Aceleración en hardware del producto escalar de punto fijo

Proyecto final

IPD432 - Diseño Avanzado de Sistemas Digitales

Alumno: Angel Cedeño Nieto

angel.cedeno@sansano.usm.cl

Abstract

En este trabajo se presenta la caracterización, diseño, implementación y análisis de un módulo para realizar la operación vectorial producto escalar. Se consideran vectores de tamaño variable N acotado por la cantidad de bloques DSPs (Digital signal processing slice) que tenga disponible la tarjeta objetivo y la frecuencia de operación requerida. Cada elemento de los vectores que se multiplican es un número con signo de n bits de ancho en el formato de punto fijo Qr.s. Se utiliza la estructura de un adder tree para realizar las sumas de los productos parciales de la operación vectorial de forma paralela. Se presentan dos versiones del módulo de producto escalar, una versión para minimizar latencia y una para mazimizar throughput.

1. Introducción

La operación vectorial de producto escalar, es el núcleo de muchas aplicaciones en áreas como róbotica, procesamiento de señales, procesamiento de imagenes, control automático, entre otras. Debido al creciente número de aplicaciones que exigen cada vez más velocidad de respuesta, nace la necesidad de herramientas de aceleración del producto escalar, lo cual está teniendo cada vez más apertura implementarlo en tarjetas FPGAs (Field-programmable gate array), las mismas que poseen potentes caraterísticas para explotar el paralelismo y el *deep pipelining* logrando frecuencias procesamiento en el orden de los 300-500 MHz. Por ejemplo, en Boland y Constantinides (2008) se usa el producto escalar implementado en la tarjeta FPGA Virtex5 LX 330T, para resolver un sistema de ecuaciones lineales de la forma $Ax = b$ y así abordar el problema de optimización denominado Minimum Residual Algorithm. En Jerez, Constantinides y Kerrigan (2011) se usa aceleración del producto escalar para resolver un problema de programación cuadrática (QP) con restricciones, en donde se usa el algoritmo de punto interior en el cual se requiere resolver la ecuación matricial $Ax = b$. En Jerez, Constantinides, Kerrigan y Ling (2011) se usa aceleración del producto escalar para el control MPC (Model Predictive Control) en donde nuevamente el producto escalar se usa para resolver el sistema $Ax = b$.

2. Formulación del problema

Por definición, el producto escalar entre dos vectores viene dado por la siguiente expresión:

$$\langle \mathbf{u}, \mathbf{v} \rangle = \sum_{i=1}^{N} u_i v_i = u_1 v_1 + u_2 v_2 + \cdots + u_N v_N = q \qquad (1)$$

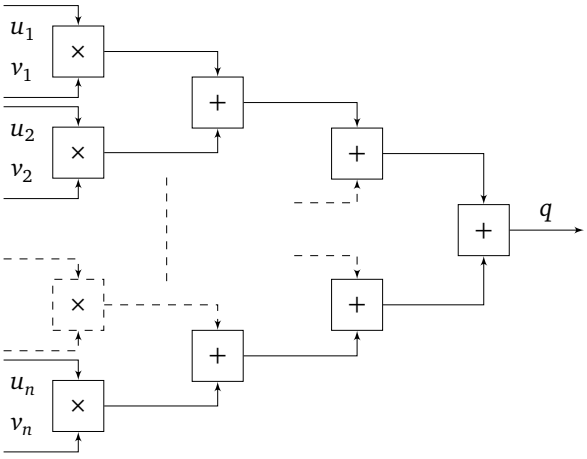

Fig. 1: Diagrama del producto escalar.

donde $\mathbf{u}, \mathbf{v} \in \mathbb{R}^N$ y $q \in \mathbb{R}$. Calcular $\langle \mathbf{u}, \mathbf{v} \rangle$ requiere el cálculo de $2N-1$ operaciones escalares, N productos y $N-1$ sumas. Esta operación puede realizarse explotando el paralelismo y el pipeline siguiendo el esquema mostrado en la Fig. 1 como se discute en Roldao y Constantinides (2010). La estructura mostrada tiene dos etapas, la primera etapa es la operación de producto entre cada par de elementos correspondientes de los dos vectores $\mathbf{u}$ y $\mathbf{v}$, la segunda etapa es un adder tree que realiza la suma de todos los productos parciales de la primera etapa en forma paralela, es decir toma pares de elementos del vector de N productos y los suma generando otro vector $N_1 = N/2$ de sumas parciales, luego realiza el mismo procedimiendo y se obtiene otro vector de $N_2 = N_1/2$ sumas parciales, y se continua así hasta llegar a la suma final de dos elementos obteniendose el resultado q.

3. Aritmética de punto fijo

La aritmética de punto fijo es la manipulación de cantidades cuya representación está limitada por un número constante de bits. Una de las ventajas es que las operaciones de la aritmética de punto fijo son esencialmente operaciones entre enteros, es decir, que operar con cantidades de punto fijo se realiza de la misma forma independiente de si las cantidades binarias representen números enteros o fraccionarios o una combinación de ambos. Para más información ver Corradini, Maksimović, Mattavelli y Zane (2015).

3.1. Representación de números con signo

Una de las formas de representar números positivos y negativos es utilizar el complemento. Los complementos en el sistema binario son el complemento restringido y el complemento verdadero llamados tambien complemento a 1 y complemento a 2 respectivamente. En el caso del sistema de numeración binaria, se usa el bit más significante para representar el signo, el 0 indica el signo positivo y el 1 el signo negativo.

En este trabajo se usa la representación en complemento verdadero o a 2 debido a que es el más utilizado hoy en día en microcontroladores, DSPs, y microprocesadores.

3.2. Complemento verdadero a 2

La representación en complemento verdadero de un numero binario es un sistema posicional de base 2 capaz de codificar números positivos y negativos con única representación del cero, a diferencia del complemento a 1 que tiene doble representación del cero lo cual puede ser una complicación si no se considera en el diseño. Sea w un número binario de n-bits codificado en complemento a 2:

$$w = [b_{n-1} b_{n-2} \dots b_1 b_0]_2 \tag{2}$$

donde el bit mas significativo b_{n-1} corresponde al signo, entonces el valor en el sistema de numeración decimal es:

$$w = -b_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times 2^i, \qquad b_i \in \{0,1\} \tag{3}$$

El rango de números positivos y negativos que se pueden ser representados con n bits es:

$$-2^{n-1} \leq w \leq 2^{n-1} - 1 \tag{4}$$

donde el número -2^{n-1} es el número más negativo y $2^{n-1} - 1$ es el número más positivo representables con n bits. Por ejemplo, con $n = 3$ bits se obtiene un rango que va desde -4 hasta 3, como se muestra en la representación circular de la Fig. 2.

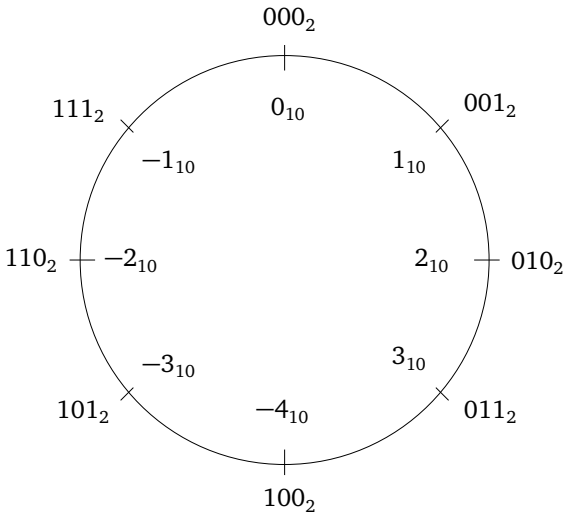

Fig. 2: Representación en complemento a 2 de un sistema aritmético de 3 bits.

3.3. Formato de punto fijo Qr.s

El formato Qr.s es un formato para representar números de punto fijo, donde r indica la cantidad de bits de la parte entera y s indica la cantidad de bits para la parte fraccionaria. Este formato se usa en dispositivos (hardware) que no poseen unidad de punto flotante o para aplicaciones que requieren resolución constante. Por ejemplo, un número representado en el formato Q8.4 es un número de 12 bits donde 8 bits corresponden a la parte entera y 4 bits a la parte fraccionaria. En la Fig. 3 se muestra un ejemplo de esta representación.

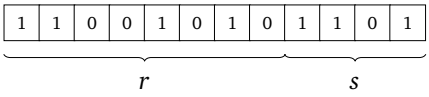

Fig. 3: Representación de un número binario en formato Qr.s - Q8.4 .

El valor decimal de la parte fraccionaria representada en el formato Qr.s viene dada por la siguiente expresión:

$$d_{\text{fraccionaria}} = \sum_{i=1}^{s} b_i \times 2^{-i} \tag{5}$$

donde b_1 es el bit mas significativo de la parte fraccionaria. La ecuación (5) indica que después del punto binario cada posición es ponderada con una potencia negativa de 2, y dado que la cantidad de bits es finita la resolución que se tiene es limitada por:

$$rs = \frac{1}{2^s} \tag{6}$$

Por ejemplo, con 4 bits para la parte fraccionaria se tiene una resolución de 0.0625 con lo cual el mayor número decimal fraccionario que se puede representar con 4 bits es 0.9375. Si se quiere aumentar la resolución hay que aumentar el

número de bits, por ejemplo con 8 bits se tiene una resolución de 0.00390625 y el mayor número decimal fraccionario representable con 8 bits es 0.99609375.

Cuando el número representado en el formato Qr.s es con signo, el bit más significativo de la parte entera es el bit del signo, y por lo tanto el rango de números que pueden ser representados en el formato Qr.s cambia respecto a los números sin signo, de la siguiente manera:

$$-2^{r-1} \leq w \leq 2^{r-1} - \frac{1}{2^s} \tag{7}$$

Por ejemplo, sean los números con signo dados por Q8.4, entonces el rango es -128 hasta 127.9375, y para el ejemplo mostrado en la Fig. 3 se tiene: $11001010.1101_2 = -53.1875$.

3.4. Extensión de signo

Al trabajar con el formato Qr.s, la cantidad de bits para la representación de un número esta definida y constante, por lo cual toda cantidad debe ser representada con exactamente $r+s$ bits, entonces números como por ejemplo 1010.1101_2 deben ser completados hasta estar en el formato Qr.s deseado, si se trabaja con Q8.4 el número debe completarse de tal manera que los m bits adicionales no alteren su valor, entonces lo que se hace es aumentar m réplicas del bit del signo en la parte más significante, esto es debido a que la contribución del signo puede escribirse como:

$$-b_{n-1} \times 2^{n-1} = -b_{n-1} \times 2^n + b_{n-1} \times 2^{n-1} \tag{8}$$

y por lo tanto se pueden aumentar arbitrariamente tantos bits del signo como se requieran sin que se altere su valor:

$$1010.1101_2 = -5.1875_{10}$$
$$11111010.1101_2 = -5.1875_{10}$$

3.5. Valor decimal de un número con signo en formato Qr.s

Como se discutió anteriormente, una vez que una cantidad es representada en formato Qr.s el tamaño en bits esta completamente definido, y el valor que representa está determinado por la posición que ocupa el punto binario. Esto indica que para una misma secuencia de bits dependiendo del lugar en donde este el punto binario se tienen valores decimales diferentes, aunque cualquier operación matemática que se realice es exactamente igual no importa donde se coloque el punto binario, o lo que esta representación signifique para el programador, así:

$$Q4.4: \quad 0101.1100_2 = 5.75$$
$$Q3.5: \quad 010.11100_2 = 2.875$$
$$Q2.6: \quad 01.011100_2 = 1.4375$$

Esta característica es muy útil cuando se tiene un bloque de arimética de punto fijo, ya que las operaciones como suma, resta, multiplicación se realizan siempre de la misma forma, pero el programador puede mover a la derecha el punto binario para poder representar números enteros grandes pero con poca resolución en la parte fraccionaria y mover el punto binario a la izquierda para representar números enteros pequeños pero aumentando la resolución de la parte fraccionaria, por ejemplo, usando 24 bits:

$$Q20.4: \quad -524288 \leq \text{entera} \leq 524287$$
$$0 \leq \text{fraccionaria} \leq 0.9375$$
$$Q4.20: \quad -8 \leq \text{entera} \leq 7$$
$$0 \leq \text{fraccionaria} \leq 0.999999046325684$$

4. Operación vectorial de producto escalar

El diseño del módulo de producto escalar que se describirá a continuación considera que:

- Se trabajará con tarjeta Nexys 4 DDR del fabricante Digilent, que incorpora el Artix-7™ Field Programmable Gate Array (FPGA) de Xilinx® con número de parte XC7A100T-1CSG324C. Para más información ver Nexys 4 DDR Reference Manual (2018). Este dispositivo posee en total 240 bloques DSPs que permiten realizar el producto entre dos catidades de 25x18 bits respectivamente, ver (7 Series DSP48E1 Slice User Guide (2018)).

- El orden del producto escalar estará entre $2 \leq N \leq 240$, es decir, se trabajará en cada caso con dos vectores $\mathbf{a}, \mathbf{b} \in \mathbb{R}^{N\times 1}$, con $N \in \mathbb{N}$.

- El número de bits para representar cada elemento de los vectores **a** y **b** será un multiplo de 8 (Byte), debido a que ese es el ancho de la palabra que se puede transmitir por el puerto serie, y por lo tanto $n \in \{8, 16, 24\}$.

Tanto N como n se pueden aumentar si se cuenta con otras tarjetas que tengan características mejores que la actual, específicamente con más bloques DSPs y de mejores prestaciones.

4.1. Diseño en HDL

El diseño presentado a continuación está basado en el diagrama de alto nivel mostrado en la Fig. 4, donde además del producto escalar se requiere de una serie de módulos adicionales para poder probar su funcionamiento. Se tiene una inferfaz que permite comunicar la tarjeta FPGA (desde ahora denominada *device*) con el computador (desde ahora denominado *host*) mediante comunicación serial. Desde el host se envían los dos vectores **a** y **b** los cuales pueden tener tamaño N y cada uno de sus elementos pueden tener ancho n. Los bytes que conforman cada elemento de los vectores son empaquetados en un *packed array* de $n/8$ bytes, y cada paquete es almacenado en un espacio de un *unpacked array* de largo N. Ver **Appendix A**. Se recibe además un comando que es decodificado para gatillar una acción definida según la Tabla 1.

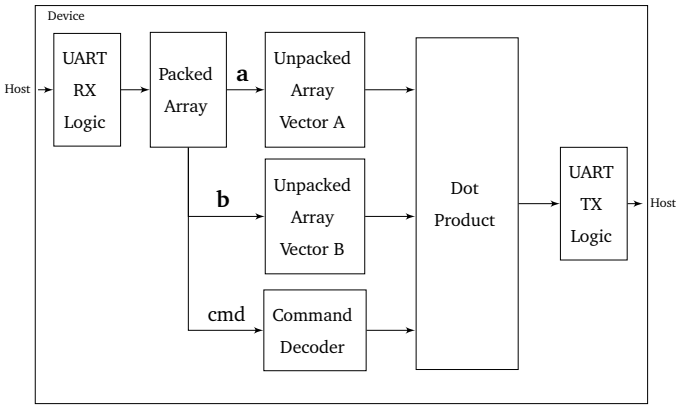

Fig. 4: Diseño de alto nivel del producto escalar incluida la interfaz entre el device y el host

Tabla. 1: Comandos y Datos para comunicación entre el host y el device.

Comando	Datos	Acción
1	Vector A	Guardar Vector A
2	Vector B	Guardar Vector B
3	Vector A y B	producto escalar entre A y B

Cuando el comando es 3 se envía al host el resultado de la operación en una trama que tiene $2n+8$ bits o $(2n+8)/8$ bytes. Este número de bits es debido a que el producto entre dos números de n bits cabe en $2n$ bits y la suma de N números de $2n$ bits podría caber en $2n$ bits dependiendo de N y de n, pero cuando ocurra un exceso (overflow) debe ser considerado para obtener el valor correcto de la suma, por lo tanto de usarán 8 bits adicionales para guardar el exceso, anticipándose a que N y n puedan llegar a ser muy grande, entonces la suma final tendrá $2n+8$ bits.

El diseño en HDL es de tipo modular, esta compuesto de un módulo *top level* donde se instancian todos los módulos internos. En la Fig. 5 se observa el módulo principal denominado *Main*, el cual tiene las siguientes entradas y salidas:

- La entrada de reloj, en la que ingresa la señal del oscilador de 100MHz que trae incluido la tarjeta FPGA.
- La entrada del puerto UART por donde ingresan los datos enviados desde el host.
- La salida del puerto UART por donde salen los datos enviados desde el device al host.

Los módulos instanciados al interior del módulo principal son los siguientes:

- **Clock.** Este módulo es una instancia del IP *Clocking Wizard*, que recibe una señal de reloj de una determinada frecuencia y genera señales de relojes de frecuencias personalizadas por el usuario, en este caso particular, el módulo Clock recibe la señal del oscilador externo de 100MHz, y entrega otra señal de la frecuencia deseada, limpia de ruidos.
- **Uart Port.** Este módulo permite setear el baud rate (baudios) al que se reciben y transmiten los datos desde el host o hacia el host, el valor que trae por defecto es 115200 baudios. Además permite configurar la frecuencia del reloj a la que trabajará el módulo.
- **Flujo.** Este módulo es básicamente un multiplexor, que se encarga de encaminar la información proveniente desde el host, ya sea a los módulos encargados de guardar estos datos en los *packed* o *unpacked arrays*, o al módulo que decodifica el comando recibido, para posteriormente gatillar las operaciones pertinentes.
- **Cmd.** Este módulo se encarga de diferenciar entre los comandos que llegan. Cuando llegan los comandos para guardar los vectores en la memoria habilita o desabilita los módulos *Save A* o *Save B* de tal manera de guardar el dato donde corresponde. Cuando recibe el comando que representa la operación de producto escalar sobre los vectores, envía dicho comando al módulo Trigger.
- **Trigger.** Este módulo recibe el comando que indica la operación de producto escalar sobre los vectores **a** y **b**, y genera un pulso que dura solo un ciclo de reloj, esto se hace con el objetivo de gatillar la operación indicada en el módulo Dot_Produc, ya que la operacion estan implementad en una máquina de estado de tal manera que se requiere un pulso para salir del estado inicial; y comenzar la operación solicitada, y una vez que termina su tarea debe volver al estado inicial y quedarse allí hasta que llegue un nuevo comando que indique un nuevo comienzo en la operación.
- **Save A.** Recibe los datos correspondiente al vector **a** que se transmiten desde el host al device y los almacena en la memoria correspondiente. Los elementos de **a** tienen un largo de n bits.
- **Save B.** Similar a Save A, con los datos correspondientes al vector **b**.
- **Product.** Recibe los dos arrays donde se almacenan los vectors **a** y **b**. Realiza la multiplicación:

$$P = \mathbf{a}_i * \mathbf{b}_i \qquad \text{con} \quad i = 0, \ldots, N-1 \tag{9}$$

  El resultado de la multiplicación es otro vector del mismo tamaño que **a** y **b**, y cada elemento en P tiene una largo de $2n$ bits.
- **Adder Tree.** Recibe el array P donde se almacena el resultado de la multiplicación de cada elemento de **a** y **b** y realiza la suma de sus elementos. El resultado de la suma es un número que tiene un largo de $2n+8$ bits. Cuando el resultado de la suma no requiere todos los 8 bits adicionales que se han considerado para el exceso

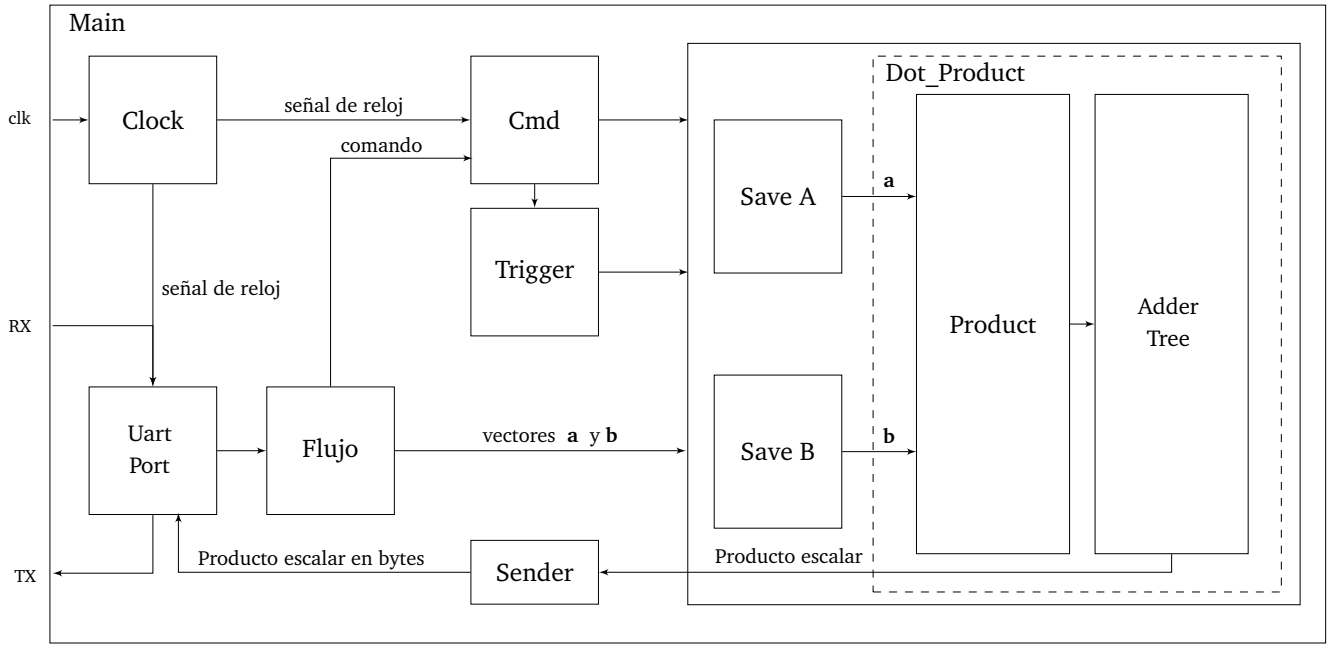

Fig. 5: Diagrama de flujo de información del circuito completo.

estos bits restantes son poblados realizando una extensión de signo de tal manera de no modificar su valor y mantener la estructura de $2n+8$ bits.

- **Sender.** Este módulo recibe el resultado del producto escalar y los divide en $(2n+8)/8$ bytes y los envía por el puerto serie hasta el device.

4.2. Diseño para minimizar latencia

En aplicaciones donde el tiempo de respuesta es crítico, es útil tener una versión del módulo de producto escalar que este orientado a minimizar latencia aún a costa de reducir la frecuencia de operación. Cuando se quiere minimizar latencia se incurre en acumular demasiado lógica combinacional entre Flip-Flops, esto se debe principalmente a la variación del largo y ancho de los vectores **a** y **b**, lo cual tienen una injerencia directa en la frecuencia de operación y en la cantidad de DSPs que se usan por cada multiplicación. En la Tabla 2 se resume como varía la frecuencia de operación en función del aumento del número de elementos de los vectores **a** y **b**, para los tres valores de n considerados. En la Fig. 6 se grafican dichos valores para observar la tendencia que tiene la frecuencia al aumentar los parámetros N y n.

Claramente se observa que a medida que el tamaño de los vectores es mas grande la frecuencia de operación disminuye. En la tarjeta FPGA objetivo se ha logrado $f = 250$ MHz como máxima frecuecia de operación en la versión para minimizar latencia cuando el orden del producto escalar es $N = 2$, y $f = 50$ MHz como máxima frecuencia de operación cuando $N = 240$, lo cual es un resultado esperado ya que al aumentar N aumenta la cantidad de lógica combinacional entre Flip-Flops, y para cumplir los requerimientos de timing es necesario disminuir la frecuencia de operación. Es también nece-

Tabla. 2: Variación de la frecuencia de operación en función de N y n.

	frecuencia en MHz								
n\N	**2**	**20**	**40**	**60**	**80**	**100**	**150**	**200**	**240**
8	250	100	65	60	60	60	60	50	50
16	250	90	75	70	70	60	50	50	50
24	170	90	70	70	60	60	-	-	-

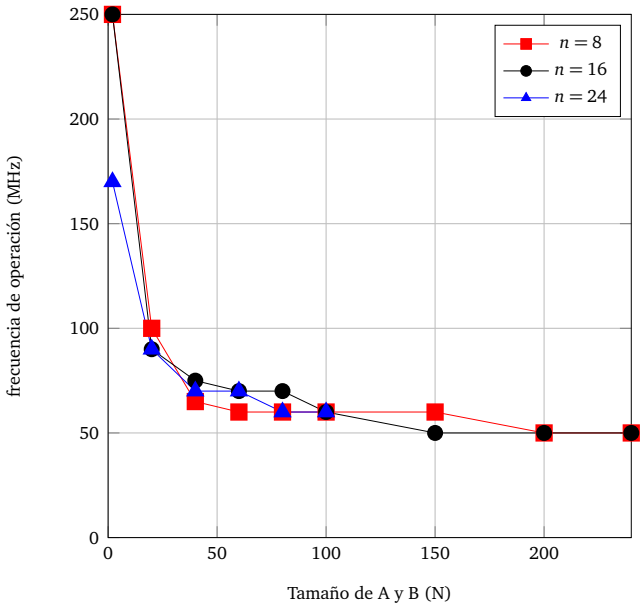

Fig. 6: Variación de la frecuencia de operación en función de N y n.

Tabla. 3: Uso de recursos en función de N y n.

n	8			16			24		
N	LUT	FF	DSP	LUT	FF	DSP	LUT	FF	DSP
2	139	181	2	210	247	2	223	327	4
20	514	263	20	837	248	20	1157	655	40
40	894	343	40	1539	259	40	2183	1126	80
60	1279	305	60	2238	355	60	3198	1630	120
80	1665	346	80	2932	392	80	4225	2047	160
100	2045	393	100	3644	435	100	5243	2531	200
150	3001	328	150	5394	524	150	-	-	-
200	3970	373	200	7146	621	200	-	-	-
240	4727	406	240	8571	692	240	-	-	-

sario realizar un análisis del uso de recursos consumidos, los cuales son función directa del tamaño de los vectores y del ancho de sus elementos. En la Tabla 3 se resume el uso de recursos como son LUTs, FFs Y DSPs a medida que se aumenta N y n.

Es importante notar que por el hecho de que la versión del producto escalar es para minimizar latencia el uso de los Flip-Flops no es elevado comparado con el total disponible, en el caso más extremo analizado, $N = 100$ y $n = 24$ se han usado 2531 de 126800 Flip-Flops que es el 2% del total disponibles. Con respecto a los LUTs se han usado 5243 de 63400 que es el 8.27% del total disponible. También hay que recalcar que el uso de DSPs depende directamente del ancho que tienen los elementos que se multiplican. Dado que los DSPs en la tarjeta objetivo permiten multiplicar dos elementos de 25x18 bits, cuando los elementos en los vectores tienen 8 y 16 bits de ancho entonces se usa un DSP por producto, lo cual indica que se podrían realizar 240 productos (si la frecuencia lo permite y no se incurre en problemas de timing), cuando los elementos en los vectores tienen 24 bits entonces se requieren 2 DSPs por lo tanto se podrían realizar 120 productos, y cuando $n > 24$ se requieren cada vez más DSPs por ejemplo cuando el ancho es 48 se requieren 9 DSPs por lo tanto se pueden realizar 26 productos. En la Tabla Tabla 2 y en la Tabla 3 , cuando $n = 24$ y $N \in \{150, 200, 240\}$ no se tienen valores de frecuencia ni valores de recursos usados debido a que se incurre en un error en la síntesis e implementación porque el uso de bloques DSPs necesarios para el orden del producto escalar es superior al número de DSPs disponibles en la tarjeta objetivo. Si se desea aumentar tanto en tamaño de los vectores como el ancho de cada elemento es necesario usar una tarjeta con más bloques DSPs.

4.3. Diseño para maximizar throughput

La versión para maximizar throughput está enfocada en aumentar el número de resultados por unidad de tiempo, lo cual se logra realizando lo que se conoce como *fully pipelined*, con esto se consigue que el hardware este trabajando en todos los ciclos de reloj similar a una línea de ensamblaje. Una de las ventajas que tiene realizar pipeline es que se puede

Tabla. 4: Variación de la frecuencia de operación en función de N y n.

	frecuencia en MHz								
n\N	**2**	**20**	**40**	**60**	**80**	**100**	**150**	**200**	**240**
8	320	310	290	280	270	260	260	240	230
16	310	270	260	250	240	220	210	200	190
24	250	230	220	210	210	200	200	170	160

aumentar considerablemente la frecuencia de operación respecto a la versión para minimizar latencia, ahora el objetivo es colocar la mayor cantidad de Flip-Flops entre la lógica combinacional de tal manera que entre Flip-Flop y Flip-Flops haya la mímima cantidad posible de elementos combinacionales. Como en el caso anterior, en la Tabla 4 se resume como varia la frecuencia de operación en función del aumento del número de elementos de los vectores **a** y **b**, para los tres valores de n considerado. En la Fig. 7 se grafican dichos valores para observar la tendencia que tiene la frecuencia al aumentar los parámetros N y n. Se observa que a medida que el tamaño de los vectores es mas grande la frecuencia de operación disminuye. En la tarjeta FPGA objetivo se ha logrado $f = 320$ MHz como máxima frecuecia de operación en la versión para maximizar throughput cuando el orden del producto escalar es $N = 2$, y $f = 160$ MHz como máxima frecuencia de operación cuando $N = 240$.

Es importante mencionar que en la implementación de la versión para maximizar throughput se ha usado el bloque IP *Multiplier* incluido en la libreria de Vivado Desing Suite 2018.2, con la siguiente configuración:

- Data Type: Signed (A y B) .
- Width: n.
- Multiplier Construction: Use Mults.
- Optimization Options: Area Optimized.
- Pipeline Stages: Valor optimo sugerido por Vivado.

En el campo *Pipeline Stages*, es posible no usar los valores recomendados por la herramienta, pero si el número de pasos pipeline elegidos es menor que el recomendado, se usa en la multiplicación más de un bloque DSPs según sea el valor de n, por ejemplo con $n = 24$ se usan dos DSPs por producto, si se elige un número mayor, aunque este valor no cambia el resultado de la operación del producto escalar si modifica el número de Flip-Flops usados y además usa un solo bloque DPS por producto. Se puede escoger hasta un máximo de 30 pasos de Pipeline, según se necesite.

Como en el caso anterior se realiza el análisis del uso de recursos, los cuales son función directa del tamaño de los vectores y del ancho de sus elementos. En la Tabla 5 se resume el uso de recursos como son LUTs, FFs Y DSPs a medida que se aumenta N y n. Se puede observar que ahora el uso de Flip-Flops es muy elevado respecto al uso en la

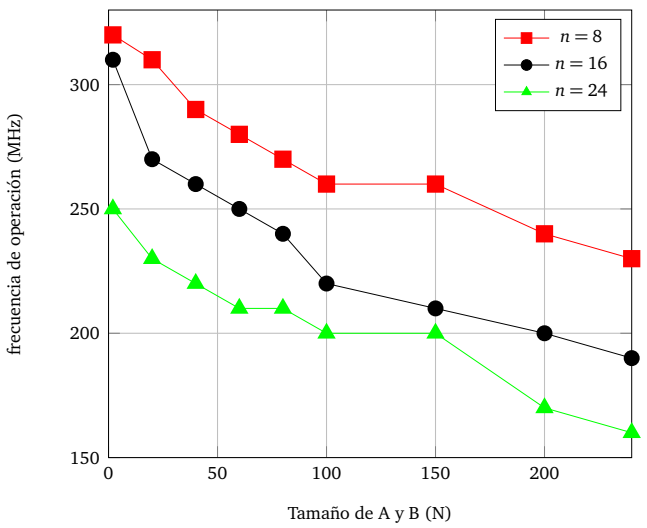

Fig. 7: Variación de la frecuencia de operación en función de N y n.

Tabla. 5: Uso de recursos en función de N y n.

n	**8**			**16**			**24**		
N	LUT	FF	DSP	LUT	FF	DSP	LUT	FF	DSP
2	180	276	2	265	439	2	644	1063	2
20	842	1517	20	1479	2845	20	5298	8971	20
40	1534	2837	40	2824	5449	40	10455	17651	40
60	2241	4137	60	4159	8009	60	15610	26275	60
80	2954	5479	80	5500	10647	80	20778	35009	80
100	3664	6778	100	6857	13207	100	25969	43637	100
150	5404	10138	150	10199	19812	150	38844	65493	150
200	7185	13390	200	13572	26209	200	37389	71444	200
240	8571	16027	240	16253	31419	240	44814	85688	240

versión para minimizar latencia. En el caso más extremo analizado, es decir cuando $N = 240$ y $n = 24$ se han usado 85688 de 126800 Flip-Flops que es el 67.58% del total disponibles. Con respecto a los LUTs se han usado 44814 de 63400 que es el 70.68% del total disponible. En los casos cuando $N \in \{200, 240\}$ se ha probado usar el número de pasos pipeline recomendado por Vivado, y se nota como el uso de LUT disminuye respecto al uso cuando $N = 150$ en donde se usó 30 pasos pipeline. Además cabe recalcar que cuando se usa el IP de multiplicación que trae la herramienta de Vivado y se realiza pipeline el número de bloques DSPs usados en cada multiplicación ya no varía con el ancho de los elementos que se multiplican como sucedió en la vesión para minimizar latencia, es por esto que ahora incluso cuando $n = 24$ se ha usado un solo DSP por multiplicación permitiendo probar el caso de $N = 240$ elementos por vector y $n = 24$ bits por cada elemento.

5. Simulación y Resultados

Para realizar la prueba del correcto funcionamiento del módulo de producto escalar se ha creado en Matlab un script llamado `Dot_Product` que genera 10000 parejas de vectores **a** y **b** aleatorios en un rango dado por (4) con N y n variables en los rangos definidos anteriormente. Se convierten dichos vectores (elemento a elemento) a su representación en complemeto a 2, esto debido que en el módulo de producto escalar se considera que los elementos de **a** y **b** son números con signo. Se separa la secuencia de bits en $n/8$ bytes para posteriormente enviarlos por el puerto serie al device y recibir el resultado. Se reciben $(2n + 8)/8$ bytes. Se obtiene su representación en complemento a2 para posteriormente obtener el valor decimal. Se calcula el error como la diferencia entre el valor calculado en el device y el calculado en el host. Se realizó esta prueba para todos los casos analizados (todas las combinaciones de N y n mostradas y analizadas en la Tabla 3 y en la Tabla 5 para las dos versiones del producto escalar). En todos los casos se obtuvo una diferencia de cero.

6. Trabajo Futuro

Muchas mejoras pueden ser implementadas en este proyecto. La primera de ellas es para abordar el problema de tiempo que supone el envío de los vectores al device y el envío del resultado desde el device hasta el host, tarea que actualmente se realiza usando la interfaz Uart a una velocidad de transmisión de 115200 baudios. Una alternativa sería reeemplazar el puerto serie por la interfaz Ethernet lo cual significaría un aumento significativo en la velocidad de transmisión ya que esta interfaz permite obtener velocidades de transmisión entre 10/100 Mb/s lo cual podría justificar el uso del módulo en una aplicación más exigente en la rapidez con la que tiene disponible el resultado del producto escalar.

También se puede explorar el uso del bloque de producto escalar en aplicaciones embedidas donde los datos se adquieran con la misma tarjeta FPGA y así evitar el tiempo perdido en la transmisión desde un computador o algún otro dispositivo.

7. Conclusiones

Se ha presentado el desarrollo del módulo para la operación de producto escalar entre dos vectores acelerado por hardware explotando las capacidades de paralelismo de las tarjetas FPGAs. Los vectores considerados tienen tamaño N y cada elemento es un número con signo de ancho n. Se ha realizado el análisis de como varía la frecuencia de operación según como varía el orden del producto escalar y el ancho de cada elemento.

Se han implementado dos versiones del módulo de producto escalar, la primera de ellas con el objetivo de minimizar latencia y la segunda para mazimizar el throughput. Las dos

versiones se han sido parametrizadas de tal manera que resulte fácil cambiar el tamaño de los vectores y el ancho de sus elementos. Esta última característica permite usar el módulo con cualquier valor de los parámetros N y n, simplemente se requiere de otra tarjeta con mas recursos.

Se encontró que en la tarjeta Nexys 4 DDR la máxima frecuencia de operación en la versión para minimizar latencia para $N = 2$ es $f = 250$ MHz y para $N = 240$ es $f = 50$ MHz. En la versión para maximizar throughput la máxima frecuencia de operación para $N = 2$ es $f = 320$ MHz y para $N = 240$ es $f = 160$ MHz.

Se ha analizado como varía el uso de bloques DSPs para la multiplicación en función del uso o no de pipeline. Se encontró que si se usa pipeline no importa el ancho de los factores que se multiplican se usa un solo bloque DSP por producto, lo que permite aumentar el tamaño N hasta el máximo número de DSPs disponibles, caso contrario mientras aumenta n aumenta la cantidad de bloques DSPs que se usan en cada mulipliación disminuyendo N considerablemente.

Appendix A. Packed y Unpacked Arrays

SystemVerilog permite manipular facilmente datos en forma de vectores y matrices, a través de los denominados Packed and Unpacked Arrays. Estos arreglos permiten almacenar y acceder a los datos mediante un índice tal como si fuese un programa de alto nivel como por ejemplo Matlab. El término *Packed Array* se usa para referirse a las dimensiones declaradas antes del nombre de la variable. El término *Unpacked Array* se usa para referirse a las dimensiones declaradas después del nombre de la variable.

```
bit [7:0] c1; // packed array
real u [7:0]; // unpacked array
```

Packed Array. Es un mecanismo para dividir un vector en sub-campos y poder acceder a ellos como si se tratara de los elementos de una matriz, por ejemplo:

```
[1:0][7:0] a;  // packed array
```

El vector a de 16 bits contiguos es tratado como un vector que contiene dos elementos de 8 bits cada uno, y se puede acceder a sus elementos de la siguiente forma:

```
A=a[0]; // firts 8-bits of packed array a
B=a[1]; // last 8-bits of packed array a
```

Packed arrays solo pueden ser creados con datos los siguientes tipos de datos: de un solo bit (bit, logic, reg), enumerados, recursivos y cualquier otro tipo de estructuras de datos empaquetados.

Unpacked Array. Funciona de la misma manera que los packed arrays pero pueden ser de cualquier tipos de datos, y además se diferencian en la forma que se accede a los datos. Los packed arrays permiten acceder a todos los bits de una vez o en sub-campos, y los unpacked array permiten acceder a cada elemento seleccionando individualmente el número del indice.

```
[1:0] a [7:0];  // unpacked array
```

```
// PARÁMETROS DEL PRODUCTO PUNTO (N, n , V)
// Tamaño de los dos vectores A y B que se procesarán
localparam N=240;
// Define el ancho en bits de cada elemento de A y B
localparam n=8;
// V=0 Minimizar latencia; V=1 Maximizar throughput
localparam V=1;
```

Fig. B.8: Cambiar los parámetros y versión del producto escalar

Empaquetar y desempaquetar datos se puede realizar usando el operador de desplazmiento por ejemplo para convertir un unpacked array en un vector:

```
logic [7:0] unpacked_a [3:0];
logic [31:0] vector_b;
vector_b <= {<<8{unpacked_a}};
```

y para convertir un vector en un unpacked array se puede realizar lo siguiente:

```
logic [31:0] Res
logic [7:0] data [3:0];
logic [7:0] temp [3:0];
assign {>>{temp}} = Res;
assign {<<byte{data}} = temp;
```

lo indicado en estos dos ejemplos fue clave en el desarrollo de este proyecto ya que permitió empaquetar los bytes que llegan del host en un packed array de tamaño n para luego ser guardado en un unpacked array de N elementos.

Appendix B. Personalizar el módulo de producto escalar

Para cambiar el tamaño N de los vectores A y B, el ancho de n de cada elemento, y la versión que se desea implementar, ubicar en el archivo *Main.sv* la sección mostrada en la Fig. B.8 y colocar los valores deseados, considerando que $V = 0$ implementa la versión para minimizar latencia y $V = 1$ implementa la versión para maximizar el throughput.

References

Boland, David & Constantinides, George. (2008). *An FPGA-based implementation of the MINRES algorithm*. International Conference on Field Programmable Logic and Applications, Heidelberg, 2008, pp. 379-384. doi: 10.1109/FPL.2008.4629967

Corradini, Luca. Maksimović, Dragan. Mattavelli, Paolo & Zane, Regan (2015). *Digital Control of high-frequency switched-mode power Converters*. John Wiley & Sons, Inc.

IEEE Computer Society, IEEE Standards Association Corporate Advisory Group (2017). *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*.

Jerez, Juan. Constantinides, George. & Kerrigan, Eric. (2011). *An FPGA Implementation of a Sparse Quadratic Programming Solver for Constrained Predictive Control*. ACM 978-1-4503-0554-9/11/02

Jerez, Juan. Constantinides, George. Kerrigan, Eric. & Ling, Keck-Voon. (2011). *Parallel MPC for Real-Time FPGA-based Implementation*. 978-3-902661-93-7/11 2011 IFAC

Roldao Lopes, Antonio & A. Constantinides, George. (2010). *A Fused Hybrid Floating-Point and Fixed-Point Dot-Product for FPGAs*. 5992. 157-168. 10.1007/978-3-642-12133-3_16.

Xilinx (2018) *Nexys 4 DDR Reference Manual*. Disponible en: https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual

Xilinx (2018) *7 Series DSP48E1 Slice User Guide*. Disponible en: https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf