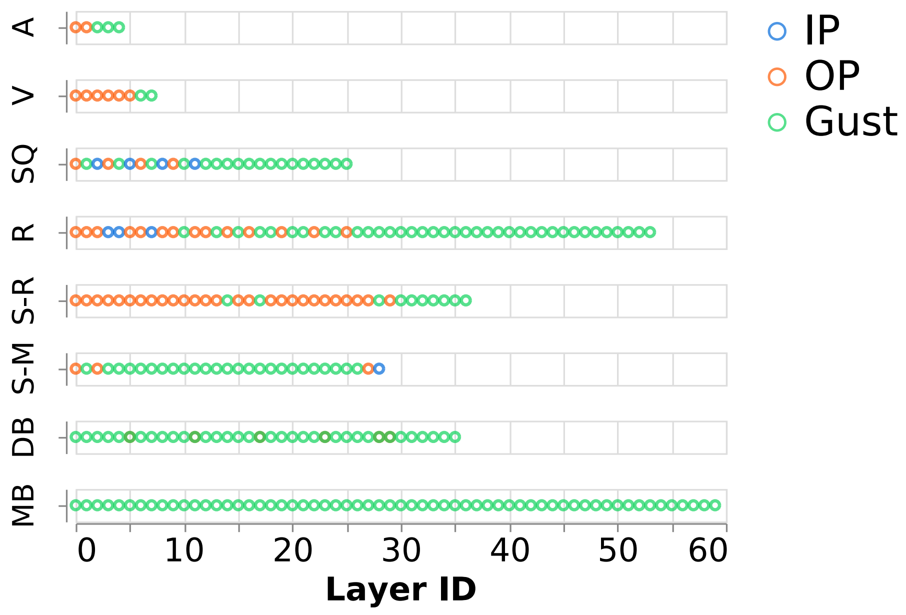


Figure 4.1: Dataflow that obtains the best performance per layer across the DNN models (see Table 4.2 that includes their sparsity ratios). Alexnet (*A*), VGG-16 (*V*), Squeezenet (*SQ*), Resnets-50 (*R*), SSD-Resnets (*S-R*), SSD-Mobilenets (*S-M*), DistilBert (*DB*) and MobileBert (*MB*). IP=*Inner Product*, OP=*Outer Product* and Gust=*Gustavson's*. Please refer to Table 4.2 for sparsity ratios in the models.

classified into `Inner Product` (IP), `Outer Product` (OP) and Row-wise-Product, often called `Gustavson's` (Gust) [44].

Table 4.1 shows prior sparse accelerators and the dataflows they support. While state-of-the-art sparse accelerators such as SIGMA [31], SpArch [124] and GAMMA [122] have been optimized for a fixed dataflow (`Inner Product`, `Outer Product` and `Gustavson's`, respectively), in this chapter, we make the important observation that *the optimal dataflow changes from a DNN model to another, and even within a DNN model, from one layer to another*, so that contemporary fixed-dataflow accelerators cannot adapt well to maximize DNN application performance.

To back up our observation, Fig. 4.1 shows the dataflow that obtains the best performance per layer given the execution of 8 entire DNN models obtained from MLPerf benchmark suite [107] as well as some extra models (details in Table 4.2). Observe that we consider heterogeneous models from different domains, sizes and sparsity ratios. For *MB*, we only show the first 60 layers, which represent 20% out of the total number of layers. To model the three dataflows, the executions have been performed on a 64-Multiplier SIGMA-like, SpArch-like and GAMMA-like architectures (further details in Section 4.4). Overall, the results show that there is a high variability in terms of the dataflow that obtains the best performance across layers belonging to different models and even across the layers within the same model. The NLP models *DistilBERT* (*DB*) and *MobileBERT* (*MB*) present a clear trend towards `Gustavson's`. Specifically, in DB, 30 out of the 36 layers works better with `Gustavson's` while MB benefits from `Gustavson's` in 100% of the layers. On the other hand, extremely sparse models, such as *SSD-Resnets* (*S-R*) and *VGG-16* (*V*), benefit from `Outer Product` in 73% and 75% of the layers, respectively. The rest of the DNN models present a high variability across layers, and the most efficient dataflow changes given the different features of each layer. This highlights that one dataflow does not fit all, and so there is an opportunity to increase efficiency via dynamic adaptation of the architectural components to the most suitable dataflow.

The value of supporting flexible dataflows has been explored extensively for dense DNNs [24, 29, 68, 70]. However, support for flexible dataflow acceleration for sparse workloads is much more challenging because of different ways in which these accelerators handle sparsity. For example, the `Inner Product` dataflow implemented in SIGMA [31] implements a reduction network called FAN to *reduce* the generated partial sums at once, as well as the capacity to perform *intersections* to execute a sparse dot product. On the contrary, the `Outer Product` and `Gustavson's` dataflows implemented in accelerators like SpArch [124] and GAMMA [122] produce partial sums instead of complete sums, and hence, require *merging* the non-zero partial sums and use merger trees for this purpose. A naive implementation using separate hardware widgets for

DNN	**Appl**	**nl**	**spA**	**spB**	**csA**	**csB**
Alexnet (A)	CV	7	70	48	0.56	13.6
Squeezenet (S)	CV	26	70	31	0.05	1.54
VGG-16 (V)	CV	8	90	80	0.55	2.90
Resnets-50 (R)	CV	54	89	52	0.19	1.30
SSD-Resnets (S-R)	OR	37	89	49	0.12	3.60
SSD-Mobilenets (S-M)	OR	29	74	35	0.16	0.31
DistilBERT (DB)	NLP	36	50	0.04	2.25	0.35
MobileBERT (MB)	NLP	316	50	11	0.10	0.07

Table 4.2: DNN models used in this work. CV=Computer Vision, OR=Object Recognition, NLP=Natural Language Processing, nl=Number of layers, sp{A,B}=Average sparsity of the matrices {A,B} (in %), cs{A,B}=Average compressed matrix size for the matrices {A,B} (in MiB). Appl=Application

reductions and merges would lead to large control overhead, significant area overhead and dark silicon (see Section 4.5.4).

To efficiently support different SpMSpM workloads to run modern sparse DNNs, we present Flexagon, the first (to our knowledge) reconfigurable sparse and homogeneous DNN accelerator that can be dynamically adapted to execute the most suited SpMSpM dataflow on a per DNN layer basis. Flexagon features a novel unified *Merger-Reduction Network (MRN)* that supports both reduction of dot products and merging of partial sums. We propose a tree-based topology where the nodes are configured to act either as accumulators or comparators, as explained in Section 4.3. Flexagon also features a new L1 on-chip memory organization composed of three customized memory structures that are able to capture the memory access pattern of each dataflow. The first memory structure is a simple read-only FIFO, which is designed for the sequential accesses that occur during some stages in the three dataflows. The second one is a low-power cache used to back-up the random accesses caused mainly by the `Gustavson's` dataflow. Finally, a customized memory structure called PSRAM , is specifically designed to store and read psums, which is essential for both `Outer Product` and `Gustavson's` dataflows. These memory structures allow us to support all the three dataflows with minimal area and power overheads. Further, our accelerator also prevents the hardware from requiring explicit expensive conversions of compression formats (i.e., from CSR to CSC or vice-versa) [102] between layers as it is possible to easily switch among the most convenient dataflow given a particular compression format (details discussed in Section 4.3).

We summarize our key contributions:

We demonstrate that each SpMSpM operation in modern sparse DNN layers presents different memory access patterns according to matrix dimensions and sparsity patterns. As a consequence, the dataflow that maximizes the performance of a particular SpMSpM operation not only can change between DNN models, but also from layer to layer within a particular DNN model.We present a new inter-layer dataflow mechanism that enables compression format conversions without explicit hardware modules. We design Flexagon, which hinges on a novel network topology (called MRN) that allows, for the first time, support for the three dataflows, and a new L1 on-chip memory organization to effectively capture the memory access patterns that each dataflow exhibits for input, output and partial sums. We extensively evaluate Flexagon using cycle-level simulations of several contemporary DNN models from different application domains, and RTL implementation of its principal elements. Our results demonstrate that Flexagon achieves average performance benefits of 4.59× (ranges between 2.09× and 7.41×), 1.71× (ranges between 1.04× and 4.87×), and 1.35× (ranges between 1× and 2.13×) with respect to the state-of-the-art SIGMA-like, SpArch-like and GAMMA-like accelerators (265% , 67% and 18%, respectively, in terms of average performance/area efficiency).

The rest of the chapter is organized as follows: Section 4.2 explains the compression formats and dataflows utilize by state-of-the-art accelerators. Section 4.3 describes Flexagon and the key design aspects that enable the support of the main dataflows. Section 4.4 explains the methodology used during this chapter and Section 4.5 evaluates Flexagon. Finally, Section 4.6 describes the related work and Section 4.7 concludes this chapter.

4.2 Background

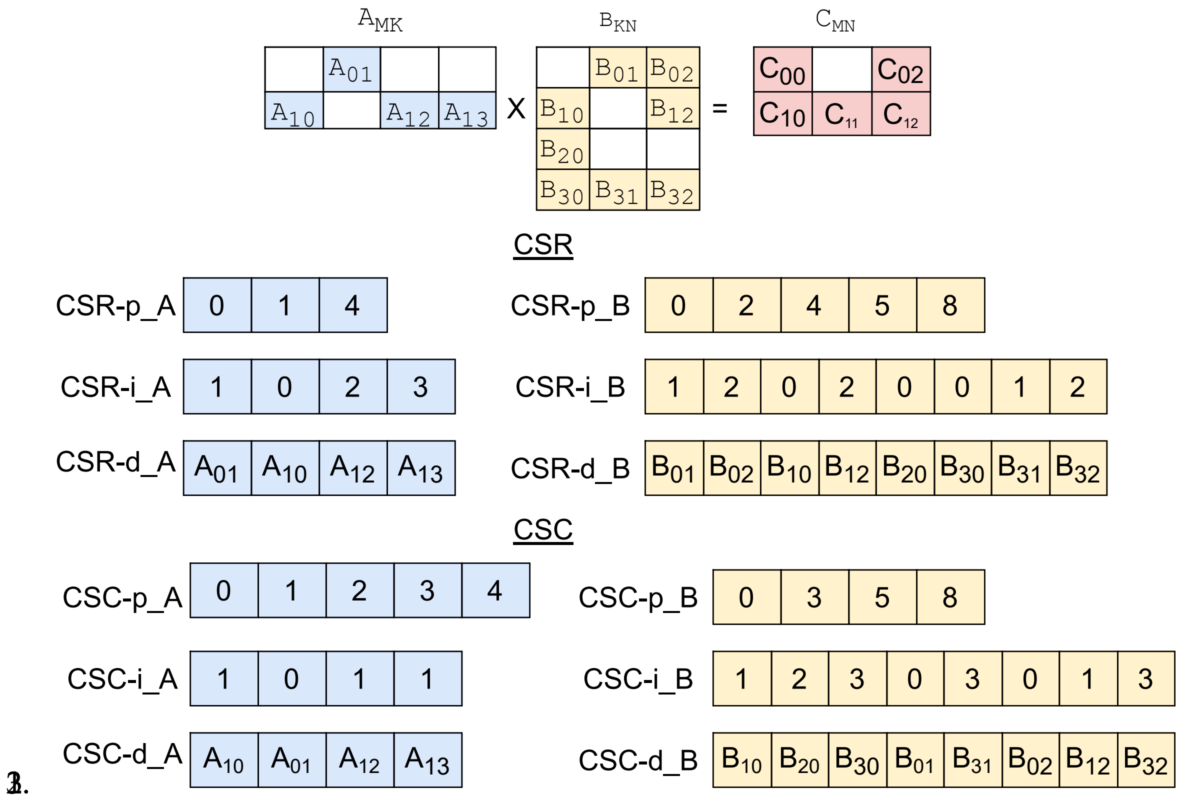
4.2.1 Compression Formats

Following the same taxonomy used in ExTensor [50], the SpMSpM operation computes the operation $C_{M,N} = A_{M,K} \times B_{K,N}$, where the three matrices are 2-dimensional tensors. Since these matrices are typically sparse (see Table 4.2), they are compressed to encode the non-zero values while preserving the computation intact (lossless compression) [64]. In our work, we focus on the widely used unstructured compression formats CSR and CSC. Fig. 4.2 shows an example. A matrix encoded in CSR format employs three 1-dimensional tensors to store the non-zero values in a row-major data layout: a data vector to represent the non-zero values, a row pointer vector to store the index position where each row begins within the data vector, and a column index vector to store the column of each non-zero value. Similarly, the CSC uses column-major data layout: a data vector, a column pointer vector to store the index position of start of column, and a row index vector to store the row index of each non-zero data value. Observe that both CSR and CSC employ the same compression method, and thus, can be seen as a single compression format. This is important as an accelerator would use the same control logic needed to handle both of them. This facilitates the implementation of the control logic (further details in Section 4.3.5) in our accelerator.

As in previous works (e.g. [122]), we will use the term *fiber* to denote each compressed row or column. Each fiber contains a list of duples (coordinate, value), sorted by coordinate. We use the term *element* to refer to one duple in the fiber.

4.2.2 SpMSpM Dataflows

SpMSpM operation is based on a triple-nested for-loop that iterates over `A`'s and `B`'s independent dimensions M and N, and co-iterates over their shared dimension K. Depending upon the level of the co-iteration in the loop nesting, *three different dataflows* have been identified for SpMSpM computation: `Inner Product` (co-iteration at the innermost loop), `Outer Product` (co-iteration at the outermost loop) and `Gustavson's` (co-iteration at the middle loop). Additionally, these dataflows result in *six possible variants* according to how the independent dimensions (M and N) are ordered for each of them (two variants per dataflow). Notice that each variant favors the stationarity of one of the dimensions (the outermost one) over the other. This way, we distinguish each variant by `(M)` if the computation is M-stationary or `(N)` if it is N-stationary. Fig. 4.3 shows the resulting six dataflow variants. Each dataflow defines how the elements flow during execution, and thus, the opportunities for data reuse. Table 4.3 gives a detailed taxonomy of each approach, which we summarize as follows:



A_{MK}
B_{KN}
C_{MN}
A_{01} A_{10} A_{12} A_{13}
X
B_{01} B_{02} B_{10} B_{12} B_{20} B_{30} B_{31} B_{32}
=
C_{00} C_{02} C_{10} C_{11} C_{12}
CSR
CSR-p_A 0 1 4
CSR-i_A 1 0 2 3
CSR-d_A A_{01} A_{10} A_{12} A_{13}
CSR-p_B 0 2 4 5 8
CSR-i_B 1 2 0 2 0 0 1 2
CSR-d_B B_{01} B_{02} B_{10} B_{12} B_{20} B_{30} B_{31} B_{32}
CSC
CSC-p_A 0 1 2 3 4
CSC-i_A 1 0 1 1
CSC-d_A A_{10} A_{01} A_{12} A_{13}
CSC-p_B 0 3 5 8
CSC-i_B 1 2 3 0 3 0 1 3
CSC-d_B B_{10} B_{20} B_{30} B_{01} B_{31} B_{02} B_{12} B_{32}
3.

Figure 4.2: Example of an SpMSpM operation.

Dataflow	**Informal Name**	**Sta**	**Sta**	**Str**	**A format**	**B format**	**C format**
	Tensor	**Fiber**	**Tensor**				
MNK	Inner Product(M)	C	A	B	CSR	CSC	CSR
KMN	Outer Product(M)	A	B	C	CSC	CSR	CSR
MKN	Gustavson's(M)	A	C	B	CSR	CSR	CSR
NMK	Inner Product(N)	C	B	A	CSR	CSC	CSC
KNM	Outer Product(N)	B	A	C	CSC	CSR	CSC
NKM	Gustavson's(N)	B	C	A	CSC	CSC	CSC

Table 4.3: Taxonomy of dataflow properties. Traversal order is given outermost-to-innermost in loop order. The stationary tensor gets the most reuse, whereas the streaming tensor has long temporal reuse distance that is difficult to exploit, while the stationary fiber falls in between.

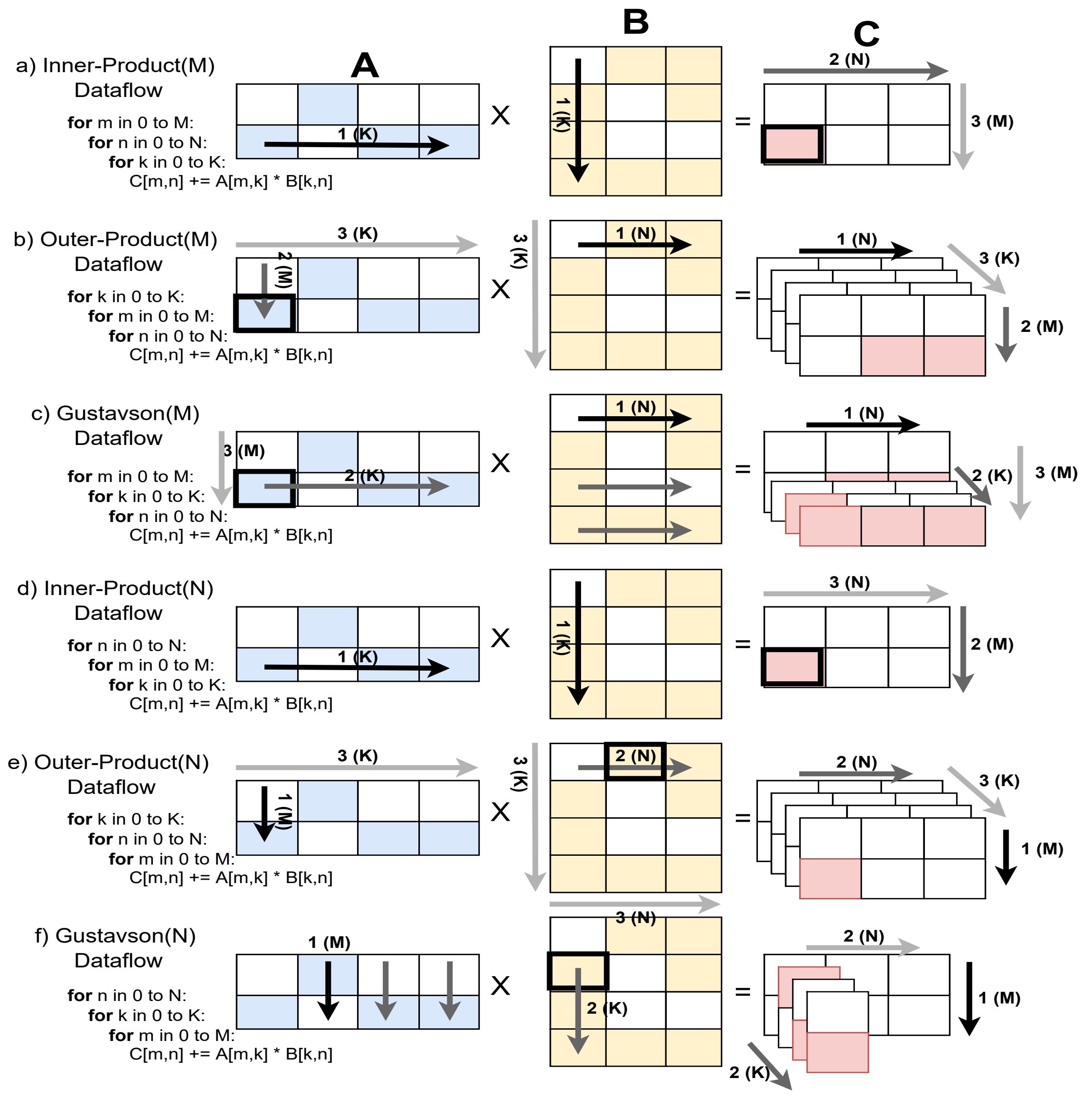


Figure 4.3: Dataflow combinations for matrix multiplication. For simplicity, non-compressed (dense) matrices are shown.

4.2.2.1 Inner Product (IP)

The order in which the *M* and *N* dimensions are traversed defines the dataflow and the order in which the outputs are being generated. If the *M* dimension is at the outermost loop –`Inner Product(M)` dataflow– (see Fig. 4.3a), the output elements are generated by rows. On the contrary, the `Inner Product(N)`dataflow (see Fig. 4.3d) generates the outputs by columns as the *N* dimension is kept stationary the longest. These two dataflows obtain good reuse for matrix `C` as the partial outputs are kept stationary, but achieves poor reuse for matrices `A` and `B`. Note that, when it comes to compressed matrices, these dataflows require to perform *intersections* to locate the matching *K* coordinates, thereby leading to ineffectual computation. Additionally, in order to traverse the `A` and `B` matrices efficiently in the correct order, matrix `A` needs to be encoded in CSR format (a row-major data layout is more efficient) while matrix `B` needs to be encoded in CSC (a column-major data layout is more efficient). Note that, the encoding of the output matrix `C` depends on the *M* and *N* iteration orders. As shown in Fig. 4.3(a), the `Inner Product(M)` dataflow generates the outputs in an order that is best suited for a row-major data layout, hence a CSR format is more efficient to traverse, while the `Inner Product(N)` dataflow generates them in a column-major data layout that is more appropriate for the CSC format.

4.2.2.2 Outer Product (OP)

This dataflow keeps the co-iteration at the outermost loop. This means that every *M* and *N* traversals generates a partial matrix which is accumulated with the rest of partial matrices (i.e., *K* partial matrices are accumulated) to produce the output `C` matrix. The `Outer Product(M)` dataflow (shown in Fig. 4.3b) generates the partial matrices by keeping the *M*-dimension stationary and traversing the *N*-dimension (i.e., by rows), generating the partial sums (psums) in this order (i.e., by rows). In contrast, the `Outer Product(N)` dataflow (see Fig. 4.3e) exchanges the *N*- and *M*-dimensions, keeping the *N* dimension stationary and traversing the *M* dimension (i.e., by columns). These dataflows achieve good reuse for matrices `A` and `B`, but very poor reuse for matrix `C`. Note that when it comes to sparse computation, the number of partial sums and their coordinates are not known a priori, meaning that they require to be merged after the generation process.

The `Outer Product` dataflow requires to traverse matrix `A` by columns and matrix `B` by rows, which must be encoded in CSC and CSR formats, respectively. Matrix `C` can be generated either in CSR or CSC depending on the outermost independent dimension (i.e., *M* or *N*). As we can see in Fig. 4.3(c), the `Outer Product` dataflow generates the outputs in row-major data layout, which is more convenient to be formatted as CSR, while the `Outer Product(N)` dataflow generates the elements of the output matrix in a data layout that is more efficient for CSC encoding (see Fig. 4.3(d)).

4.2.2.3 Gustavson's (Gust)

In this dataflow, the co-iteration happens at the middle of the triple-nested loop. The `Gustavson's(M)` dataflow (see Fig. 4.3c) (a.k.a., row-wise product) computes one entire row at a time, by traversing a row of `A` and linearly combining the rows of `B` for which the row of `A` has non-zero coordinates. Similarly, the `Gustavson's(N)` dataflow (i.e., column-wise product) shown in Fig. 4.3f computes one entire column at a time, by traversing a column of `B` and linearly combining the columns of `A` for which the column of `B` has non-zero coordinates.

Similar to the `Outer Product` dataflow, `Gustavson's` also requires merging partial sums when the matrices are compressed.

Nevertheless, this dataflow still achieves good reuse for the matrix that is kept stationary (i.e., `A` in the `Gustavson's(M)` dataflow and `B` in the `Gustavson's(N)` dataflow), and reduces significantly the number of partial outputs generated with respect to the `Outer-Product` algorithm, improving the reuse for matrix `C`. However, this dataflow also comes with its own inconveniences. First, the number of partial sums generated in a particular row or column can be still high, introducing pressure on the memory hierarchy for producing matrix `C`. And second, the reuse for matrix `B` in the `Gustavson's(M)` dataflow depends on the particular values of matrix `A`, as its coordinates are used to index matrix `B`. This causes that sometimes the rows of `B` to be fetched are too far from each other, producing poor reuse among rows and causing a degradation on the energy efficiency. The same happens in the `Gustavson's(N)` dataflow for matrix `A`, exchanging the M and N dimensions.

Finally, `Gustavson's` requires different matrix encoding for both dataflows. `Gustavson's(M)` traverses both matrices `A` and `B` by rows, and also generates matrix `C` by rows, requiring the CSR format for the three matrices. In contrast, `Gustavson's(N)` requires CSC encoding for the three matrices as it performs column-wise processing.

For the rest of the chapter, we will pedagogically use M-stationary dataflows during the explanations, although everything would apply for the N-stationary dataflows as well.

4.3 Flexagon Design

Fig. 4.4a shows a high-level overview of the architecture of the Flexagon accelerator. As observed, Flexagon consists of a set of multipliers, adders and comparators, as well as three on-chip SRAM modules specifically tailored to

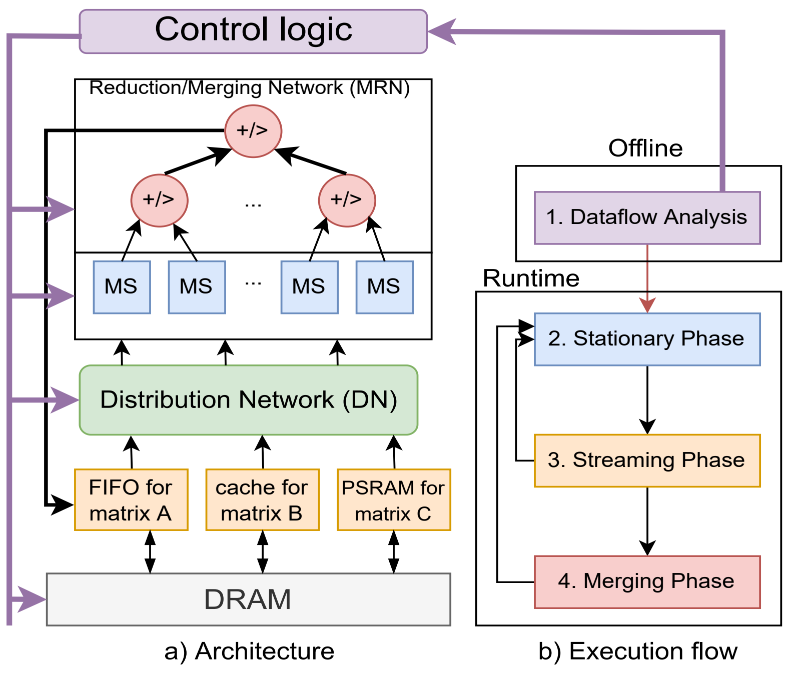


Figure 4.4: Flexagon high level overview.

the storage needs of matrices `A`, `B` and `C` for the three SpMSpM dataflows. In addition, in order to allow for the highest flexibility, all the on-chip components are interconnected by using a general three-tier reconfigurable network-on-chip (NoC) composed of a Distribution Network (DN), a Multiplier Network (MN), and a Merger-Reduction Network (MRN), inspired by the taxonomy of on-chip communication flows within AI accelerators [70]. These components are controlled by the control unit which is configured by the mapper/compiler before the execution.

Flexagon's execution phases are shown in Fig. 4.4b. The process begins with a dataflow analysis (phase 1), which is carried out offline. Here, a mapper/compiler examines the features of the SpMSpM operation to be executed (i.e., matrix dimensions and sparsity patterns) and decides the dataflow (between the six available described in Section 4.2) that best matches the operation, generating the tiling scheme and the particular values for the signals that configure the operation of the accelerator for the rest of the phases. The next three phases are performed during runtime according to these generated signals and are repeated several times according to the number of execution tiles. The first runtime phase is called **stationary phase** (phase 2), which delivers data that will be kept stationary in the multipliers to reduce the numer of costly memory accesses. According to the dataflows description presented in Section 4.2 for M-stationary dataflows, this stationary data belongs to matrix `A`, while matrix `B` is streamed during the **streaming phase** (phase 3). For N-stationary dataflows, this happens in the reverse order. These two phases generalize for the three dataflows. The **merging phase** (phase 4) is only necessary for both `Outer Product` and `Gustavson's` dataflows and is the one in charge of merging the fibers of partial sums that have been previously generated during the streaming phase. This phase is skipped in the `Inner Product` dataflow as no merging is required.

In this work, we focus our attention on the accelerator design as well as on the way the three phases operate in order to give support to the six possible dataflows (three SpMSpM dataflows, two variants, M or N-stationary, each) over the same hardware substrate. We leave the study of the tool required for dataflow analysis, tiling selection and generation of the configuration file for the accelerator (phase 1 in the Offline part in Fig. 4.4b) for future work.

4.3.1 On-chip Networks

One of the main novelties of Flexagon is its ability (through proper configuration) to support the six dataflows described in Section 4.2 using the same hardware substrate. To do so, the accelerator follows the three-tier configurable NoC taxonomy described in Chapter 2. In this way, it is equiped with three-tier subnetworks able to adapt to the communication features of each dataflow. Next, we describe each subnetwork in detail.

4.3.1.1 Distribution Network (DN)

This module is used to deliver data from the SRAM structures to the multipliers. In order to enable the high flexibility that the three SpMSpM dataflows require, the DN needs to support unicast, multicast and broadcast data delivery. To achieve this, and at the same time ensure high energy efficiency, we utilize a Benes network similar to previous designs like SIGMA [31]. This network is an N-input, N-output non-blocking topology with $2 \times log(N) + 1$ levels, each with N tiny 2×2 switches.

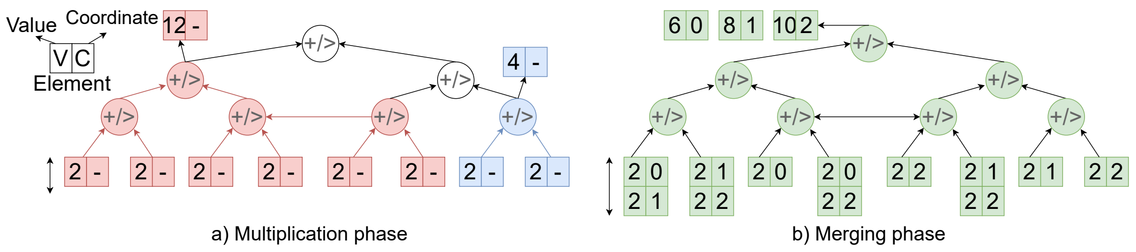


Figure 4.5: a) MRN in the multiplying phase. b) MRN in the merging phase.

4.3.1.2 Merger-Reduction Network (MRN)

Previous designs like MAERI [70] or SIGMA [31] have used specialized tree-based reduction networks (RNs) such as ART or FAN to enable non-blocking reduction of multiple clusters of psums. These RNs provide high flexibility for the `Inner Product` dataflow as its purpose is to reduce a cluster of psums. In case of `Outer Product` and `Gustavson's` dataflows, other works such as [122, 124] employ a tree-based topology to perform the merge operation of the psums once they are generated. In our design, we have, for the first time, unified this concept, and have designed a merger-reduction network able to both reduce and merge psums. Figure 4.5 shows a 8-wide MRN operating on both multiplying (Figure 4.5a) and merging (Figure 4.5b) phases. The multiplication phase shows 2 clusters of multipliers reducing in parallel. The first cluster is shaped by 6 multipliers (i.e., colored by red) and the second one is composed of two (i.e., colored by blue). As we can see, the role of the nodes is to add the value of each element into a resulting value that is sent to the parent. Here, the coordinate field is not needed so it is not used. In contrast, the merging phase shows 8 duples of value (i.e., the partial sum) and coordinate being merged into four final elements. As we can see, our MRN employs an augmented tree with forwarding links between nodes that do not share the parent. This helps in the multiplication phase to create multiple non-blocking clusters of multipliers reducing in parallel. Also, the MRN topology augments the nodes with comparators and switching logic able to exchange the mode of operation. An example of the microarchitecture of these nodes is shown in Figure 4.6 This allows to perform both operations while keeping low area and power logic and enables direct support for the three SpMSpM dataflows, as we describe later. The selection of the configuration is done by the mapper/compiler which generates the signals to route the nodes and its operation modes accordingly to the dataflow and layer dimensions.

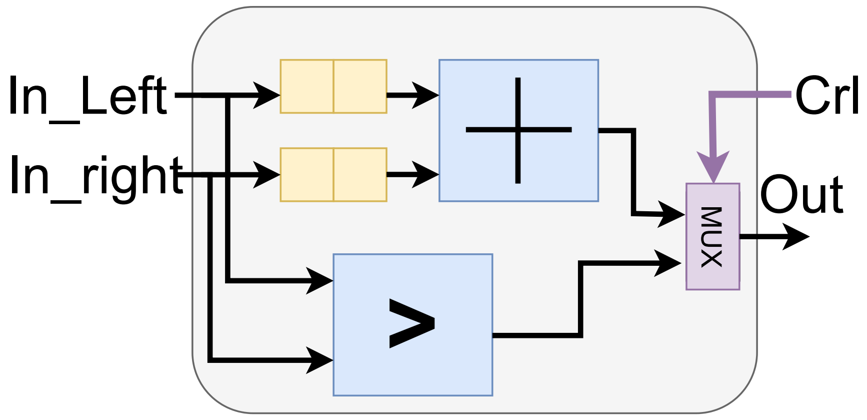


Figure 4.6: Microarchitecture of MRN's nodes.

Multiplier network (MN): Similar to other designs such as MAERI, this network is composed of independent multipliers that can operate in two different modes: i) *Multiplier mode*: the unit performs a multiplication and sends the result to the MRN. This mode is used during the entire execution when the `Inner Product` dataflow is configured, and during the streaming phase when either the `Outer Product` or `Gustavson's` dataflows are configured. ii) *Forwarder mode*: the multiplier forwards directly the input, which is typically a psum, to the MRN. As we will clarify in the examples bellow, this mode is essentially configured during the merging phase in both the `Outer Product` and `Gustavson's` dataflows.

4.3.2 Walk-Through Examples

4.3.2.1 Example of Inner-Product Dataflow

We use the `Inner Product(M)` dataflow, but the `Inner Product(N)` could be executed in the same manner by exchanging matrices `A` and `B`. To ease the explanation, as illustrated in Fig. 4.7, we assume a simple 4-multiplier accelerator, and we walk through the activity of the three sub-networks. In the explanation, we will mention the on-chip SRAM modules needed for storing matrices `A`, `B`, `C` and psums (see the yellow boxes in Fig. 4.4b). Section 4.3.4 provides an in-depth description of these memory structures.

Stationary phase: First, during the stationary phase, the controller maps as many fibers of matrix `A` as possible, reading all the elements sequentially from the dedicated SRAM structure called FIFO for matrix `A`. Each cluster of multipliers will perform the dot product operation The values (without coordinates) are

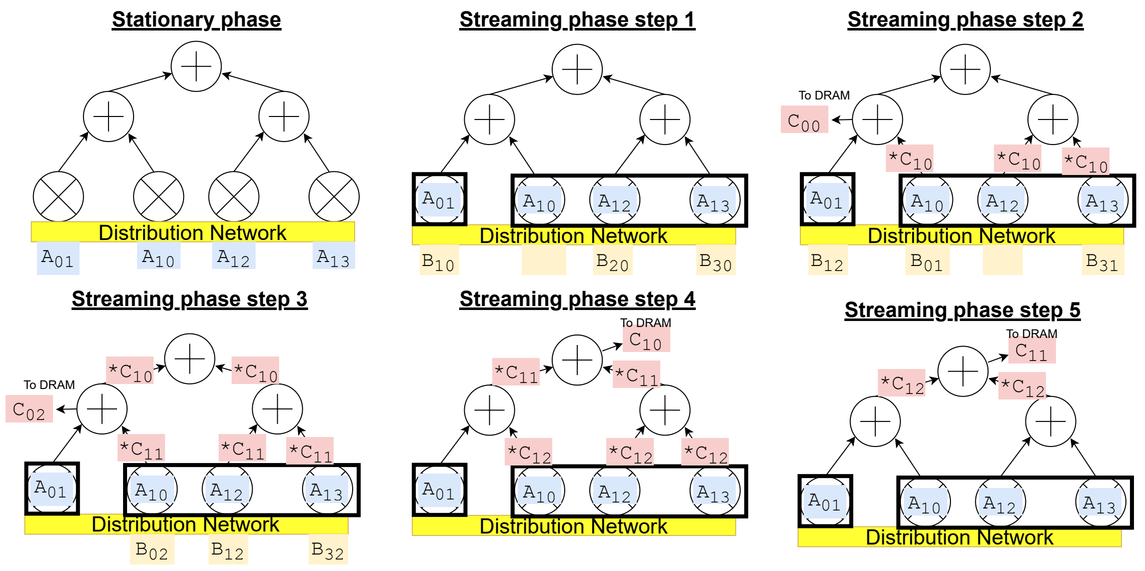


Figure 4.7: Example of Flexagon running SpMSpM using an `Inner-Product(M)` dataflow.

kept spatially in the multipliers and will be reused during each step during the streaming phase. This way, each cluster of multipliers will perform the dot product operation.

Streaming phase: After filling the multipliers with the fibers of `A`, the controller multicasts each fiber of matrix `B` (i.e., each column) to the configured clusters in the MN. To do so, the controller uses the row coordinate of each element in the fiber of `B` to detect whether it intersects with the column coordinate in the fiber of `A`. If this happens, the value is sent out to the corresponding multiplier. As we can see, in the example, all the psums generated by the multipliers are reduced using the MRN, producing the final output at the top of each sub-tree and sending it out directly to memory.

4.3.2.2 Example of Outer-Product Dataflow

Fig. 4.8 shows the same example as before but now assuming the `Outer Product(M)` dataflow. We also show the customized SRAM structure for `C` called PSRAM and that is utilized for storing the psums for matrix `C`. This structure is further described in Section 4.3.4 and stores blocks of elements (coordinate, value).

Stationary phase: During the stationary phase, the fibers of matrix `A` (i.e., columns of `A`) are delivered to the multipliers sequentially following the CSC compression format. In our particular case, the four multipliers store the elements $A_{1,0}$, $A_{0,1}$, $A_{1,2}$, and $A_{1,3}$.

Streaming phase: During the streaming phase, each multiplier keeps stationary an element $A_{m,k}$, given m in range [0,M) and k in range [0,K), in order to linearly combine the non-zero elements $B_{k,0-N}$, generating a psum fiber where all the elements share the row (m) and a particular k iteration (i.e., the partial matrix where these elements belong to). Consecutive multipliers generating psums for different rows for the same k iteration, do not need the psum to be merged together. Thus, the generated psums must be sent out to the SRAM structure, in order to be merged in a third phase. Also, since multiple rows can run in parallel, the PSRAM 's set is indexed by rows. Furthermore, since the number of non-zeros in matrix A is not known a priori, it might happen that multiple fibers from matrix A may fit in a single iteration, causing that multiple partial outputs for the same row, but for different k iterations may run in parallel. Since the number of psums for a particular row and for a particular k iteration is not known at runtime, we must assign static space in the PSRAM to store the psums from different k iterations that may be running in parallel and storing in the same row. To do so, we divide each row in the PSRAM in blocks, and each block contains a valid bit to indicate the validity of the data, a k value, indicating the k iteration that belongs to that group of partial sums and the block of data. By doing this, each block can hold, at a particular time, psums for different k iterations for a particular row. This way, if the number of psums for a particular iteration exceeds the block size, it may use another block from the row, even if the next block is already being used by another k iteration. The details about the organization and operation of the PSRAM are given in Section 4.3.4. In the example in Fig. 4.8, we see three steps regarding the streaming phase. In the first step, the controller sends the first element of the four fibers (across the K-dimension) to its corresponding multiplier. For example, the first multiplier which keeps stationary the element $A_{1,0}$ receives the first element of the fiber for the row (i.e., iteration k) 0. In step 2, each multiplier generates a psum (indicated by the symbol *), which is the first element for the 4 fibers generated across the K-dimension. These psums are then stored in the PSRAM . The first psum $*C_{1,1}$ is allocated in the set 1, as it is indexed by its row coordinate. Use of sets allows us to execute multiple rows in parallel. Then, since the first line is free, the psum is stored there, enabling the valid bit and indicating that the element belongs to K0. Dividing rows into blocks allows holding psums corresponding to different K for a particular row. The second psum, $*C_{0,0}$ is allocated to the set 0 (its row coordinate) and since the first line is here, the cache enables the valid bit and tags the line with K1. The last two elements share coordinates (i.e., s $*C_{1,0}$), but belong to a different partial matrix (K2 and K3). These two elements go to the same set in the PSRAM but to different lines, each tagged with its iteration k (i.e., K2 and K3). This allows to locate the psum fibers in the correct order during the merging phase.

In step 3, the second element for the four fibers are produced, following the same execution scheme. For the sake of brevity, we do not show how the last element from the longest psum fiber (i.e., fiber K3) is produced, and directly show the contents of the PSRAM just before starting the merging phase (merging phase step 1).

Merging phase: The merging phase proceeds row by row. For each row, the controller fetches the elements for the different k-iteration fibers from the PSRAM . These elements are stored in different blocks and can be identified by their tags, consuming the elements and sending them to the MRN in order to be merged. Each unit in the MRN compares the column coordinate (i.e., the N-dimension). If the coordinates match, then the values of the elements are accumulated. Otherwise, the node sends up the tree the element with the lowest coordinate. The last two rows in Fig. 4.8 show 8 merging steps. The 4 first steps (Merging phase step 1 to step 4) merge the first row. In the second row, there are 3 psum fibers ready to be merged. In step 5, the first element for the three fibers (K0, K2 and K3) are sent to the MRN. In step 6, the psums $*C_{1,1}$ and $*C_{1,0}$ compare their column coordinate. Since they do not match, and element $*C_{1,0}$ has a lowest column coordinate, this element is sent up to the MRN first. The same procedure is executed in a pipelined-manner for the rest of the elements in the fiber until all the psums have been merged in a single fiber and sent to DRAM. In case the number of fibers in a row is greater than the number of multipliers (i.e., leaves in the tree), the controller needs to perform multiple passes to complete the final merge.

4.3.2.3 Example of Gustavson's Dataflow

Finally, for the same example matrices, Fig. 4.9 illustrates how Flexagon proceeds when the `Gustavson's(M)` dataflow is selected. Similarly, the operation in this case proceeds in three well-differentiated phases.

Stationary phase: First, during the stationary phase, as many fibers of `A` (i.e., rows in matrix `A`) as possible are mapped spatially and sequentially in the multipliers. This makes this phase similar to the stationary phase for inner-product shown in Fig. 4.7, where the `A` matrix is delivered in CSR traversal order.

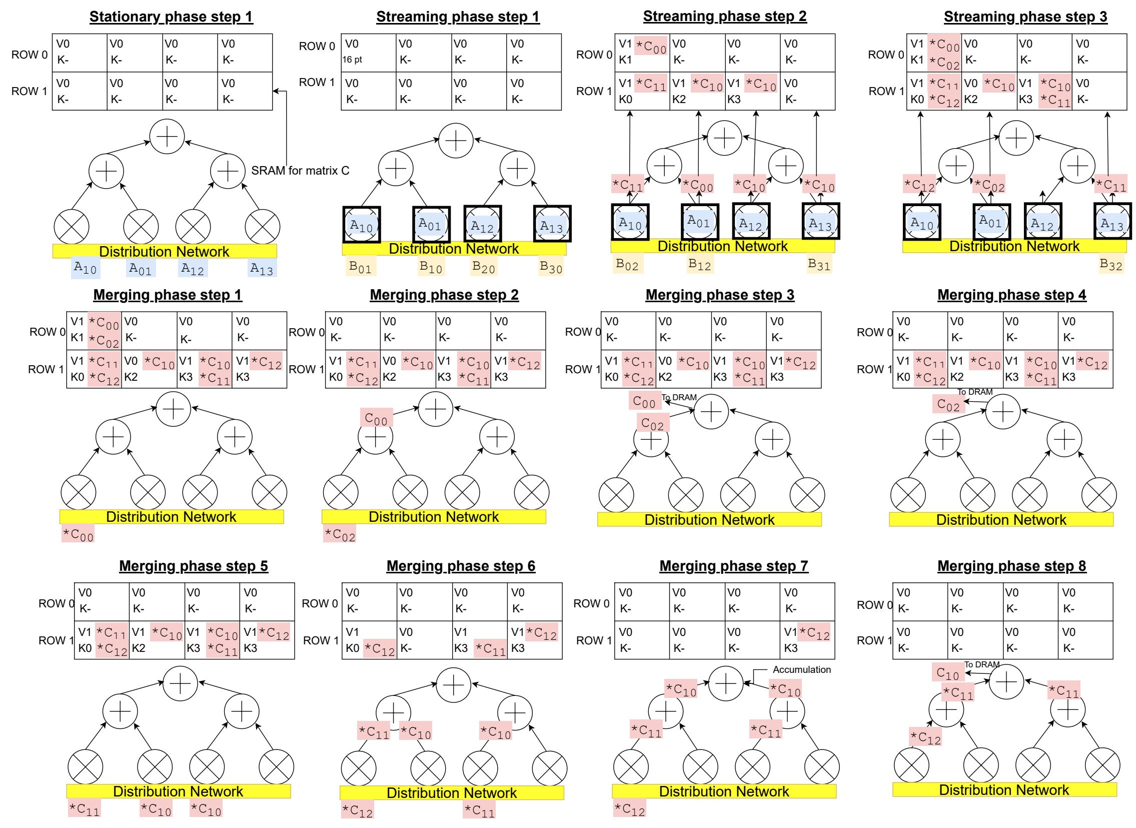


Figure 4.8: Example of Flexagon running SpMSpM using an `Outer-Product(M)` dataflow. “*” indicates that the outputs produced by the accelerator are psums and not final outputs. *“V”* in the PSRAM represents the valid bit and *“K”* indicates the *k* iteration tagged for a particular block.

The multipliers, then keep two clusters, each in charge of calculating the psums for a different output row (i.e., rows 0 and 1 in the example).

Streaming phase: In the streaming phase, for each multiplier, the memory controller fetches and delivers the fiber of `B` (i.e., row of `B`) that corresponds to the column coordinate (i.e., *k*-iteration) associated to the mapped element of `A` in the multiplier. Every multiplier generates a partial output fiber which is merged with the rest of partial output fibers generated by the other multipliers allocated to the same fiber of `A`. An example of this generation is shown in Fig. 4.9. Here, we depict 6 streaming steps. The first multiplier keeps stationary the only one element in matrix `A` ($A_{0,1}$) so it receives the fiber of `B` indexed by the column 1 (i.e., the row 1). The multipliers 2-4 keep the elements $A_{1,0}$, $A_{1,2}$ and $A_{1,3}$ so they

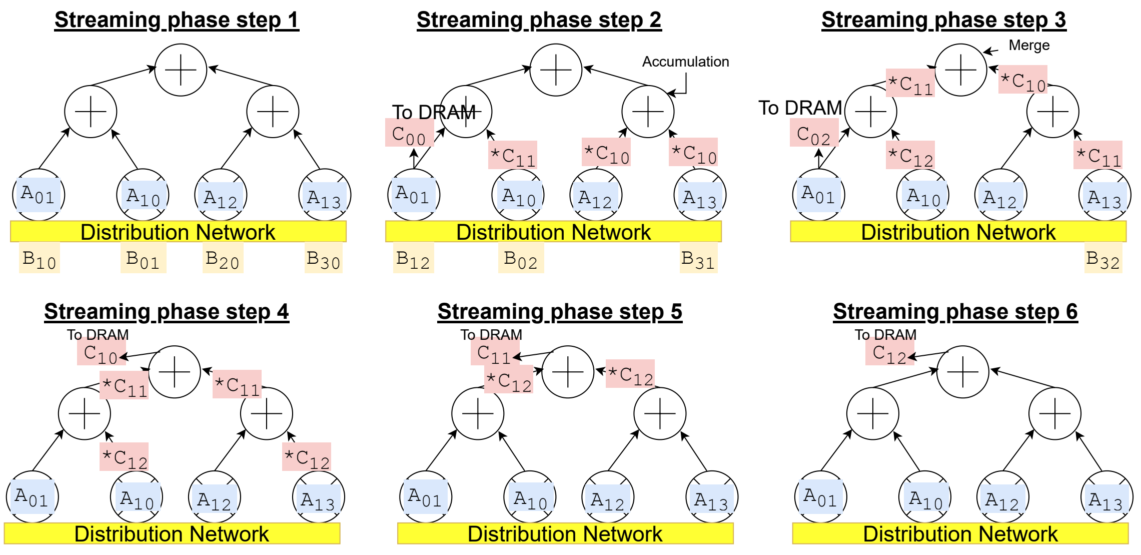


Figure 4.9: Example of Flexagon running SpMSpM using the `Gustavson(M)` dataflow. "*" indicates that the outputs produced by the accelerator are psums and not final outputs.

receive the fibers of `B` 0, 2 and 3, respectively. The first 3 steps show how the elements from the fibers of `B` are delivered cycle by cycle.

Merging phase: Similar to the `Outer Product` dataflow, the merging phase combines both the accumulation and the merging operation, accumulating the elements (i.e., its values) in a certain node if their column coordinates match, or sending the element with the lowest column coordinate value. On the other hand, in `Gustavson's` dataflow, we can merge the psums immediately after their generation, as a cluster of multipliers always generates fibers for the same row, but for different k iterations. When the number of elements in `A` fits into a cluster of multipliers, the output fiber generated by that cluster will be a final fiber, and the outputs can be sent directly to DRAM without being stored in the SRAM. Otherwise, when the number of elements in `A` exceeds the number of multipliers, the output fiber will be a partial fiber as multiple iterations are required, and therefore the fiber will require to be stored in the PSRAM, similar to what happens in the `Outer Product` dataflow.

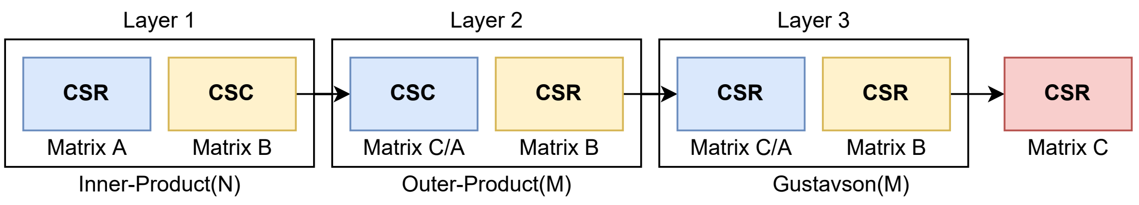


Figure 4.10: Example of three DNN layers being executed by running the combination `Inner-Product`, `Outer-Product` and `Gustavson` dataflows.

4.3.3 Combinations of Inter-Layer Dataflows

As Table 4.3 shows, M-stationary dataflows output the elements in CSR format while N-stationary dataflows output the elements in CSC format. Flexagon supports the six dataflows and takes advantage of this observation to appropriately execute every possible sequence of DNN layers without requiring costly explicit hardware format conversions and is the first work to support compressed outputs without explicit conversions. Fig. 4.10 shows an example of a DNN composed of three layers, demonstrating this feature. The first and the second layer are configured to execute inner and outer products respectively. Since second layer needs activation in CSC, first layer is Inner Product (N). The weights are assumed to be stored offline in both formats. The second layer produces the matrix in CSR format if it uses M-stationary. As a result it could choose from inner product or Gustavsons(M). The first layer (i.e., the DNN input) uses matrix `A` (i.e., the input activations) in CSR format. This layer is selected by the compiler to be executed by using an `Inner-Product` dataflow, which requires the weights to be fetched in CSC format. In contrast, the second layer (i.e., Layer 2) is selected to be executed with an `Outer-Product` dataflow. According to this, the dataflow selected to execute the first layer has to be `Inner-Product(N)` as it directly generates matrix `C` in the format required for the second layer (CSC format). The second layer is selected to be executed using the `Outer-Product(M)` dataflow, for which matrix B needs to be compressed in a CSR format. However, being matrix `B` the weight matrix, it could have been pre-processed offline and stored in both formats. This dataflow generates matrix `C` in CSR format, which forces the next layer to utilize either `Inner Product(M)`, or `Inner Product(N)`, or `Gustavson's(M)`dataflows (see Table 4.3). In this case, the compiler selects the use of `Gustavson's(M)`, generating the final output in CSR format.

Table 4.4 shows the transitions for each dataflow combination that do not

	IP(M)	*OP(M)*	*Gust(M)*	*IP(N)*	*OP(N)*	*Gust(N)*
IP(M)	✓	EC	✓	✓	EC	EC
OP(M)	✓	EC	✓	✓	EC	EC
Gust(M)	✓	EC	✓	✓	EC	EC
IP(N)	EC	✓	EC	EC	✓	✓
OP(N)	EC	✓	EC	EC	✓	✓
Gust(N)	EC	✓	EC	EC	✓	✓

Table 4.4: Dataflow transitions allowed without requiring Explicit format Conversion (EC). Different rows represent the different outputs of the first layer and different columns represent the corresponding input to the second layer.

require an explicit format conversion (green tick) and those that do (*Explicit Conversion* or EC). As we can see, exploitation of the six possible dataflows enables direct combinations of the three SpMSpM dataflows. For example, we can always transition from/to any `Inner Product` dataflow to/from any `Outer Product` dataflow just by modifying the order of the loops *M* and *N*. Transitioning from any `Gustavson's` dataflow to any `Inner` or `Outer Product` is more complicated as it may require an intermediate transition step. For example, transitioning from `Gustavson's(M)`to `Outer Product(M)`or `Outer Product(N)`requires an intermediate layer processed through `Inner Product(M)` as exchanging the loop order to `Gustavson's(N)` is not possible due it requires the input matrices in a different format. These combinations can utilized by the mapper/compiler to generate the best sequence of dataflows that lead to the best performance and energy efficiency for a particular DNN execution.

4.3.4 Memory Organization

In order to capture all dataflows, we have designed a customized L1 memory level specifically tailored for the common and different patterns among the three dataflows. Fig. 4.11 shows a schematic design for this L1 memory level. We use a separate memory structure and a different buffer idiom for data movement from/to each structure. To do so, every memory structure is operated by two controllers, the **tile filler** interfacing with the DRAM, and the **tile reader** interfacing the datapath of the accelerator (i.e., the multipliers). Next, we describe each memory structure in detail:

Memory structure for the stationary matrix (FIFO): The elements of the stationary matrix are always read once and sequentially for the three dataflows, as they are kept stationary in the multipliers. To hide the access latency, we

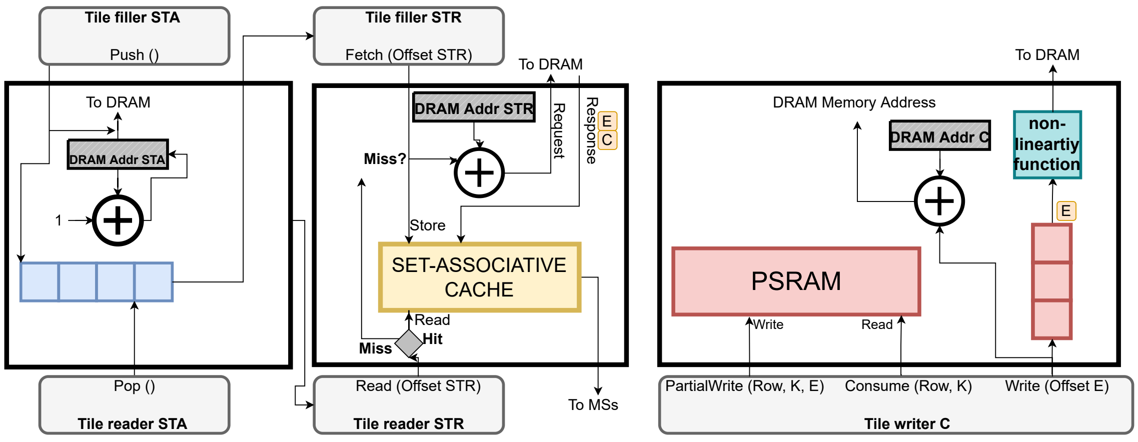


Figure 4.11: Memory structures in Flexagon.

implement a 512-byte read-only FIFO. In order to save bandwidth and reduce the complexity: (1) the memory structure keeps the DRAM location of the stationary matrix in a register, so that the fibres are pushed implicitly into FIFO, (2) we employ a single-port for read and write.

Memory structure for the streaming matrix (Cache): The streaming matrix presents a more heterogeneous memory access pattern. In `Inner Product`, every stationary-phase (i.e., every iteration) causes the streaming of the entire matrix. In other words, there is significant spatial locality and temporal locality every time the matrix is re-loaded. In the `Outer Product` dataflow, the fibers of the streaming matrix are read just once and sequentially. In `Gustavson's` dataflow, every fiber of the stationary matrix gathers F fibers of the streaming matrix, F being the number of non-zero elements in the fiber of the stationary matrix which are typically scattered all over the matrix, causing an irregular and unpredictable memory access pattern. To factor the worst-case `Gustavson's` dataflow, we implement the memory structure for the streaming matrix as a traditional read-only set-associative cache. However, we implement this cache to operate on a virtual address space relative to the beginning of the streaming matrix, which let us use shorter memory addresses and therefore save bandwidth and reduce the tag lengths.

Memory structure for matrix C (PSRAM): To store the psums, we have designed a new buffer idiom called *PSRAM* , which is used for both `Outer Product` and `Gustavson's` dataflows. Fig. 4.8 shows the way this memory structure works, Fig. 4.11 shows a high-level diagram, and Fig. 4.12 delves into detail.

The memory is organized into sets corresponding to different rows and each set into blocks for different K dimension within a row. Each block has a valid bit. Besides, we use a register as a line tag to keep the column coordinate (i.e., the *k*-iteration) assigned to that line. Since the length of the output fiber is undetermined, it may occupy several (and non-consecutive) lines in the same row. This is essentially a way-combining scheme tagged by the *k*-iteration. The register is used by the row to locate whether a certain output fiber is still placed in the PSRAM. In order to read several fibers in parallel from the same set (i.e., to merge a particular row or column) we implement a multi-bank scheme organized across the lines within a set. Finally, we also include two registers to keep the byte location where the first and last elements are in the line. This metadata is used by the next two operations which are managed by the controller **tile writer C**:

PartialWrite(*row, k, E*): This operation is used to place an element in the PSRAM. The logic, indexes the element by the *row* argument and then searches in parallel the line where is being stored the output fiber with the *k* identifier. If the output fiber exists (i.e., the *k* tags match), the PSRAM places the new element *E* into the last available position (indicated by the register *Last* in the metadata) of the last line. If the fiber does not exist, the logic searches the first available line and stores the element *E* in the first position of the line, enabling the valid bit and updating the *K*, *First* and *Last* registers in order to continue storing elements for the same *K* identifier in future accesses.

Consume(*Row, k*): The elements within a partial output fiber are placed in the PSRAM temporarily. They are read once to feed the accelerator and are no longer used again. This allows us to incorporate the **consume** operation, which reads and erases a particular element from the memory structure. In particular, the controller merges the partial output fibers row by row. To do so, the controller needs to read as many fibers as possible for the same row and for each fiber it uses the **consume** operation indicating the *row* and the fiber *k* to search. If there is an active line keeping the *k* fiber, the structure reads the next element from that fiber (indicated by the register *First*) and consumes it by increasing by one element this register. When the *First* and *Last* registers store the same value, the PSRAM detects that the line has been consumed and invalidates the line by setting the valid bit to 0.

Write(Offset, E): Apart from the PSRAM which is used to store partial output fibers, we also augment our memory structure with a FIFO which is used as a write buffer to hide the latency of sending out final output fibers to DRAM. This structure is employed by the **Write** operation which receives the location of the

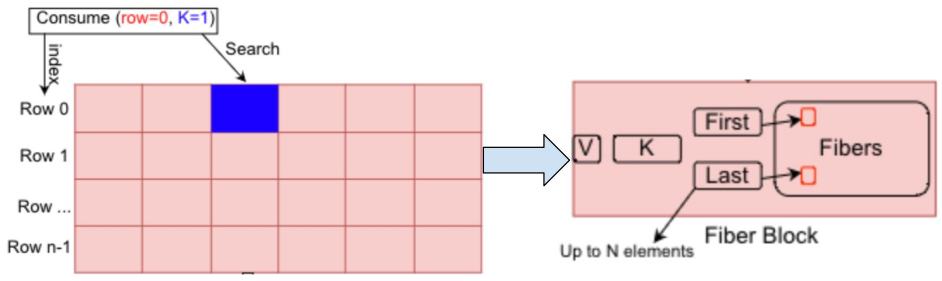


Figure 4.12: PSRAM overview.

element in matrix C to be stored (i.e., address offset from the beginning of matrix C) and the element *E*. We also place a look-up table between the write buffer and the DRAM to apply the non-linearity function on the output elements.

4.3.5 Memory Controllers

As we have stated in Section 4.3.4, we employ an explicit decoupled data orchestration approach where the data movement for the three memory structures is operated by independent memory controllers that interface both DRAM and the datapath. In our design, these memory controllers are one of the key sauces to feed the accelerator given the configured dataflow, and still preserve low area and power overheads.

Having one memory controller for each combination of dataflow and memory structure would be very costly in terms of area and power as it would require 30 logic modules to orchestrate the data (*6 dataflows* $\times$ *5 memory controllers*). In our design, we have unified the logic and each controller is able to be configured according to the memory access pattern of each dataflow. This way, as shown in Fig. 4.11, we only need two controllers to orchestrate the data for the memory structure which is kept stationary (i.e., the tile filler STA and the tile reader STA), two memory controllers to orchestrate the memory structure for the streaming matrix (i.e., the tile filler STR and the tile reader STR) and a single controller to orchestrate the memory structure for C (i.e., the tile writer C). Fig. 4.13 shows the code of these unified memory controllers.

Inputs

- p_A: Row Pointer Vector STA	- i_A: Col Pointer Vector STA	- d_A: Data Vector STA
- p_B: Row pointer vector STR	- i_B: Col Pointer Vector STR	- d_B: Data Vector STR
- p_C: Row pointer vector C	- i_C: Col Pointer Vector C	- d_C: Data Vector C

Tile Filler/Reader STA

```
for (int i=0; i<p_A.size; i++) {
    for (int j=p_A[i]; j<p_A[i+1]; j++):
        mem.push() / mem.pop()
```

Tile Filler/Reader STR

```
int high_L1=(gustavsons||inner_product) : p_A.size ? 1
for (int i=0; i<n_high_L1; i++):
    int low_L2=(gustavsons) : p_A[i] ? 0
    int high_L2=(gustavsons) : p_A[i+1] ? p_B.size
    for (int j=low_L2; j<high_L2; j++):
        curr_pointer=(gustavsons) : receive(i_A[j]) ? j
        int low_L3 = p_B[curr_pointer]
        int high_L3 = p_B[curr_pointer+1]
        for (int k=low_L3; k<high_L3; k++):
            mem.fetch[k] / mem.read[k]
```

Tile Writer C

```
int rcv_elems=0
  while (pck=pending_element()):
      value=pck.data()
      row=pck.row()
      col=pck.col()
      if(pck.is_partial_sum()):
         PartialWrite(row, col, value);
      else:
         d_c[rcv_elems]=value
         i_C[rcv_elems]=col
         if(new_row(row)):
            p_C[row]=rcv_elems
```

Figure 4.13: Pseudo-code of the tile filler STA, tile reader STA, tile filler STR, tile reader STR and the tile writer C. We fuse the fillers and readers in the same text box. STA: Stationary, STR: Streaming.

4.4 Experimental Methodology

4.4.1 Simulation Infrastructure

For a detailed evaluation of Flexagon, we have implemented a cycle-level microarchitectural simulator of all on-chip components of our accelerator by leveraging the STONNE framework [82] described in Chapter 2. To faithfully model the whole memory hierarchy including an HBM 2.0 off-chip DRAM, we have connected the simulated accelerator to the Structural Simulation Toolkit [60] which processes the memory requests by means of its built-in memory hierarchy element library. To do this, we have used as a development framework the SST-STONNE version of the STONNE simulator, previously described in Section 2.7.2. Table 4.5 shows the main parameters of the architecture we have configured for the rest of the evaluation. We compare our results against three state-of-the-art accelerators: SIGMA-like as an example of an `Inner Product` accelerator, SpArch-like as an example of an `Outer Product` accelerator and GAMMA-like as an example of a `Gustavson's` accelerator. For the three accelerators, we model the same parameters presented in Table 4.5, and we only change the memory controllers to deliver the data in the proper order according to its dataflow.

To demonstrate the benefits of Flexagon, our evaluation methodology considers the following three different angles:

4.4.2 End-to-End Evaluation

To truly prove the performance benefits of Flexagon, we have carried out end-to-end execution of complete DNN models (see Table 4.2) in our simulated accelerators. These models are present in the MLPerf benchmark suite [106] and we take other models for completeness. Specifically, similar to the benchmarks used in our work described in Chapter 2, we consider Alexnet [67] (A), Squeezenet [56] (SQ), VGG-16 [113] (V), Resnets-50 [49] (R), SSD-Resnets [77] (S-R), SSD-Mobilenets [75] (S-M), DistilBERT [111] (DB) and MobileBERT [116] (MB). To execute every DNN model's layer, we use the PyTorch DNN framework [5] to extract the SpMSpM kernels, prune the weight matrix based on the sparsity levels shown in the table and encode the matrices using both CSR and CSC formats. After this, we feed our accelerator and our baselines to run the kernels on the four simulated architectures.

Parameter	Description
Number of Multipliers	64
Number of Adders	63
Distribution bandwidth	16 elems/cycle
Reduction/Merging bandwidth	16 elems/cycle
Total Word Size (Value+Coordinate)	32 bits
L1 Access Latency	1 cycle
L1 STA FIFO Size	256 bytes
L1 STR cache Size	1MiB
L1 STR Cache Line Size	128
L1 STR Cache Associativity	16
L1 STR Cache Number of Banks	16
DRAM size	16 GiB
DRAM access time	100 ns
DRAM Bandwidth	HBM 2.0

Table 4.5: Configuration parameters of Flexagon.

4.4.3 Layer-wise Evaluation

Since explaining the results requires delving into a finer-grained detail, we have selected 9 representative layers extracted from the execution of the DNN models. Table 4.6 shows these layers together with the SpMSpM dimensions (i.e., M, N and K), the sparsity of the matrices A and B, and the resulting compressed size of the matrices A, B and C expressed in KiB.

4.4.4 RTL Results

We implemented the main building blocks (i.e., the DN, MN, RN and the on-chip memory) of the accelerators considered in this work (shown in Table 4.7) and represent the SIGMA-like, SpArch-like, GAMMA-like and Flexagon accelerators. For an apples-to-apples comparison of overheads, the four architectures use the same tree topology for the DN, the same linear array of multipliers for the MN and vary the RN. For the SIGMA-like architecture, we utilize the FAN network [31] as the RN for flexible-sized reductions. For the SpArch-like and GAMMA-like architectures, we use a merger [97,124] to merge the partial sums produced after the multiplications. Finally, for Flexagon we utilize the unified MRN explained in Section 4.3.

Layer	M, N, K	spA	spB	csA	csB	csC
SQ5	64, 2916, 16	68	11	1.2	162	728
SQ11	128, 729, 32	70	10	4.8	82	364
R4	256, 3136, 64	88	9	7.6	709	3136
R6	64, 2916, 576	89	53	16	3086	728
S-R3	64, 5329, 576	89	46	16	6422	1332
V0	128, 12100, 576	90	61	29	21357	12321
MB215	128, 8, 512	50	0	128	16	4
V7	512, 144, 4608	90	94	921	177	288
A2	384, 121, 1728	70	54	777	373	181

Table 4.6: Representative DNN layers selected for the evaluation. sp{A,B}=sparsity of matrix {A,B} (in %), cs{A,B,C}=compressed size of matrix {A,B,C} (in KiB).

	SIGMA-like	Sparch-like	GAMMA-like	Flexagon
DN	Tree	Tree	Tree	Tree
MN	Linear	Linear	Linear	Linear
RN	FAN	Merger	Merger	MRN

Table 4.7: Main building blocks to model the SIGMA-like, Sparch-like, GAMMA-like and Flexagon accelerators. DN=Distribution Network, RN=Reduction Network and MN= Multiplier Network.

For synthesis, we use MAERI BSV [3] to generate the 64-MS distribution network and the multiplier network. In addition, we have implemented in RTL a 64-wide merger and our MRN. We use Synopsys Design Compiler and Cadence Innovus Implementation System for synthesis and place-and-route, respectively, using TSMC 28nm GP standard LVT library at 800 MHz. To obtain the area and power numbers of the memory structures, we have used CACTI 7.0 [54] for the same technology node and frequency.

4.5 Results

4.5.1 End-to-End Results

Figure 4.14 compares the performance obtained with the three contemporary fixed-dataflow accelerators (SIGMA-like, SpArch-like and GAMMA-like) and with Flexagon when running the 8 DNN models (speed-ups with respecto to the SIGMA-like accelerator).

The first observation is that there is no fixed-dataflow accelerator that can obtain the highest performance for all the 8 DNN models. In particular, for *Alexnet* (`A`), *VGG-16* (`V`), *Resnets-50* (`R`) and *SSD-Resnets* (`S-R`) the SpArch-like accelerator is 5.26× and 1.49× on average faster than the SIGMA-like and GAMMA-like architectures, respectively. Conversely, for *Squeezenet* (`SQ`), *SSD-Mobilenets* (`SM`), *DistilBert* (`DB`) and *MobileBert* (`MB`), the GAMMA-like accelerator obtains the best performance (average improvements of 3.28× and 2.41× against the SIGMA-like and SpArch-like, respectively).

The second and most noteworthy observation is that Flexagon can outperform the other three fixed-dataflow accelerators in all cases, attaining average speed-ups of 4.59× (vs. SIGMA-like), 1.71× (vs. SpArch-like) and 1.35× (vs. GAMMA-like). This is due to the combination of its flexible interconnects, explicitly decoupled memory structures and unified memory controllers that enable using the most efficient dataflow for each layer.

4.5.2 Layer-wise Results

Detailing the reasons behind the benefit observed for some DNN models for a particular dataflow requires a deeper delve into every DNN layer execution. To make the study feasible (we run over a hundred of layers), next, we present a comprehensive study for a selected set of nine representative DNN layers (Table 4.6). These layers are chosen according to the dataflow from which they benefit the most –The first three layers in the table benefit from `Inner Product` (*SQ5*, *SQ11* and *R4*), the second ones from `Outer Product` (*R6*, *S-R3* and *V0*), and the third ones from `Gustavson's` (*MB215*, *V7* and *A2*).

Figure 4.15 shows a performance comparison running these selected layers using our simulated accelerators (again, speed-ups are computed with respect to SIGMA-like). Note the y-axis uses a logarithmic scale. Each bar shows a breakdown depending upon the fraction of execution time spent either on Multiplication phase and on Merging phase. First, note that there is high sensitivity in terms

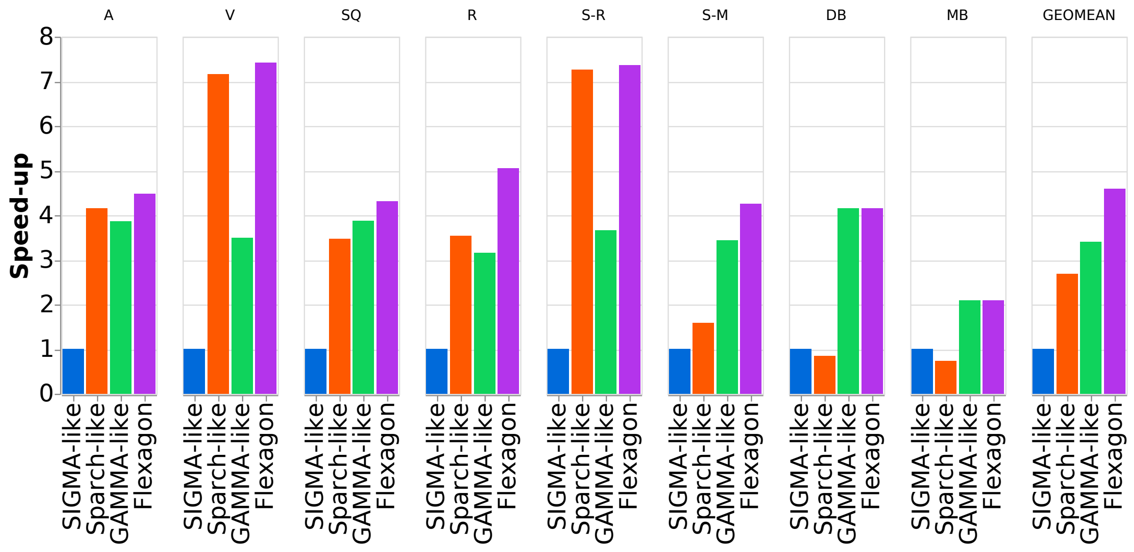


Figure 4.14: Performance comparison between SIGMA-like, SpArch-like, GAMMA-like and Flexagon architectures across the 8 DNN models (speed-up against the SIGMA-like one).

of layers that exist across the different models. As we can see, the results may be divided in three different groups, each containing three layers. The first three layers *SQ5*, *SQ11* and *R4* benefit from the `Inner Product` dataflow. The second group of three layers *R6*, *S-R3* and *V0* benefit from the `Outer Product` dataflow and the last group of three layers composed of *MB215*, *V7* and *A2* benefit from the `Gustavson's` dataflow. As expected, as shown in the figure, in case of the first group of `Inner Product`-friendly layers, the SIGMA-like architecture obtains average speed-ups of 1.53$\times$ and 1.40$\times$ against the SpArch-like and the GAMMA-like architectures, respectively. The next three `Outer Product`-friendly layers (i.e., *R6*, *S-R3* and *V0*), the SpArch-like architecture obtains an average increased performance of 5.07$\times$ and 2.66$\times$ against the SIGMA-like and GAMMA-like architectures. Finally, the last three `Gustavson's`-friendly layers, the best performance is obtained by the GAMMA-like architecture, experimenting 4.37$\times$ and 3.19$\times$ faster executions than the SIGMA-like and the SpArch-like architectures, respectively. More remarkable is that Flexagon beats all of them, always reaching the performance of the best case.

Overall, by properly configuring the control logic of Flexagon according to the most suitable dataflow for each layer, our accelerator is able to attain 2.81$\times$, 1.69$\times$,

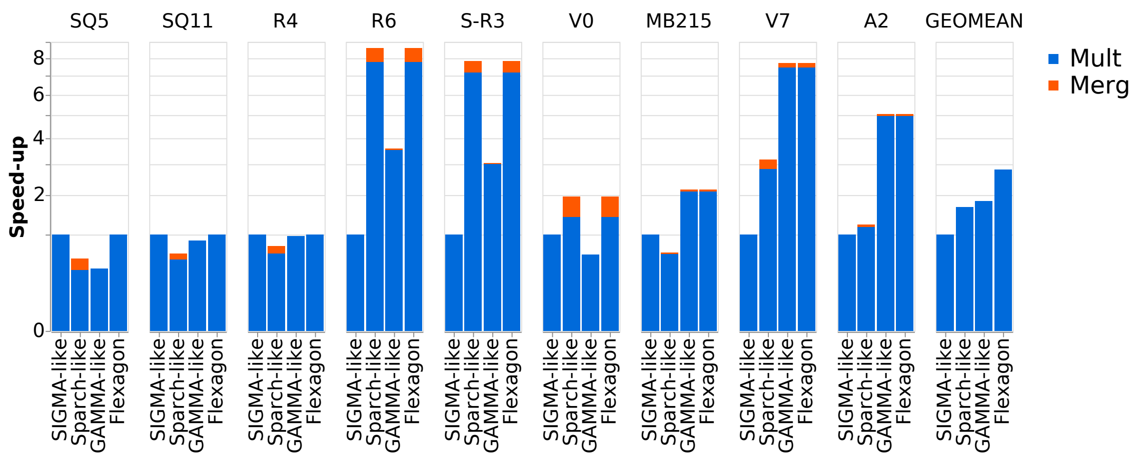


Figure 4.15: Performance comparison between SIGMA-like, SpArch-like, GAMMA-like and Flexagon architectures across our 9 DNN layers (speed-up against the SIGMA-like one).

and 1.55× speed-ups against the SIGMA-like, SpArch-like and GAMMA-like accelerators.

Figures 4.16, 4.17 and 4.18 help us understand these results. Specifically, Figure 4.16 shows the amount of on-chip memory traffic (expressed in MBs) that relays between our on-chip memory hierarchy (i.e., the reads from the STA FIFO and from the STR cache and the reads/writes from/to the PSRAM) and the distribution network after running the SIGMA-like, SpArch-like, GAMMA-like and Flexagon architectures across our nine DNN layers. Figure 4.17 plots the cache miss rate of the STR cache after running the layers, and Figure 4.18 shows the amount of off-chip traffic (expressed in KiBs) that in consequence, flows between this STR cache and the DRAM.

The first observation that we would like to make from Figure 4.16 is the negligible traffic that is fetched from the memory structure for the STA matrix (inappreciable fractions of the bars in blue color). This is basically due to the fact that the stationary data is kept stationary in the multipliers once it is read for the rest of the execution, as it is explained in Section 4.3. For this reason, this memory structure does not have a significant impact on the final performance of the executions regardless of the dataflow that is configured. In contrast, the amount of traffic required to fill the structure for the STR matrix and the PSRAM heavily varies layer by layer and across dataflows (fractions of the bars in orange

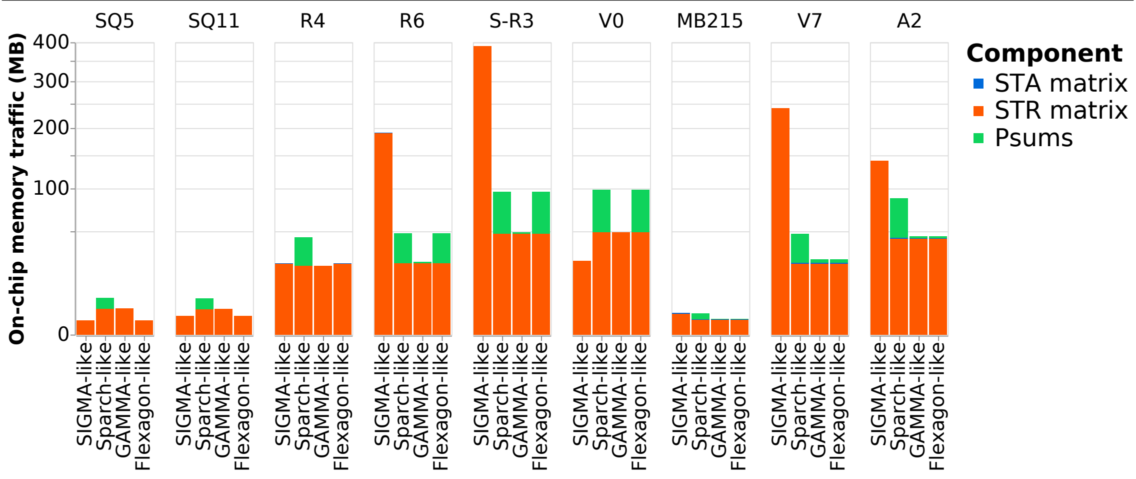


Figure 4.16: Memory traffic (MB) that flows through the on-chip memory hierarchy for SIGMA-like, SpArch-like, GAMMA-like and Flexagon architectures across our 9 DNN layers.

and green colors respectively), hence determining the final performance of the layer execution.

Since the `Inner Product` dataflow does not require to merge the partial sums as they are internally accumulated (observe the number of partial sums sent to the PSRAM for the SIGMA-like architecture is always 0) this dataflow obtains the best performance. An outlier for this behaviour is observed for the *V0* layer. Here, the traffic generated for the STR matrix in the SIGMA-like architecture is lower than the traffic generated in the SpArch-like and GAMMA-like architectures. However, this workload experiences higher runtime. The reason of this is the large size of the matrix B (21.3 MiB) which causes that it has to be reloaded several times, experimenting a L1 miss rate of 3.13% (see Figure 4.17), significantly higher than the L1 miss rates obtained for the SpArch-like and GAMMA-like architectures (i.e., 0.36% and 2.30%) which translates into increased off-chip memory traffic (see Figure 4.18). This higher traffic provokes that the multiplying phase takes longer for the SIGMA-like architecture than for both the multiplying and merging phase for the SpArch-like architecture. When the number of intersections is low, the SIGMA-like architecture experiments higher number of cycles overheads due to this architecture accesses to many more data elements. This is also observed in the six layers that do not benefit from the SIGMA-like architecture (i.e., *R6*, *S-R3*, *V0*, *MB215*, *V7* and *A2*), experiencing on average 5.68$\times$ and 2.27$\times$ higher on-chip traffic than the SpArch-like and GAMMA-like architectures.

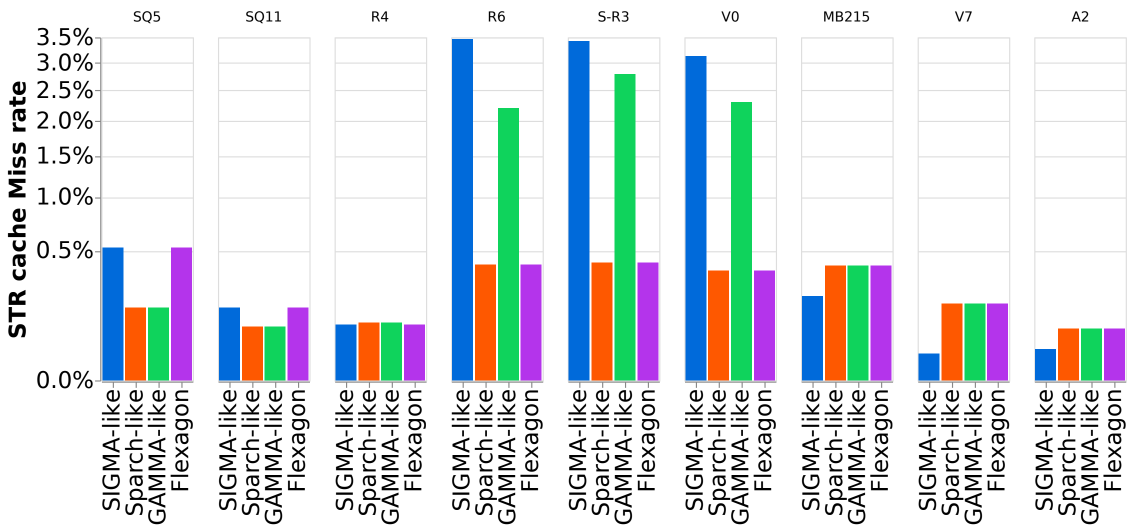


Figure 4.17: STR cache miss rate for the SIGMA-like, SpArch-like, GAMMA-like and Flexagon architectures across 9 DNN layers.

On the other hand, out of these six layers, the main difference of performance that defines them comes from the size of the matrix B. The second group of layers (i.e., *R6*, *S-R3* and *V0*) that benefit from the SpArch-like architecture have a large size of matrix B (see Table 4.6). This implies that the GAMMA-like architecture cannot fit the rows of B entirely in the memory structure for the STR matrix, causing higher L1 miss rates. Observe the average L1 miss rate (see Figure 4.17) experimented in the execution of these three layers is 0.39% for the SpArch-like architecture and 2.43% for the GAMMA-like architecture. This translates into 6.25× more traffic for GAMMA which causes the degradation in performance.

In the last group of layers (i.e., *MB215*, *V7* and *A2*) the size of matrices B are much smaller (up to 373 KB as observed in Table 4.6) and therefore both SpArch-like and GAMMA-like architectures experience the same L1 miss rates and off-chip data traffic. In this scenario, the GAMMA-like architecture is more efficient as it is able to compute the merging phase and the merging phase at the same time –Observe the orange bar for the GAMMA-like cases in the Figure 4.15 is not significant as the merge phase is computed in parallel within the multiplying phase (i.e., blue bar).

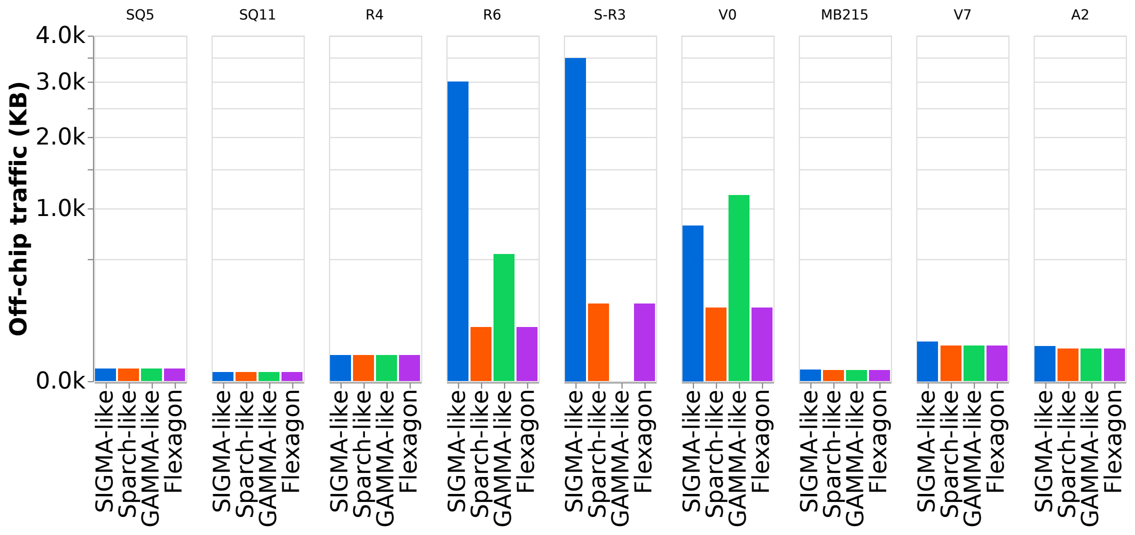


Figure 4.18: Off-chip data traffic for the SIGMA-like, SpArch-like, GAMMA-like and Flexagon architectures across 9 DNN layers.

4.5.3 Mapper Insights

Upon our previous analysis, we expose some insights that would help a mapper for Flexagonto quickly identify whether a layer has to be configured to execute an `Inner Product`, `Outer Product` or `Gustavson's` dataflow. Based on the previous observations, we can see that the key is mostly the matrix B. The layers that benefit from `Inner Product` are the ones that has a matrix B size inferior to the cache size and the sparsity pattern advocate to match the elements of the matrix A. If one these requirements is not accomplished, the layer would be candidate either for the `Outer Product` or the `Gustavson's` dataflow. To select one, a mapper would have to look up the sparsity patterns of A and the size of the matrix B. Both determine how many rows and the size of the rows to fetch during the streaming phase. If there are too many rows or the size of them exceeds the cache size, then using a `Gustavson's` dataflow would increase the L1 miss rate and therefore would cause a degradation of performance. Consequently, in this case, it would be more convenient to use an `Outer Product` dataflow. Otherwise, the preferable dataflow is the `Gustavson's` dataflow.

Obviously, these are just superficial insights that would help a tool to take a decision. However, the study of a mapper able to select the appropriate dataflow given the details of a layer requires more effort and is not the focus of this work.

Component	SIGMA-like	Sparch-like	GAMMA-like	Flexagon
Area Results				
DN (mm^2)	0.04	0.04	0.04	0.04
MN (mm^2)	0.07	0.07	0.07	0.07
RN (mm^2)	0.17	0.07	0.07	0.21
Cache (mm^2)	3.93	3.93	3.93	3.93
PSRAM (mm^2)	-	1.03	0.51	1.03
Total (mm^2)	4.21	5.14	4.62	5.28
Power Results				
DN (mW)	2.18	2.18	2.18	2.18
MN (mW)	3.29	3.29	3.29	3.29
RN (mW)	248	64.48	64.48	312
Cache (mW)	2142	2142	2142	2142
PSRAM (mW)	-	538	269	538
Total (mW)	2396	2750	2481	2998

Table 4.8: Post-layout area and power obtained for SIGMA-like Sparch-like, GAMMA-like and Flexagon accelerators.

4.5.4 RTL Results

Table 4.8 shows a breakdown of the total amount of area (mm^2) and power (mW) obtained for the 64-MS SIGMA-like, SpArch-like, GAMMA-like and Flexagon accelerators. For each case, we show the results for the main architectural components: Distribution Network (DN), Multiplier Network (MN), Reduction/Merger Network (RN), the cache structure for the streaming matrix (Cache) and the PSRAM .

In terms of area, we observe that Flexagon introduces an overhead of 25%, 3% and 14% with respect to the area of the SIGMA-like, SpArch-like and GAMMA-like accelerators, respectively. As we can see, the area of the four accelerators is mostly dominated by the memory structures. Specifically, we observe that the cache for the streaming matrix represents a 93%, 76%, 85% and 74% of the total amount of area for the SIGMA-like, SpArch-like, GAMMA-like and Flexagon architectures, respectively. Besides, the area of the PSRAM represents a 20%, 11% and 19% with respect to the SpArch-like, GAMMA-like and Flexagon accelerators, respectively. Since the SIGMA-like architecture employs an `Inner Product` dataflow, this accelerator does not need this structure, which explains the reason of having the lowest area. Also, the area of the PSRAM is the GAMMA-like accelerator is half the area in the SpArch-like and Flexagon accelerators as it requires to store less partial sums, which explains the area reduction. Obviously, Flexagon needs support for the worst-case `Outer Product` dataflow and needs the highest PSRAM overhead. Finally, note that our MRN is 28% and 128% larger than the area of the FAN and the merger, but this does not translates into high overall overhead as the MRN takes only a 4% out of the total area for Flexagon.

In terms of power, we observe the same trends. We find that the Flexagon accelerator consumes 25%, 9% and 21% more power than the SIGMA-like, SpArch-like and GAMMA-like accelerators. The slightly higher overhead of Flexagon against the aforementioned area results comes mostly from the Merger/RN as this module represents a larger fraction of total consumption (10%, 2.34%, 2.60% and 10.41% out of the SIGMA-like, SpArch-like, GAMMA-like and Flexagon accelerators are observed, respectively). This, together with the fact that the MRN consumes 25% and 284% more than the FAN RN and the merger, explains the results.

In spite of the overhead introduced, in Figure 4.19 we illustrate that Flexagon is still more performance/area efficient. Specifically, we consider both achieved speed-ups and area requirements of each design. The area requirements are normalized with respect to the SIGMA-like case, which is also the reference for the calculation of the speed-ups. Note that the NLP models like *MobileBert* (*MB*) and *DistilBert* (*DB*) achieves a better efficiency with the GAMMA-like accelerator. Nevertheless, this is due to as explained before, most of the layers (84% in DistilBert (*DB*) and 100% in MobileBert (*MB*)) for these models work better with the `Gustavson` dataflow, making the area overhead introduced by the Flexagon accelerator unnecessary.

Consequently, we can clearly see that, overall, Flexagon reaches the best compromise between performance and area consumption (the higher Speed-up/Area values). In comparison, we find that, on average, our accelerator obtains 18%, 67% and 265% better performance/area efficiency across the execution of our 8 DNN models with respect to the GAMMA-like, SpArch-like and SIGMA-like accelerators. This makes Flexagon the best candidate for running heterogeneous sparse DNN workloads.

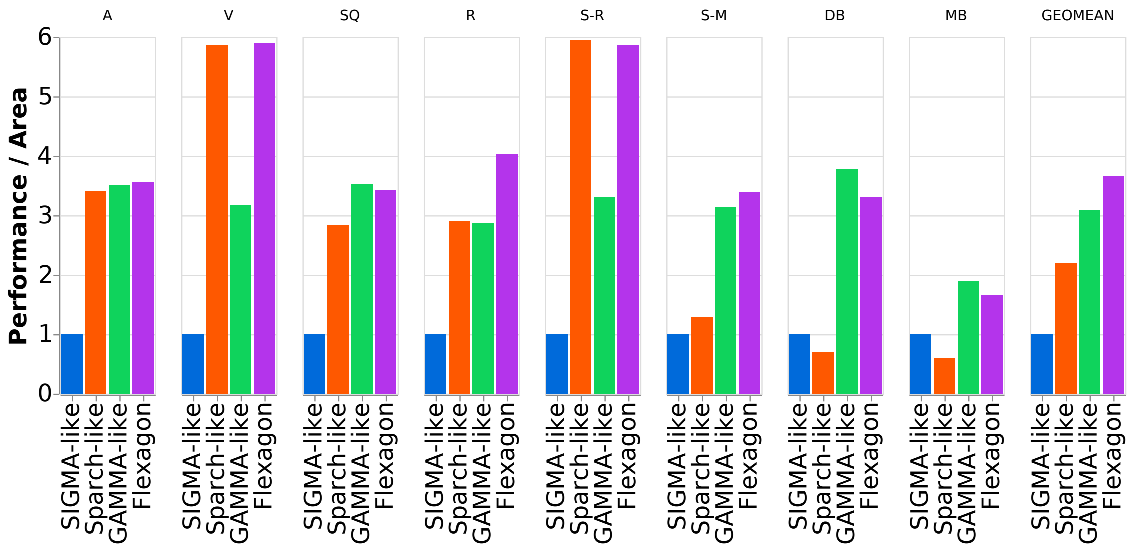


Figure 4.19: Performance/Area obtained after running the SIGMA-like, SpArch-like, GAMMA-like and Flexagon architectures across our 8 DNN models.

4.6 Related Work

Sparse DNN Accelerators: Sparse matrix multiplications (SpMM, SpMSpM and SpGEMM) have been prime targets of acceleration for AI and HPC workloads. Several sparse DNN accelerators have been proposed for SpMM, SpGEMM and Sparse convolution [24,31,42,46,50,63,72,97,98,114]. These accelerators have support for sparse execution via compression of one or both operands into formats like CSR, CSC, bitmap, CSF etc. This reduces the memory footprint and the number of multiplications. Some prior works like Eyeriss [22] focus on sparsity via zero-gating of the multiplier where multiplication by zero is skipped, this saves the number of multiplications but does not reduce the on-chip memory footprint. Eyeriss compresses data between the Global buffer and the DRAM via run-length coding. As Table 4.1 shows, prior sparse accelerators have picked either one of `Inner Product`, `Outer Product` and `Gustavson's`(row-wise product) dataflows. We show that flexibility to support multiple dataflows is beneficial for performance and performance per area over state-of-the art accelerators.

Frameworks for flexible accelerators: In order to adapt the dataflow to the workload, prior flexible accelerators have been proposed for Dense DNNs. Some of the state-of-the-art flexible DNN accelerators include MAERI [70] and Flexflow [76]. Other accelerators include Reconfigurable Dataflow Accelerators

(RDAs) [123] aka CGRAs, for example Plasticine [101]. However, flexible sparse accelerators need to consider the ability to support multiple formats and the ability to execute on all compressed inputs, weights or outputs depending on the dataflow. Even though, SIGMA [31] provides flexibility in choosing arbitrary number of reduction tile sizes, it still supports only `Inner Product` and we demonstrate in this work that `Outer Product` and `Gustavson's` can be beneficial over `Inner Product` for many the workloads. Prior works in the direction of flexibility include hardware widgets and design-space exploration tools for CGRAs. MINT [103] is an efficient format converter widget that supports multiple sparse formats both for storage and for compute. Prior works Garg et al. [41], coSPARSE [35] and SparseAdapt [96] propose frameworks for efficient sparse execution on CGRAs. However, to the best of our knowledge, this is the first work that proposes a flexible accelerator for Sparse DNNs which has the ability to exploit all the three dataflows.

4.7 Conclusion

This chapter proposes Flexagon, the first SpMSpM accelerator design that offers `Inner Product`, `Outer Product` and `Gustavson's` dataflows on a homogeneous hardware substrate. Our proposal revolves around a novel tree-based network (MRN) that supports both reduction of dot products and merging of partial sums which are the required operations for the three dataflows. To do so, each node of the tree is composed of an adder and a comparator. For `Inner Product` dataflow the tree is configured to act as a dot product engine. In contrast, for `Outer Product` and `Gustavson's` , the tree is configured as a merger.

On the other hand, in order to capture all dataflows, Flexagon implements a customized L1 memory level specifically tailored for the common and different patterns among the three dataflows. In particular, the memory is organized as three main SRAM blocks: A memory structure for the stationary matrix (FIFO) that stores the elements of the stationary matrix in the three dataflows, a memory structure for the streaming matrix which is organized as a traditional cache and is used to capture a more irregular memory access pattern generated mostly by `Gustavson's` dataflow, and a memory structure for matrix C which is called PSRAM and is specifically designed to efficiently write and read temporal psums that are generated during the `Outer Product` and `Gustavson's` dataflows.

By using the dataflow that best matches the characteristics of each DNN layer, we have shown that Flexagon brings significant improvements in performance/area efficiency over state-of-the-art fixed-dataflow sparse SIGMA-like, SpArch-like and GAMMA-like accelerators. In particular, in our performance/area comparison we have found that, on average, our accelerator obtains 18%, 67% and 265% better performance/area efficiency across the execution of our 8 DNN models with respect to the GAMMA-like, SpArch-like and SIGMA-like accelerators. This makes Flexagon accelerator the best candidate for running heterogeneous sparse DNN workloads.

CHAPTER 5

Conclusions and Future Ways

5.1 Conclusions

The last four decades have been known as the golden years of computer architecture. In the last decade and a half, the trend towards designing very aggressive and complex superscalars processors was replaced by the trend of adding more and more simpler cores in the same chip. Regardless of the strategy followed to leverage the transistors, architects ensured that the effect of Moore's Law was always accomplished, doubling the performance of processors every 2 years, driven by the advances in technology and node shrinking.

Unfortunately, this golden age is coming to an end and the strategy of taking advantage of putting more and more transistors in the same chip is no longer feasible (at least with current technology). The transistor size is reaching its physical limit and the only one path left to continue improving computer performance is specialization. Architects need to find common algorithms or applications and design specific hardware to accelerate them. This is the beginning of a new golden age for computer architecture.

Artificial intelligence and specifically DNNs are the greatest target for specialization nowadays, as these algorithms constitute a promising breakthrough for a large number of artificial intelligence applications. A common compute-intensive algorithm that can be leveraged across multiple artificial intelligence domains. *Could we imagine a better scenario?*

As expected, this has recently fueled interest in the development of specific accelerator architectures for DNNs capable of meeting their stringent performance and energy consumption requirements.

The deployment of a DNN model comprises two phases called *training* and *inference*. During the training phase, the DNN model adjusts the values of its set of weights. Subsequently, during inference, the trained DNN model is used to solve the problem it was designed for (e.g., image classification). Currently, training is mostly carried out using clusters of several GPUs, although some proposals for customized training platforms have also been developed by both industry (e.g. Google's Cloud TPU and Microsoft's Project Brainwave) and academia. In contrast, the fact that their inference phase must be primarily done *in-situ* has paved the way for the development of a plethora of accelerator architectures. The key behind all of these recent architectures has been the capture of the different patterns of data reuse in what is known as a dataflow.

First-generation DNN inference accelerators focused their designs on fixed-size clusters of multipliers-and-accumulate units interconnected by means of a fixed tightly-integrated on-chip network fabric specifically tailored to efficiently support a particular dataflow. For example, the Google TPU is built by interconnecting 256×256 Multiply-Accumulate (MAC) units to a tightly-coupled 2D grid and supports a weight-stationary dataflow, while ShiDianNao groups 8×8 MAC units supporting an output-stationary dataflow. Unfortunately, as DNN models evolve at a rapid pace, these fixed designs fail to adapt well to the particular characteristics of contemporary DNN models. The emerging of wider and deeper models with highly heterogeneous layers in terms of sizes and types, as well as the increased number of zeros in the DNN models (i.e., sparse models) forced to the introduction of the next generation of architectures. Some of these latest designs are known as flexible architectures, like MAERI or SIGMA, which allows to the creation of multiple-number and multiple-size groups of cluster of processing elements (i.e., multipliers). Others like ExTensor, SpArch or GAMMA focus on the sparsity support and are capable to run the sparse kernels in the DNNs (i.e., SpMSpM operation) in a very efficient manner.

Due to the explosion in terms of the number and type of new specific architectures, we required a way to compare all of them.

Traditionally, architectural simulators have become an integral part of computer architecture research and design process, since they permit fast and accurate design-space exploration and rapid quantification of the efficacy of architectural enhancements in the early stages of a design. Therefore, architectural simulators have been extensively used during the design process of CPU and GPU architectures. However, and quite surprisingly, the same had not taken place until now for DNN accelerator architectures. To the best of our knowledge, there was no detailed, cycle-accurate, open-source microarchitectural simulator for extensive and accurate design-space exploration of DNN inference accelerators.

To bridge this gap, the first proposal of this thesis has been the development and evaluation of STONNE (which stands for Simulation TOol of Neural Network Engines), the first attempt to derive a cycle-accurate, highly-modular and highly-extensible simulator for next-generation flexible DNN inference accelerator microarchitectural exploration.

STONNE framework is composed of three major modules involved in the end-to-end simulation flow: First, the Input Module which is used to define the DNN to be run and to load the parameters of the layer and the initial inputs and weights onto the simulated accelerator. Once the accelerator has been configured, the Simulation engine module carries out a cycle-by-cycle microarchitectural simulation of the accelerator during the execution of the feed-forward computation of the layer (i.e., the inference procedure), collecting statistics during the process. After this, the results collected during the execution of the layer are sent back to the CPU, and finally, once the simulation of each simulated layer is completed, the Output Module takes in the values of the counters collected by the simulated architecture and produces several useful statistics of the execution, such as performance and energy consumption. Area numbers can also be reported by STONNE by using a table-based scheme like the one presented in Accelergy framework. The simulation engine is equipped with all the major components required to construct first-generation and flexible DNN accelerators. All these on-chip components are interconnected by using a three-tier network fabric composed of a Distribution Network (DN), a Multiplier Network (MN), and a Reduce Network (RN). These networks can be configured to support any topology (and therefore any accelerator) and any configuration parameter. We have validated STONNE against actual RTL hardware designs obtaining that the simulator achieves an error accuracy between 0.14% to 3.10% (1.53% on average), demonstrating that STONNE closely mimics the characteristics of the hardware versions.

Through three use cases, we demonstrated in this thesis how STONNE can be used to conduct comprehensive evaluations of several DNN accelerator architectures running complete DNN models.

The aim of the first use case was to directly compare three different accelerator architectures (namely, TPU, MAERI and SIGMA) considering their achievable performance, energy consumption and required area. All the simulations were performed considering the complete inference processing of 7 state-of-the-art

DNN models. Overall, the results of these use case showed that a MAERI-like architecture reaches average performance improvement of 20% over the TPU-like architecture for the execution of the seven DNN models, with a maximum of 231% for Mobilenets and a minimum of 9% for Resnets-50. Besides, we found that a SIGMA-like architecture is 91% faster on average than a MAERI-like one thanks to the sparsity support. In the second use case, we have used STONNE to model the data-dependent accelerator SnaPEA. This architecture that aims to optimized CNN processing, exploits a property in which all the activation values in the convolution operations are either zero or positive. This use case proved how the back-end of STONNE can be easily extended to model other accelerators. The results obtained with STONNE showed that SnaPEA can bring average speedups of 35%, closely approaching the 30% originally reported in the original work, demonstrating the usability of the STONNE simulator. In the third use case we proposed a new scheduling technique that help exploiting the filter sparsity in a more efficient way. In particular, we observed that scheduling the filters according to a certain heuristic such as largest filter first may improve the performance of processing some DNN layers up to 13% (11% on average). This case of study reveals the necessity of performing end-to-end and cycle-accurate simulations to accurately evaluate DNN executions on sparse accelerators.

Once the STONNE simulator was completely validated, we were able to use this tool to improve some accelerator architectures from the state-of-the-art. With this aim, the second proposal of this thesis has been the design of a new RN for flexible accelerators called STIFT (which stands for Spatio-Temporal Integrated Folding Tree). Recent *flexible* accelerators advocated having physically separated and reconfigurable DN, MN and RN fabrics. The RN is built from more configurable AUs called Adder Switches (ASs). Each AS is an AU augmented with a tiny switch to enable arbitrary cluster reductions of variable size over the same physical RN. The type of RN that materializes this flexibility is a Spatial Reduction Tree (S-Tree) used in both the ART and the FAN RNs proposed for MAERI and SIGMA accelerators, respectively. This type of RN enables efficient reduction by employing a binary tree-based accumulation so that an ideal whole reduction operation should take $O(\log_2 n)$ to be completed. Besides, to enable the parallel execution of any number of arbitrary-size clusters, they utilize augmented links that allows for data traversal between nodes that do not share the same parent.

The challenge we have addressed in our second proposal, which had not been analyzed with detail in any of the previous (i.e., MAERI and SIGMA) works, is how to manage the common situation of folding, in which the number of multiplications in a dot product is larger than the number of multiplier units implemented in hardware. In this specific purely spatial approach, which is the one implemented by them, the psum obtained in each cluster iteration is spatially sent to the Global Buffer, and subsequently redistributed from it to a dedicated Multiplication Switch, responsible only for forwarding it back to the RN. This way, the entire accumulation process is performed spatially. As we have observed in our evaluations, this design results into low effective utilization of the mapped MSs as this implementation impedes to iterate over the same dot product in a pipelined manner.

In order to overlap multiplications and sums of consecutive iterations of the same dot product, and thus, be able to attain a seamless pipelined execution for folding, it is necessary to break the dependency between two consecutive iterations by composing a Spatio-Temporal Tree (ST-Tree). To this end, one particular extension over the previous S-Tree, leveraging the design discussed for some ST-based rigid accelerators, was to add a set of accumulators, connected with the ASs, in charge of temporarily accumulating the different psums being calculated for each cluster. This approach was called ST-Tree+Accumulators (ST-Tree$_{ac}$) RN. As discussed during this thesis, this approach has the major inconvenient of entailing significant area and power overhead as it requires to significantly increase the number of adders in the RN (i.e., the extra accumulators).

To ensure efficient folding support in flexible accelerator architectures and to avoid the addition of such extra accumulators, we have proposed in this thesis a novel Reduction Network fabric, specifically suited for flexible accelerator architectures. This was called, STIFT. Similarly to ST-Tree$_{ac}$ RN using accumulators, STIFT is capable of running any number of dynamic-size clusters in a non-blocking manner, but unlike this one, it enables efficient and flexible support to ensure full non-blocking processing of folding. The observation behind STIFT is that for the S-Tree RN employed in recent accelerator proposals, there are free ASs when two or more clusters are configured. Therefore, we can use those (as long as we add the proper links) free ASs to perform the accumulations, without the need of duplicating the hardware. In short, STIFT network is a binary-tree topology with horizontal links between the nodes belonging to the same level that do not share the same parent as well as some other additional links that ensure that every cluster can use an AS to accumulate its psums and produce its final output value.

To prove the benefits of STIFT as the RN of a flexible accelerator architecture, we have considered different angles for our evaluation. First, we have analyzed the power and on-chip area overheads that the different RN solutions entail. In particular, we have implemented both STIFT and ST-Tree$_{ac}$ RNs in BSV (Bluespec System Verilog), and use Synopsys Design Compiler and Cadence Innovus Implementation System for synthesis and place-and-route, respectively, using TSMC 28nm GP standard LVT library at 800 MHz. For comparison, we have also also considered MAERI's ART design as an example of S-Tree RN. We have studied how the different RN fabrics scale by exploring different RN widths (number of multiplier units) and data formats (i.e., INT16, FP16 and FP32). Second, we have conducted a comparison of the performance (runtime) that the three flavours of RNs (S-Tree, ST-Tree$_{ac}$ and STIFT) achieve for reduction operations. To do so, we have implemented the three RNs in STONNE simulator and have executed the inference procedure of seven full DNN models on the three resulting accelerator configurations. The RTL results reveals that STIFT reduces area and energy demands of ST-Tree$_{ac}$ by 32% and 31%, respectively, on average. Obviously, when compared to S-Tree, STIFT introduces area and power overheads, as it adds the accumulation logic needed to perform temporal reductions. On average across all the design points, the extra power and area overheads are 17% in both cases, much lower than the 39% and 40%, respectively, added by the accumulation buffer in ST-Tree$_{ac}$. Nevertheless, as we showed, unlike S-Tree and similar to ST-Tree$_{ac}$, STIFT enables pipeline execution of consecutive folding iterations, resulting in much better performance results than S-Tree (and, also, less amount of total energy consumed). Finally, we have seen that when we consider both achieved speed-ups and area requirements of each design STIFT reaches the best compromise between performance and area consumption (the higher Speed-up/Area values). This is due to, compared with ST-Tree$_{ac}$, STIFT achieves virtually the same performance but it requires significantly less area. Overall, we showed that on average across the seven DNN models, STIFT reaches a speed-up/area ratio of 5.13 while this value is reduced to 3.67 in the case of ST-Tree$_{ac}$.

In the third and last proposal of this thesis, we make a step forward and we have designed an entire specific accelerator for sparse DNNs called Flexagon. Existing Sparse-Sparse Matrix Multiplication (SpMSpM) accelerators are tailored to a particular SpMSpM dataflow (i.e., Inner Product, Outer Product or Gustavson's), that determines their overall efficiency. We have demonstrated that this static decision inherently results in a suboptimal dynamic solution because different SpMSpM kernels show varying features (i.e., dimensions, sparsity pattern, sparsity degree), which makes each dataflow better suited to different data sets.

Flexagon is the first SpMSpM reconfigurable accelerator that is capable of performing SpMSpM computation by using the particular dataflow (i.e. Inner Product, Outer Product or Gustavson's) that best matches each case.

Flexagon consists of a set of multipliers, adders and comparators, as well as three on-chip SRAM modules specifically tailored to the storage needs of matrices `A`, `B` and `C` for the three SpMSpM dataflows. In addition, and similar to the previous approaches, in order to allow for the highest flexibility, all the on-chip components are interconnected by using a general three-tier reconfigurable network-on-chip (NoC) composed of a Distribution Network (DN), a Multiplier Network (MN), and a Merger-Reduction Network (MRN), inspired by the taxonomy of on-chip communication flows within AI accelerators.

They two key sauces behind the Flexagon's design are: 1) A novel tree-based network (MRN) that supports both reduction of dot products and merging of partial sums in the same hardware substrate. To do so, each node of the tree is augmented with an adder and a comparator. In dataflows like Inner Products, the hardware is configured to act a ST-Tree, as mentioned previously. In the cases of Outer Product and Gustavson's dataflows the tree is reconfigured to perform the merging operation in a very efficient manner. 2) A special L1 on-chip memory organization, specifically tailored to the different access characteristics of the input and output compressed matrices. In particular, the memory structure is divided in three main SRAM blocks: A Memory structure for the stationary matrix (FIFO) that keeps the elements of the stationary matrix in the three dataflows, a memory structure for the streaming matrix which is a traditional cache used to capture a more heterogeneous memory access pattern generated mostly by Gustavson's dataflow, and a memory structure for matrix C which is called PSRAM and is specifically designed to efficiently write and read temporal psums that are generated during the Outer Product and Gustavson's dataflows.

For a detailed evaluation of Flexagon, we have implemented a cycle-level microarchitectural simulator of all these on-chip components of our accelerator by leveraging the SST-STONNE framework. We have compared our results against three state-of-the-art accelerators: SIGMA-like as an example of a Inner Product accelerator, SpArch-like as an example of an Outer-Product accelerator and GAMMA-like as an example of a Gustavson's accelerator. Using our simulator, we have executed 8 DNN models extracted from MLPerf. In addition, we have implemented the main building blocks (i.e., the DN, MN, RN and the on-chip memory) in RTL of the considered accelerators.

Our results have shown that there is no fixed-dataflow accelerator that can obtain the highest performance for all the 8 DNN models. Furthermore, we have seen that Flexagon can outperform the other three fixed-dataflow accelerators in all cases, attaining average speed-ups of 4.59× (vs. SIGMA-like), 1.71× (vs. SpArch-like) and 1.35× (vs. GAMMA-like). This is due to the combination of its flexible interconnects, explicitly decoupled memory structures and unified memory controllers that enable using the most efficient dataflow for each layer. In terms of area, we observe that Flexagon introduces an overhead of 25%, 3% and 14% with respect to the area of the SIGMA-like, SpArch-like and GAMMA-like accelerators, respectively. In terms of power, we have observed the same trends, finding that the Flexagon consumes 25%, 9% and 21% more power than the SIGMA-like, SpArch-like and GAMMA-like accelerators.

In spite of this, Flexagon still reaches the best compromise between performance and area consumption. In our performance/area comparison we have found that, on average, our accelerator obtains 18%, 67% and 265% better performance/area efficiency across the execution of our 8 DNN models with respect to the GAMMA-like, SpArch-like and SIGMA-like accelerators. This makes Flexagon accelerator the best candidate for running heterogeneous sparse DNN workloads.

As a final conclusion from all above, we can affirm that our three proposals represent a step forward towards the resolution of the challenges that specific architectures for DNN accelerators will pose to computer architects.

5.2 Future Ways

The three proposals presented in this thesis open a large number of new research paths to explore.

The development of the STONNE framework enables for the first first time cycle-level modeling of specific accelerators for DNNs. However, we envision that STONNE can be improved to explore much more:

- The current version of STONNE requires to manually configure the mapping of each layer to be executed. This is slow and tedious for the user as it requires to run previously a mapper tool, such as mRNA, and then, manually configure the simulation. One major contributions that we could do to improve the usability of the tool is to integrate the mRNA tool within the input module of STONNE to feed automatically the mappings.

- The current version of STONNE only supports principal kernels, such as convolution layers, fully-connected layers and matrix multiplications both dense and sparse. Although other types of layers are less significant in terms of computation, adding to STONNE the support for layers like pooling or normalization layers may result interesting.

- STONNE could be used to explore not only DNNs, but also other specific domains. One clear example is the OMEGA framework which is explained in this thesis and is built on top of STONNE to explore specific architectures for GNNs. However, *why do we limit STONNE only to DNNs and GNNs?* We believe that STONNE can be extended and utilized as a reference tool to explore new avenues such as recommendation systems, HPC applications, attention layers, and many more domains that are emerging nowadays.

- STONNE may also enable the study of the interaction of the accelerator-CPU interface. To do so, the STONNE framework could be connected and integrated with some reference simulation tool for CPUs like Gem5, and the simulated CPUs could off-load the DNN layers by using the STONNE API already developed.

- The study of specific accelerators for DNNs within a more heterogeneous system could also be a interesting path to follow. We envision the development of a new programming interface that allows to divide the kernel into smaller pieces and launch them into several simulated devices. With this aim, the simulator SST-STONNE mentioned in this thesis could be the best candidate tool.

The design of STIFT also opens a large number of research paths which are described as followed:

- In this thesis, we have evaluated STIFT for designs similar to MAERI and SIGMA. However, *how could this design be integrated in other accelerators such as the Google's TPU, the Eyeriss design or even in sparse accelerators such as Flexagon?*

- We believe that STIFT could be used in large-scale datacenters. A comprehensive study in terms of scalability could be useful to understand the behaviour of the design at this scale.

Finally, our Flexagon accelerator is an actual step forward towards the design of efficient accelerators for sparse DNNs. In this way, we believe there are still many aspects that may be explored:

- The clearest path is the study of a mapper able to select the appropriate dataflow given the details of a layer. In this thesis, we have exposed some insights that would help a mapper for Flexagon to quickly identify whether a layer has to be configured to execute an Inner Product, Outer Product or Gustavson's dataflow. However, we are still far from understanding the actual facts that could lead to an optimal decision. This study is probably the next research path we will take.

- A more comprehensive study of a memory hierarchy with support for the three dataflows Inner Product, Outer Product and Gustavson's. In this thesis, we have briefly described three SRAM structures that enables the support for the three dataflows. However, *is this decision the optimal one? What is the best configuration (i.e., cache size, line size, etc) that could be utilized?*. Even, *Can the use of these structures be more optimized so that some dataflows can take advantage of the otherwise non-utilized space in some of them?*

- The benchmarks used to evaluate Flexagon are diverse but they are all within the DNN domain. We envision that Flexagon can become the reference sparse accelerator not only for DNNs, but also for other types of applications such as HPC, GNNs or recommendation systems. Therefore, an interesting path to take is to expand the set of benchmarks to these domains and perform a comprehensive study to discover what are the best configuration parameters (i.e., memory sizes and number of processing elements) that bring to the light these domains in Flexagon.

Bibliography

[1] Bluespec System Verilog (BSV). http://wiki.bluespec.com/. (document), 2.2.2, 3.4.1

[2] Caffe website. http://caffe.berkeleyvision.org/. (document), 2.3

[3] MAERI code v1. https://github.com/hyoukjun/MAERI. 2.2.2, 3.2.2, 3.4.1, 4.4.4

[4] OMEGA: Observing Mapping Efficiency Over GNN Accelerators . https://github.com/stonne-simulator/omega. 2.7.1

[5] PyTorch. https://pytorch.org/. (document), 2.1, 2.3, 3.4.3, 4.4.2

[6] SIGMA code v1. https://github.com/georgia-tech-synergy-lab/SIGMA. 2.2.2, 3.2.2

[7] SST-STONNE framework. http://sst-simulator.org/SSTPages/SSTBuildAndInstall_12dot0dot1_SeriesAdditionalExternalComponents/#sst-stonne. 2.7.2

[8] STONNE: A Simulation Tool for Neural Networks Engines . https://stonne-simulator.github.io/. 2.8

[9] The Structural Simulation Toolkit. http://sst-simulator.org//. 2.7.2

[10] Cristina S. Anderson, Jingwei Zhang, and Marius Cornea. Enhanced vector math support on the intel®avx-512 architecture. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, pages 120–124, 2018. 1.2

[11] S. Arora, T. Leighton, and B. Maggs. On-line algorithms for path selection in a nonblocking network. *In Proc. of the 22nd annual ACM symposium on Theory of Computing*, pages 149–158, April 1990. 3.1

[12] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002. 1, 2.1

[13] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, 2009. (document), 1, 2.1

[14] Nathan Binkert and Beckmann et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, 2011. (document), 1, 2.1

[15] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011. 1, 2.1

[16] Lukas Cavigelli and Luca Benini. Origami: A 803-gop/s/w convolutional network accelerator. *IEEE Transactions on Circuits and Systems for Video Technology*, pages 2461–2475, July 2016. 3.2.1

[17] A. Chakrabarty, M. Collier, and S. Mukhopadhyay. Matrix-based nonblocking routing algorithm for beneˇs networks. *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, April 2009. 3.1

[18] Kun-Chih (Jimmy) Chen, Masoumeh Ebrahimi, Ting-Yi Wang, and Yuch-Chi Yang. Noc-based dnn accelerator: A future design paradigm. In *Proceedings of the 13th IEEE/ACM International Symposium on Networks-on-Chip*, pages 1–8, october 2019. 3.1

[19] Kun-Chih (Jimmy) Chen, Masoumeh Ebrahimi, Ting-Yi Wang, and Yuch-Chi Yang. Noc-based dnn accelerator: a future design paradigm. In *2019 IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8, 2019. 3.1

[20] Xiaoming Chen, Danny Z. Chen, and Xiaobo Sharon Hu. moDNN: Memory optimal dnn training on gpus. *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018. (document), 1.4

[21] Yiran Chen, Yuan Xie, Linghao Song, Fan Chen, and Tianqi Tang. A survey of accelerator architectures for deep neural networks. *Engineering*, 2020. (document), 1.5

[22] Yu-Hsin Chen, Joel S. Emer, Tushar Krishna, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, January 2017. 1.4, 1.4.3, 2.1, 2.4.1.3, 3.1, 3.1, 3, 3.5.1, 4.6

[23] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. Using dataflow to optimize energy efficiency of deep neural network accelerators. *IEEE Micro*, 37(3):12–21, June 2017. (document), 1.4, 2.1, 3.1, 3.2.1

[24] 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*, 9(2):292 – 308, June 2019. (document), 2.4.1, 2.4.1.3, 3.1, 3.1, 4.1, 4.6

[25] Intel Corporation. Intel math kernel library, month = march, year = 2007,. 1.3.2.4

[26] H.G. Cragon and W.J. Watson. The ti advanced scientific computer. *Computer*, 22(1):55–64, 1989. 1.2

[27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv: 1810.04805v2 (2019)*, May 2019. 2.1, 3.4.3, 4.1

[28] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. ShiDianNao: Shifting vision processing closer to the sensor. *2015 International Symposium on Computer Architecture (ISCA)*, pages 92–104, June 2015. (document), 1.4.1, 2.1, 2.4.1.3, 3.1, 3.1, 3.2.1

[29] A. Parashar et al. Timeloop: A systematic approach to dnn accelerator evaluation. *Int'l Symp. on Performance Analysis of Systems and Software*, March 2019. (document), 2.1, 2.2.1, 4.1

[30] Ananda Samajdar et al. SCALE-Sim: Systolic cnn accelerator simulator. *arXiv preprint arXiv: 1811.02883v1 (2019)*, February 2019. (document), 2.2.1, 2.5.1

[31] 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*, March 2020. (document), 1.4, 1.4.2, 1.4.3, 2.1, 1, 2.1,

2.1, 2.2.1, 2.4.1, 2.4.1.1, 2.4.1.2, 2.4.1.3, 2.6.1.2, 2.6.3.1, 3.1, 3.1, 3.2.1, 3.2.2, 3.3.1, 4.1, 4.1, 4.1, 4.3.1.1, 4.3.1.2, 4.4.4, 4.6

[32] Shang Li et al. DRAMsim3: A cycle-accurate, thermal-capable dram simulator. *IEEE Computer Architecture Letters*, 19(2), 2020. 2.4.2

[33] Vahideh Akhlaghi et al. SnaPEA: Predictive early activation for reducing computation in deep convolutional neural networks. *int'l Symp. on Computer Architecture*, pages 662–673, June 2018. (document), 1.4, 1.4.3, 2.1, 2.1, 2.6.2, 1, 2.6.2.2, 2.6.2.3, 2.6.2.3

[34] Yakun Sophia Shao et.al. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, October 2019. 1.4

[35] Siying Feng, Jiawen Sun, Subhankar Pal, Xin He, Kuba Kaszyk, Dong-hyeon Park, Magnus Morton, Trevor Mudge, Murray Cole, Michael O'Boyle, Chaitali Chakrabarti, and Ronald Dreslinski. Cosparse: A software and hardware reconfigurable spmv framework for graph analytics. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 949–954, 2021. 4.6

[36] Message P Forum. MPI: A message-passing interface standard, April 1998. 1.2

[37] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2018. 1.4

[38] G. E. Moore. Cramming more Components onto Integrated Circuits. *Electronics*, 38(8):114–117, 1965. (document), 1.1

[39] G. Hinton, D. Sager, M. Upton, D. Boggs, Desktop Platforms Group and Intel Corp. The Microarchitecture of the Pentium-4 Processor. *Intel Technology Journal*, 1:1–13, 2001. 1.1

[40] R. Garg et al. Understanding the design-space of sparse/dense multiphase gnn dataflows on spatial accelerators. In *IPDPS*, 2022. 1.5.1, 2.7.1

[41] R. Garg et al. Understanding the design-space of sparse/dense multiphase gnn dataflows on spatial accelerators. In *IPDPS*, 2022. 4.6

[42] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. Sparten: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 151–165, 2019. 4.6

[43] B. Graham. Fractional max-pooling. *arXiv preprint arXiv: 1412.6071 (2014)*, December 2014. 1.3.2.3

[44] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, sep 1978. 4.1

[45] H. Dwyer and H. C. Torng. An Out-of-Order Superscalar Processor with Speculative Execution and Fast, Precise Interrupts. In *Proceedings of the 25th IEEE/ACM International Symposium on Microarchitecture*, 1992. 1.1

[46] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254, June 2016. 4.6

[47] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv: 1510.00149v5 (2016)*, February 2016. 4.1

[48] Genc Hasan, Haj-Ali Ameer, Vighnesh Iyer, Amid Alon, Mao Howard, Wright John, Schmidt Colin, Zhao Jerry, Ou Albert, Banister Max, Shao Yakun Sophia, Nikolic Borivoje, Stoic Ion, and Asanovic Krste. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. *arXiv preprint arXiv:1911.09925*, 2019. 2.2.2

[49] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv: 1512.03385v1 (2015)*, December 2015. 2.1, 3.4.3, 4.4.2

[50] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. ExTensor: An accelerator for sparse tensor algebra. In *Proceedings of the*

52nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 319–333, 2019. (document), 1.4.3, 4.1, 4.2.1, 4.6

[51] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, February 2019. 1

[52] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, September 1997. 1.3.2.3

[53] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv: 1704.04861 (2017)*, April 2017. 2.1, 3.4.3

[54] HP Laboratories. CACTI 7.0: A Tool to Model Caches/Memories, 3D stacking, and off-chip IO. https://github.com/HewlettPackard/cacti. 4.4.4

[55] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: simulating shared-memory multiprocessors with ilp processors. *Computer*, 35(2):40–49, 2002. 1, 2.1

[56] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: Alexnet-level accuracy with 50x fewer parameters and $<$ 0.5mb model size. *arXiv preprint arXiv: 1611.10012 (2016)*, November 2016. 2.1, 3.4.3, 4.4.2

[57] Intel Corporation. *Intel Core Laptop Processors Comparision Chart*. 1.1

[58] International Technology Roadmap for Semiconductors. http://www.itrs.net/Links/2011ITRS/Home2011.htm. 1.1

[59] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 6th edition, 2019. (document), 1.1, 1.1

[60] Janssen, Curtis L., and et al. A simulator for large-scale parallel computer architectures. *IJDST vol.1, no.2*, pages 57–73, 2010. (document), 4.4.1

[61] Norman P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *44th Int'l Symp. on Computer Architecture*, 2017. (document), 1.2, 1.4, 1.4.1, 1.4.2, 2.1, 1, 2.1, 2.4.1, 2.4.1.3, 3.1, 3.1, 3.2.1, 3.5.1

[62] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Willians and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences. University of California at Berkeley, 2006. 1.1

[63] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 600–614, 2019. 4.6

[64] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. taco: a tool to generate tensor algebra kernels. *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 943–948, October 2017. 4.2.1

[65] Tushar Krishna, Hyoukjun Kwon, Angshuman Parashar, Michael Pellauer, and Ananda Samajdar. Data orchestration in deep learning accelerators. *Synthesis Lectures on Computer Architecture*, 15(3):1–164, 2020. 2.4

[66] Tushar Krishna, Hyoukjun Kwon, Angshuman Parashar, Michael Pellauer, and Ananda Samajdar. *Data Orchestration in Deep Learning Accelerators*, pages 1–164. July 2020. 3.1

[67] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *International Conf. on Neural Information Processing Systems (NIPS)*, pages 1106–1114, December 2012. 2.1, 3.4.3, 4.4.2

[68] Hyoukjun Kwon et al. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. *Int'l Symp. on Microarchitecture (MICRO)*, October 2019. (document), 1.4, 2.1, 2.1, 2.2.1, 2.7.1, 3.1, 3.1, 3.2.1, 4.1

[69] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Rethinking nocs for spatial neural network accelerators. In *Proceedings of the Eleventh IEEE/ACM International Symposium on Networks-on-Chip, NOCS 2017, Seoul, Republic of Korea, October 19 - 20, 2017*, pages 19:1–19:8, 2017. 3.1

[70] 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*, March 2018. (document), 1.4, 1.4.2, 2.1, 1, 2.1, 2.1, 2.2.1, 2.3.4, 2.4.1, 2.4.1.1, 2.4.1.3, 2.6.1.2, 3.1, 3.1, 3.2.1, 3.2.2, 3.3.1, 3.3.1, 3.3.3.2, 3.4.2, 4.1, 4.3, 4.3.1.2, 4.6

[71] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 1.3.2.2

[72] Ching-En Lee, Yakun Sophia Shao, Jie-Fang Zhang, Angshuman Parashar, Joel Emer, Stephen W Keckler, and Zhengya Zhang. Stitch-x: An accelerator architecture for exploiting unstructured sparsity in deep neural networks. In *SysML Conference*, volume 120, 2018. 4.6

[73] T.T. Lee and Soung-Yue Liew. Parallel routing algorithms in benes-clos networks. *IEEE INFOCOM '96. Conference on Computer Communications*, March 1996. 3.1

[74] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollar. Microsoft coco: Common objects in context. *arXiv preprint arXiv: 1405.0312v3*, February 2015. (document), 2.1

[75] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. *arXiv preprint arXiv: 1512.02325v5 (2015)*, December 2015. 2.1, 3.4.3, 4.4.2

[76] Wenyan Lu et al. FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks. *IEEE Int'l Symp. on High Performance Computer Architecture*, pages 553–564, May 2017. (document), 1.4, 1.4.2, 2.1, 2.1, 4.6

[77] Xin Lu, Xin Kang, Shun Nishide, and Fuji Ren. Object detection based on ssd-resnet. *IEEE 6th International Conference on Cloud Computing and Intelligence Systems (2019)*, December 2019. 4.4.2

[78] Tao Luo, Shaoli Liu, Ling Li, Yuqing Wang, Shijin Zhang, Tianshi Chen, Zhiwei Xu, Olivier Temam, and Yunji Chen. DaDianNao: A neural network

supercomputer. *IEEE Transactions on Computers*, 66(1):73–88, January 2017. 1.4, 1.4.1, 2.1, 3.1, 3.2.1

[79] Robert C. Martin. Design principles and design patterns. *objetcmentor*, April 2000. (document), 2.3

[80] Francisco Muñoz Matrínez, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. STONNE: A detailed architectural simulator for flexible neural network accelerators. In *2nd Workshop on Accelerated Machine Learning co-located with ISCA 2020 conference*, 2020. 1.5.1

[81] Francisco Muñoz Matrínez, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. Stonne: Enabling cycle-level microarchitectural simulation for dnn inference accelerators. *IEEE Computer Architecture Letters*, (01), July 2021. 1.5.1

[82] Francisco Muñoz Matrínez, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. STONNE: Enabling cycle-level microarchitectural simulation for dnn inference accelerators. In *IISWC*, 2021. 1.5.1, 2.7.1, 3.4.2, 4.4.1

[83] Francisco Muñoz Matrínez, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. Stift: A spatio-temporal integrated folding tree for efficient reductions in flexible dnn accelerators. *ACM Journal on Emerging Technologies in Computing Systems*, (01), May 2022. 1.5.1

[84] Francisco Muñoz Matrínez, Raveesh Garg José L. Abellán, Manuel E. Acacio, Clay Hughes, Siva Rajamanickam, and Tushar Krishna. SST-STONNE: Enabling cycle-level simulation of flexible spatial accelerators for dnns and gnns with a detailed memory hierarchy. In *Workshop on Modeling Simulation of Systems and Applications*, 2022. 1.5.1

[85] Francisco Muñoz Matrínez, Raveesh Garg José L. Abellán, Manuel E. Acacio, and Tushar Krishna. SST-STONNE: Enabling cycle-level simulation of flexible spatial accelerators for dnns and gnns with a detailed memory hierarchy. In *STONNE tutorial co-located with the 27th edition of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022. 1.5.1

[86] Francisco Muñoz Matrínez, Raveesh Garg José L. Abellán, Manuel E. Acacio, Michael Pellauer, and Tushar Krishna. Flexagon: A multi-dataflow sparse-sparse matrix multiplication accelerator for efficient dnn processing. In

Submitted to the 28th edition of the International Conference on Architectural Support for Programming Languages and Operating Systems. This paper is under review at the moment of writing this thesis, 2023. 1.5.1

[87] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. Mlperf training benchmark. *arXiv preprint arXiv:1910.01500*, 2019. 4.1

[88] Atefeh Mehrabi, Donghyuk Lee, Niladrish Chatterjee, Daniel J. Sorin, Benjamin C. Lee, and Mike O'Connor. Learning sparse matrix row permutations forefficient spmm on gpu architectures. *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 48–58, March 2021. 2.6.3.1

[89] Francisco Muñoz-Martinez, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. A novel network fabric for efficient spatio-temporal reduction in flexible dnn accelerators. In *To appear in 15th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 2021. 1.5.1

[90] N. P. Jouppi and S. J. E. Wilton. Tradeoffs in Two-Level On-Chip Caching. In *Proceedings of the 21st International Symposium on Computer Architecture*, 1994. 1.1

[91] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019. 4.1

[92] NVIDIA. *CUDA C Programming Guide v.9.0*. September 2017. 1.2

[93] Chigozie Enyinna Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv: 1811.03378 (2018)*, November 2019. 1.3.2.2

[94] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008. 1.2

[95] P. Marcuello, A. González and J. Tubella. Speculative Multithreaded Processors. In *Proceedings of the 12th International Conference on Supercomputing*, 1998. 1.1

[96] Subhankar Pal, Aporva Amarnath, Siying Feng, Michael O'Boyle, Ronald Dreslinski, and Christophe Dubach. Sparseadapt: Runtime control for sparse linear algebra on a reconfigurable accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 1005–1021, New York, NY, USA, 2021. Association for Computing Machinery. 4.6

[97] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *ISCA*, 2018. (document), 1.4.3, 4.1, 4.4.4, 4.6

[98] 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)*, pages 27–40, June 2017. (document), 1.4, 1.4.1, 2.1, 4.1, 4.6

[99] A. Peleg and U. Weiser. Mmx technology extension to the intel architecture. *IEEE Micro*, 16(4):42–50, 1996. 1.2

[100] Michael Pellauer et al. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. *International Conference on Architectural Support for Programmling Lenguages and Operating Systems (ASPLOS)*, pages 137–151, April 2019. 2.4.2

[101] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel paterns. *SIGARCH Comput. Archit. News*, 45(2):389–402, jun 2017. 4.6

[102] Eric Qin, Geonhwa Jeong, William Won, Sheng-Chun Kao, Hyoukjun Kwon, Sudarshan Srinivasan, Dipankar Das, Gordon E Moon, Sivasankaran

Rajamanickam, and Tushar Krishna. Extending sparse tensor accelerators to support multiple compression formats. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1014–1024. IEEE, 2021. 4.1

[103] Eric Qin, Geonhwa Jeong, William Won, Sheng-Chun Kao, Hyoukjun Kwon, Sudarshan Srinivasan, Dipankar Das, Gordon E. Moon, Sivasankaran Rajamanickam, and Tushar Krishna. Extending sparse tensor accelerators to support multiple compression formats. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1014–1024, 2021. 4.6

[104] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv: 1606.05250v3*, October 2016. (document), 2.1

[105] S.K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming simd extensions on the pentium iii processor. *IEEE Micro*, 20(4):47–57, 2000. 1.2

[106] V. J. Reddi et al. MLPerf inference benchmark. *47th International Symposium on Computer Architecture*, May 2020. 2.1, 3.4.3, 4.4.2

[107] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459. IEEE, 2020. (document), 1.4.2, 4.1, 4.1

[108] Kamil Rocki, Dirk Van Essendelft, Ilya Sharapov, Robert Schreiber, Michael Morrison, Vladimir Kibardin, Andrey Portnoy, Jean Francois Dietiker, Madhava Syamlal, and Michael James. Fast stencil-code computation on a wafer-scale processor. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020. 1.4.1

[109] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. 1.3.1

[110] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C.Berg, and L. FeiFei. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, December 2015. (document), 2.1

[111] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv:1910.01108 (2019)*, February 2019. 4.4.2

[112] Paul B. Schneck. *The CDC STAR-100*, pages 99–117. Springer US, Boston, MA, 1987. 1.2

[113] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv: 1409.1556v6 (2016)*, April 2016. 2.1, 3.4.3, 4.4.2

[114] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 766–780. IEEE, 2020. (document), 1.4.3, 4.1, 4.6

[115] Yifan et al. Sun. Mgpusim: Enabling multi-gpu performance modeling and optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*, page 197–209, 2019. (document), 1, 2.1

[116] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices. *arXiv:2004.02984 (2020)*, April 2020. 4.4.2

[117] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. Efficient processing of deep neural networks: A tutorial and survey. *arXiv preprint arXiv: 1703.09039v2 (2017)*, August 2017. (document), 1.6, 1.3.2.2, 1.3.2.3, 2.1, 3.1, 1, 2

[118] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017. 1.3.2.3

[119] Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. Accelergy: An architecture-level energy estimation methodology for accelerator designs. *International Conference On Computer Aided Design (ICCAD)*, November 2019. (document), 2.3.3

[120] Sam Xi et al. SMAUG: End-to-end full-stack simulation infrastructure for deep learning workloads. *ACM Trans. Archit. Code Optim.*, 17(4), November 2020. 2.2.2

[121] 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)*, pages 15–29, 2020. 2.7.1

[122] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. GAMMA: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 687–701, 2021. (document), 1.4.3, 4.1, 4.1, 4.1, 4.2.1, 4.3.1.2

[123] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. *SARA: Scaling a Reconfigurable Dataflow Accelerator*, page 1041–1054. IEEE Press, 2021. 4.6

[124] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. SpArch: Efficient architecture for sparse matrix multiplication. *International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–274, February 2020. (document), 1.4.3, 2.4.1.2, 4.1, 4.1, 4.1, 4.3.1.2, 4.4.4

[125] Zhongyuan Zhao, Hyoukjun Kwon, Sachit Kuhar, Weiguang Sheng, Zhigang Mao, and Tushar Krishna. mRNA: Enabling efficient mapping space exploration for a reconfigurable neural accelerator. *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 282–292, April 2019. 2.4.2, 3.1, 3.4.3

[126] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv: 1710.01878v2 (2017)*, October 2017. 2