Facultad
de
Ciencias

ACELERADORES DE CÓDIGO BASADOS EN FPGA: QUÉ SON Y CÓMO HACERLOS

(FPGA-based code accelerators: what are they and how to build one)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Jose Luis Vázquez Gutiérrez

Director: Eugenio Villar Bonet

Co-Director: Pablo Pedro Sánchez Espeso

Septiembre - 2016

Desde aquí quiero expresar mi agradecimiento a los compañeros del grupo de microelectrónica de TEISA que también han ayudado en la realización de este trabajo.

Indice

Introducción

Seguramente todos hemos pensado alguna vez algo similar a “¿por qué será tan lento mi PC?”, sea porque hemos utilizado uno más actualizado, porque nuestro sistema de refrigeración requiere una limpieza o porque tenemos demasiadas pestañas abiertas en el navegador de internet. Sea la razón que sea, me quiero centrar en el caso de haber utilizado un computador moderno, probablemente la causa más extendida.

En otros campos de la computación, ese fue durante mucho tiempo el sistema de aceleración de código: esperar a un computador más nuevo, ya que en unos meses llegará un nuevo chip con una frecuencia de reloj mucho más elevada. Esta estrategia funcionó durante mucho tiempo, pero llegó un día en el que no resultaba viable debido al elevado consumo de energía[1]. Al fin y al cabo, la energía que consume un procesador es directamente proporcional a la frecuencia de su reloj[1], por lo que llegó el día en el que no se disponía de sistemas de refrigeración lo suficientemente avanzados para soportar una mayor frecuencia.

La forma de conseguir programas más rápidos a partir de ese momento pasaba por realizar más instrucciones en paralelo, para poder conseguir mayor capacidad de cómputo con menor frecuencia. Primero con CPUs multinúcleo y después con GPGPU se fomentó la construcción de código paralelo o paralelizable, para poder explotar estas capacidades.

Con el tiempo se vio que siguen siendo soluciones con un consumo energético relativamente elevado cuando hablamos de HPC y de sistemas de datacenter, por lo que algunos valientes se han decidido a tratar de acelerar el cómputo mediante hardware reconfigurable o FPGAs[2][3]. Estos son esfuerzos relativamente recientes, y a lo largo de este trabajo voy a tratar de presentar y explicar el por qué de estos esfuerzos, así como en qué consisten. También voy a presentar un acelerador desarrollado en gran medida por mí mismo, su configuración paso a paso y las partes de software que se necesitan para la comunicación con un programa corriendo en un PC.

El trabajo se estructura de la siguiente manera. Tras esta introducción se hace una presentación del estado actual de la aceleración de código, tanto en tipos de aceleradores como en las distintas soluciones existentes en el mercado. Después presento mi acelerador, comparándolo con otros esfuerzos similares. Pasaremos a una fase en la que se habla del proceso de construcción, desde la investigación necesaria hasta un ejemplo funcional con resultados reales. Por último se extraen conclusiones de ese trabajo.

Introduction

We've all been there, where we are using our PC and we think "why is it running so slow?" It might be because we've used a more up-to-date one, because it needs some serious fan cleanup or because we have way too many tabs open in our web browser. While the appreciation might stem from any of those cases, I'd like to focus on the first one, which might be the most common one.

In more advanced fields of computation, "a more up-to-date computer" was the general acceleration approach: you wated for a new chip with a faster clock because it would come around in a few months anyways. This was a viable strategy for a very long time, until power concerns made it no longer viable. After all, power draw is proportional to clock frequency, so a day came where we didn't have access to sufficiently powerful cooling systems to deal with the higher clock frequencies.

At this point in time, code acceleration started heavily relying on parallelism, running several lines of code at a time so we could achieve greater performance with lower clock frequencies. First the multi-threaded CPU and later the GPGPU encouraged the writing of parallel or at least parallel-oriented code in order to exploit these systems.

Over time we realized that these solutions were still very power-hungry, specially in HPC situations or large servers. That's why a few brave companies have tried to accelerate code execution via hardware, generally reconfigurable FPGA fabrics. These are recent efforts, and over the course of this project I will try to present the reason behind these efforts, as well as what they are all about. I will also present my own FPGA accelerator, its step-by-step configuration and the software needed to run on a PC host.

This memory is structured as follows: After this introduction we make a brief presentation of the state of the art, both in types of code accelerators and in different market solutions, as well as making the case for FPGA accelerators. We then present our own effort, as opposed to other similar solutions. The following section will talk about the building process, from the first approaches to the running accelerator. Finally, I share my final thoughts on the subject.

Situación actual

En este apartado se hablará de la situación actual en el mundo de la aceleración de código, centrándonos en los aceleradores por FPGA pero sin olvidarnos de mencionar y explicar qué otros tipos de aceleradores existen. Esto nos servirá también para introducir el caso a favor de los aceleradores FPGA. También se expondrán una serie de aceleradores FPGA existentes, presentando su funcionamiento y las barreras para utilizarlo a nivel didáctico, y otros trabajos similares de aceleradores domésticos por FPGA.

Tipos de aceleradores

Existen aceleradores de código de muchos tipos, aunque los más extendidos parecen ser los basados en GPGPU por su facilidad de uso. Hoy en día podemos hablar de tres tipos:

Basados en coprocesadores / GPGPU

Los más extendidos por su simplicidad[4], debido a que sólo requieren del acercamiento estándar al desarrollo y un compilador especial. Su desventaja es que, aunque son ligeramente más efectivos que una CPU sin más, siguen requiriendo bastante energía[5]. En resumen, consiste en emplear una GPU (tanto profesionales como las nVidia Quadro como una AMD RX 460 normal y corriente) o un coprocesador de características similares (como Xeon Phi[6]) para realizar tareas de cálculo abstracto en lugar de procesamiento gráfico. Existen muchas iniciativas para utilizarse, e incluso nVidia desarrolló CUDA como lenguaje para explotar las capacidades de cómputo de sus GPU.

Basados en ASIC

Los ASIC o Application Specific Integrated Circuit son circuitos integrados de propósito único, y como tales no muy extendidos al ser rígidos en su naturaleza. Aun así son la forma más efectiva y eficiente de conseguir velocidad de cómputo, por lo que en su momento se intentaron utilizar en proyectos[7] y hoy en día siguen teniendo cierto predicamento para el minado de criptomonedas.

Basados en FPGA

Las FPGA o Field Programmable Gate Array son el paso intermedio entre la especificidad de los ASIC y la programabilidad de las CPU, y por ello están abriéndose paso entre el mercado de los aceleradores de código. En concepto se trata de un campo de puertas lógicas que se pueden interconectar de la manera que nos resulte más interesante, consiguiendo un circuito integrado de

cómputo específico[8]. Son menos efectivos que los ASIC pero siguen siendo tremendamente eficientes (hasta 100 veces más que una CPU[5]), y al cortarle la corriente se pueden reconfigurar para otro circuito. Tienen como gran desventaja que, salvo en muy contadas excepciones[11], requieren de alguien que se encargue de desarrollar el circuito que irá en el interior de la FPGA.

El caso a favor de los aceleradores FPGA

Una vez presentados los tres tipos de acelerador de código existentes, es lógico que uno se pregunte cuáles son las ventajas de cada uno, en especial el acelerador FPGA. Al fin y al cabo, se puede pensar que la mayoría de sistemas de síntesis hardware mediante lenguajes de alto nivel (o HLS) están más preparados para producir ASICs[7], que además son más eficientes. O que no merece la pena la inversión necesaria para crear un acelerador por FPGA cuando un acelerador por GPGPU tiene un precio similar y es más fácil de implementar, aparte de no necesitar personal específico.

La ventaja principal del acelerador FPGA radica en que se tiene lo mejor de ambos mundos. Es quizá algo menos eficiente que un ASIC[5], pero la diferencia la suple con creces al ser reconfigurable[8]. De hecho, hubo intentos de desarrollo de aceleradores via ASIC que fueron descartados[7] por su rigidez - habría que “tirarlos” cada poco tiempo.

De cara a enfrentarlos con las GPGPU, la desventaja de requerir a alguien que se encargue de programar la circuitería se puede suplir con el aumento de rendimiento[5] respecto a estas soluciones y, sobre todo, a la reducción de consumo energético, tan necesario hoy en día. Mientras que es raro encontrar una GPU correctamente optimizada para cómputo que requiera menos de 200W de potencia (la TitanXP recién estrenada de nVidia requiere 250W y seguramente las Quadro de la serie Pascal necesiten más), hay FPGAs de alta gama con consumos energéticos tan reducidos como 25W[2], reduciendo así tanto la necesidad de alimentación externa (pueden tomarla del conector PCIe[9] en el caso de estar conectadas de esta manera) y el cableado asociada a ésta como de refrigeración activa y pasiva.

Por si acaso estas ventajas no son suficientes, las FPGA son más asequibles que otras soluciones de aceleración, pudiendo conseguir una FPGA de gama media con conector PCIe por unos 3000$ frente a los casi 6000$ de una tarjeta gráfica profesional, y el gasto de ingeniería asociado a un ASIC, que además se va a tener que descartar en muy poco tiempo, puede superar fácilmente el valor de mercado de las FPGA.

Si hubiera que presentar una desventaja real para el uso de FPGAs tendríamos que hablar del coste de desarrollo. Desarrollar código de descripción de hardware a mano es costoso[2] y, como hemos dicho, gran parte de las herramientas HLS están optimizadas para la creación de ASICs[7], por lo que se acaba teniendo que modificar el código de descripcion obtenido a mano. Por suerte, existen ciertas herramientas HLS más orientadas a las FPGAs[10], pero son específicas de cada fabricante de chips, aunque generalmente la compra de una placa trae consigo una licencia de uso.

En resumen, los aceleradores vía FPGA parecen ser la opción más sensata de cara al futuro, al ser más eficientes que una GPGPU tanto en rendimiento energético como en la compleja relación capacidad de cómputo por vatio y dólar y ser más flexibles que un ASIC permitiendo así su mayor longevidad dentro del servidor donde se encuentre asentada.

Plataformas propietarias de aceleración

En este sub-apartado voy a hablar de una plataforma hardware y un framework software de carácter propietario relacionados con las FPGA. La plataforma hardware presenta un acercamiento a la aceleración mediante una FPGA débilmente acoplada a la CPU en un entorno de memoria compartida. El framework software presenta una ayuda inestimable para el desarrollo pero a un precio elevado y para un conjunto reducido.

Microsoft Catapult

Microsoft es uno de los primeros usuarios de aceleradores FPGA en producción[2], junto con Baidu[3], y lo hicieron mediante la creación de su plataforma Catapult[2]. Esta plataforma se desarrolló con el objetivo de mejorar la eficiencia de los servidores que Microsoft ha puesto al servicio del proyecto OpenCompute[12] con el objetivo de aumentar el rendimiento de los servidores que distribuía anteriormente en un 100% bajo dos restricciones claras:

- El consumo extra de potencia no debía superar al de un dispositivo de prueba PCIe (10W) por servidor de ½ U de ancho.
- El TCO (Total Cost of Ownership) no podía verse incrementado en más de un 30%

El diseño fue pasando por diversas modificaciones. Originalmente se trataba de un blade extra con 6 placas Xilinx Virtex serie 5 para cada 6 blades, pero ese diseño se acabó descartando por su poca flexibilidad (si se necesitaban más de 6 FPGA para el circuito integrado resultante no se podía implementar) y dificultad de mantenimiento (era imposible resolver un problema que solo afectase a una de las FPGA sin parar el servidor por completo). Después se

plantearon distribuir la localización de los chips y pasaron a construir un módulo que se conectaría a los PCIe del blade. Ahí encontraron un problema de refrigeración que solucionaron pasándose a FPGAs para el entorno industrial[2].

El resultado final consistió en añadir estos módulos, cada uno equipado con conexiones al back-plane y una FPGA Altera Stratix serie 5 que, empleando las dos conexiones de red SAS y un poco de inventiva, conseguía una red en toro 2D de 6x8 módulos para el acelerador. De esta manera, en el hipotético caso de requerir un circuito muy grande, se dispone de 48 FPGAs para implementarlo[2].

Desplegado desde finales de 2015 en un entorno de producción como los servidores del buscador Bing[7], han conseguido incrementar la eficiencia de su carga de trabajo (generación de resultados) en cerca de un 90% en throughput y un 30% en velocidad, todo con un aumento del TCO de apenas un 10% y el aumento de consumo energético en unos reducidos 10W por cada servidor. Para ello han empleado un circuito con un tamaño de 7 FPGAs en pipeline replicado 6 veces en cada malla, dejando en cada pipeline una FPGA a modo de backup en caso de que alguna falle, que se encarga del algoritmo de ranking de las páginas[2].

La gran desventaja de este sistema es que sigue siendo principalmente una plataforma hardware y resulta tremendamente complicado desarrollar para ella, quizá más incluso que para una única FPGA debido a las comunicaciones.

Xilinx SDAccel

Desde hace bastante tiempo, Xilinx vio la capacidad del empleo de su hardware reconfigurable en entornos de computación de alto rendimiento, pero también eran conscientes de las enormes desventajas del uso de hardware en estos entornos e nivel de coste del desarrollo, coste de las pruebas, tiempo de desarrollo[13] y otros factores. Por ello han tratado paso a paso de reducir esos tiempos, tanto en su plataforma de desarrollo de sistemas como en un módulo de HLS optimizado para sus FPGA.

Todos estos avances acercaron poco a poco la aceleración por hardware al gran público, pero como veremos más adelante se seguían necesitando grandes cantidades de tiempo de desarrollo enfocados no a solucionar el problema que se tenía entre manos si no a solucionar otros problemas que poco tenían que ver[13].

El auge de OpenCL como lenguaje de programación en computación de alto rendimiento y su esquema de computación por kernels acoplables les llevó a

pensar en dar un paso más, y con ello crearon SDAccel, un compilador de OpenCL para FPGAs con su consiguiente suite de pruebas previas al paso al hardware[14].

El objetivo de este framework es el ya mencionado: acercar lo más posible la programación de hardware reconfigurable al estilo de programación clásico. En una situación perfecta, SDAccel convertiría el ritmo de trabajo con hardware en uno prácticamente idéntico al ritmo de trabajo con GPGPU[13][14].

Por supuesto no todo son buenas noticias. El soporte para SDAccel es reducido a día de hoy[11], e incluso en el futuro requerirá que las placas se encuentren dentro del conjunto de placas soportadas, que generalmente serán placas específicas de gama media-alta (de las dos placas soportadas actualmente una es de la gama de productos Kintex UltraScale y la otra es de la gama de productos Virtex 7) similares a los kits de evaluación y con el conector PCIe integrado.

Además el funcionamiento puede llegar a ser notablemente más complejo debido, entre otras cosas, a la necesidad de empaquetar antes de compilar (no puede cargar código de forma tan simple como una GPGPU[14]) y a que el propio compilador presenta varias fases de prueba y verificación antes del binario final[14]. Pruebas que por otra parte habrían sido necesarias de todas formas y que de esta manera se pueden hacer sin tener que abandonar el entorno de trabajo[13].

A cambio, una vez se realiza la inversión en uno de estos sistemas, se recibe la licencia de uso de SDAccel y la posibilidad de utilizar este sistema en ese proyecto, reduciendo ostensiblemente la necesidad de expertos en hardware para hacer funcionar el acelerador. Además, al ser un compilador OpenCL (con compatibilidad para C y C++), se puede utilizar este entorno de desarrollo en cualquier cluster normal sin encontrar grandes diferencias en el código legado, pudiendo portar de manera cómoda código pensado para GPGPU[13].

Otros trabajos similares

Debido a la juventud de este campo, existen muchos trabajos recientes sobre la aceleración mediante FPGAs con muchos acercamientos distintos. Existen los estudios sobre las plataformas ya mencionadas, existen librerías de código abierto para construir aceleradores como éste que hemos construido desde cero[15][16] (y que trataremos en cierta profundidad en este apartado) e incluso planteamientos más sencillos o más complejos que el de emplear la FPGA como un coprocesador específico por PCIe.

Esto se debe en parte a la enorme flexibilidad que hemos mencionado anteriormente de las FPGAs. Si se quiere realizar un trabajo de cómputo sobre semiconductor reconfigurable, existe la opción de hacerlo todo personalizado en función de las necesidades del sistema.

Como ejemplos radicales, en mi investigación encontré dos aceleradores que llamaron mi atención. Uno de ellos era acelerador a modo de tarjeta externa por USB para la renderización de pinceles[17], debido a que la tasa de datos que necesitaban no era demasiado elevada y podían permitirse cargar el conjunto de datos en la memoria DDR de la que disponía la placa de trabajo. De esta manera se ahorraron escribir un driver para la comunicación y dedicar todo el tiempo de trabajo al código y su implementación hardware. El segundo era un chip FPGA integrado en lugar de una CPU en cierto modelo de cluster Cray[24], que según sus usuarios dificultaba enormemente la programación pero el rendimiento obtenido era innegable.

Librerías de comunicación con FPGA

Una vez mencionados estos casos, pasaré a hablar de ciertas librerías enfocadas a la construcción de aceleradores tipo tarjeta PCIe (en las que baso en parte mi propio trabajo de construcción). Cada una tiene sus características y su enfoque particular: EPEE está más enfocada al control absoluto del dispositivo PCIe[15] mientras que RIFFA busca el mayor rendimiento en la tasa de transferencia de datos[16] aunque esto suponga reducir la funcionalidad PCIe del dispositivo.

EPEE

Desarrollado de manera conjunta por la Universidad de California / Los Ángeles y la Universidad de Beijing, EPEE busca ofrecer la capacidad que tienen los bloques hardware integrados en las FPGA para la interconexión PCIe sin la necesidad de hacer el trabajo a mano. La librería tiene múltiples capacidades de transferencia de datos (desde la transmisión básica hasta transmisiones

zero-copy) implementadas en el driver incluido, además de incluir una definición HW sintetizada contando con los bloques integrados de PCIe.

La desventaja frente a otras soluciones es la velocidad de transferencia (Empieza a resentirse en cuanto se sale de PCIe 2.0 y nunca es superior al 80% de la velocidad alcanzable) y la existencia de ejemplos sólo para placas de referencia Xilinx, dificultando su uso en chips instalados en placas que no sean los kits de evaluación y prácticamente imposibilitando la utilización de otro fabricante de FPGAs como Altera[19].

La ventaja frente a otras soluciones es la libertad para acceder a todo el espacio de configuración de PCIe con todas las ventajas que ello conlleva, tanto a la hora de leer y modificar la configuración en sí como el uso de ciertas capacidades (como User Defined Interrupts) que otras librerías no permiten. También permite el acceso a registros de usuario definidos en la FPGA, permitiendo un control más fino del acelerador que otras soluciones si nuestro acelerador tiene cierta complejidad. Además es libre y de código abierto, que aunque no la diferencie de la otra librería de la que hablaremos sí la diferencia de otras soluciones como Xillybus[15][20].

RIFFA

RIFFA son las siglas de “Reusable Integration Framework for FPGA Accelerators”[16], y es exactamente lo que dice el nombre - una librería reutilizable para integrar aceleradores por FPGA, entendiendo por “integrar” el hecho de conectarlos a un computador mediante PCIe y ejecutar código en él. RIFFA está desarrollado por la Universidad de California / San Diego, y se mantiene en constante evolución. En el momento de realizar este trabajo, la versión 2.2.1 ya soportaba múltiples placas Xilinx y Altera.

De hecho, esa es una de sus grandes ventajas. Pese a que sigue ciñéndose principalmente a los modelos de referencia (con sus consecuentes dificultades de adaptación si se trabaja con modelos distintos a los kits de evaluación), RIFFA tiene modelos funcionales para chips de varias gamas de Xilinx y de Altera, aumentando enormemente su usabilidad. Un avance importante si tenemos en cuenta que Altera también tiene más de un 30% de la cuota de mercado de las FPGA[19].

En cuanto a capacidades, es similar a EPEE en el sentido de que también permite transferencias de varios tipos - simple, scatter-gather, zero-copy[16]. Al igual que EPEE, tiene una parte de hardware que se sintetiza sobre el bloque duro de PCIe y que, como ya hemos dicho, está portada a otras placas[16]. De cara al driver, el funcionamiento es ligeramente más simple debido a que esta

librería no tiene soporte para UDI o registros controlados por el usuario, reduciendo ligeramente su usabilidad.

A cambio, las velocidades de transferencia alcanzadas se acercan a la saturación del enlace incluso en conexiones PCIe 3.0 x8, que hasta ahora es la conexión con mayor ancho de banda entre las placas de FPGA. Además, tienen APIs para su uso con Java, Python o Matlab además del clásico C o C++[21], permitiendo una mayor flexibilidad a la hora de programar para una plataforma empleando esta librería respecto a cualquier otra solución.

Nuestra plataforma

Pese a las enormes bondades de ambas plataformas, existen tres razones por las cuales no se utilizó ninguna de las dos de cara a la realización del acelerador, aun sabiendo que habría sido la solución más sensata.

- La placa. Nuestra placa no era ningún kit de evaluación de Xilinx o Altera, si no un módulo MMP 7K410 de Avnet[22] (equipado con una FPGA Kintex-7 410 de Xilinx) montado sobre la MMP Baseboard para disponer de conectividad PCIe 2.0 x4. Esto implicaba que, en el caso de emplear cualquiera de esos frameworks, habríamos necesitado un duro trabajo de adaptación del código, tanto al modelo de placa como al módulo como al empaquetado de pines de la FPGA utilizada.
- Trabajo que se habría podido hacer si se hubiera dispuesto de tiempo, pero ese no fue el caso. Más adelante se especificará qué problemas de tiempo reales se tenían, pero de momento basta con saber que había que tener el acelerador listo y funcionando en una fecha concreta y que no permitía realizar ese trabajo de adaptación de forma completa.
- La tercera razón en parte es didáctica. Si se hubiera utilizado una de esas librerías lo único que habríamos hecho sería coger un trabajo ajeno y hacerlo funcionar. Un trabajo interesante en sí mismo, pero objetivamente no se aprende lo mismo que desarrollando tu propio sistema y un driver funcional para él. De esta manera, construyendo una nueva plataforma desde cero, no sólo aprendemos como funcionan esas librerías a modo de trabajo de investigación sino que aprendemos a hacer la nuestra con herramientas que necesitan menor especialización a todos los niveles.

Eso no significa que este sistema sea completamente inútil. Se ha construido utilizando solo IPs proporcionadas por el programa Xilinx Vivado[23] (salvo el código de ejecución, que ha requerido un trabajo de diseño externo y que lo necesitará en el caso de que otra persona quiera utilizar este trabajo), lo que

significa que es compatible con cualquier placa de Xilinx solo con cambiar una línea del driver. Puede servir como herramienta didáctica, el driver es de código abierto y el rendimiento es aceptable. Y el driver por defecto admite una cantidad importante de comandos, permitiendo configurar varias partes del sistema desde el programa base.

Menciono que nuestra placa no es un Evaluation Kit, y es importante por diversas razones. La más fácil de presentar es que no es directamente compatible con ciertos frameworks, pero no es la única. El uso de la Baseboard con el módulo 7K410 tiene ventajas sobre ciertos EKs - es más asequible, más robusta y tiene su propia fuente de alimentación[22], además de ser el chip modelo 7K410 en lugar del 7K325, con menor cantidad de lógica programable.

Pero no todo son ventajas para el conjunto MMP más Baseboard. Este hardware requiere cierto ensamblaje[24], conectando cada parte en su sitio con suma precisión para no inutilizar una placa que, pese a ser más asequible que un Evaluation Kit, sigue sin ser precisamente barata. Necesita cierta alimentación externa, reduciendo las ventajas de eficiencia. Esta alimentación es muy reducida por lo que no es el mayor de los escollos, pero no llega a los niveles de eficiencia de los que hablábamos que alcanzaba Catapult.

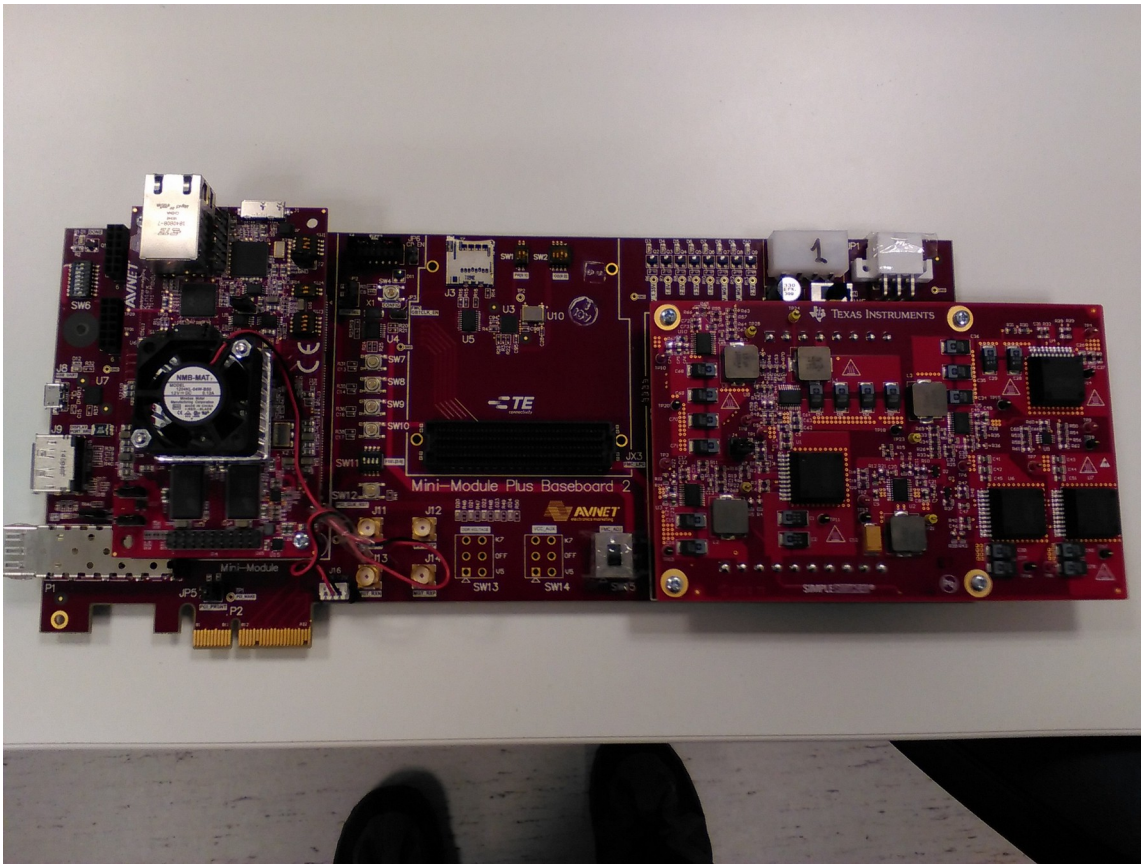


Imagen 1: Placa desconectada. Se observan las 3 partes (Baseboard, Módulo con FPGA, fuente de alimentación)

Y luego está el mayor de los escollos de esta placa: el tamaño. Una vez ensamblada tiene unas dimensiones similares a las de una tarjeta gráfica para el público entusiasta, que por si no queda claro es un tamaño más que considerable. Para asentarlo en el banco de trabajo tuvimos que hacer un trabajo importante de reubicación de muchas piezas incluyendo uno de los ventiladores, y se ha de plantear seriamente qué tipo de acelerador se quiere hacer a la hora de emplearlo porque no va a caber en un rack fácilmente.

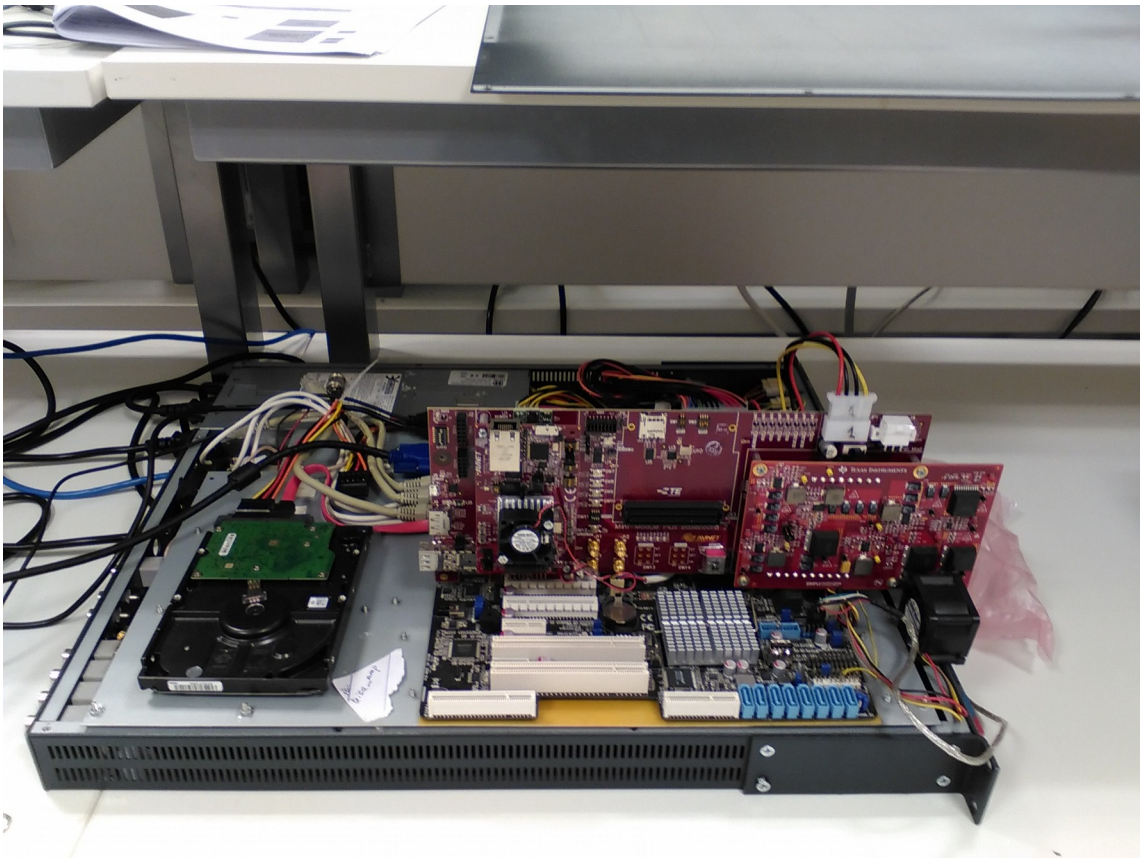


Imagen 2: Placa conectada al banco de trabajo. Se observa el trabajo de relocalización de elementos

El proceso de construcción

Ésta es la parte más importante del trabajo, donde se hablará de todo el proceso de planificación, construcción y puesta en marcha de nuestro acelerador, se presentarán los resultados y finalmente llegaremos a conclusiones. A lo largo de esta sección se discutirán las diversas decisiones tomadas de cara a cada paso que se dió y por qué se tomaron esas decisiones.

Antes de nada vamos a hablar de los objetivos de este trabajo. Dado que solo es una prueba de concepto realizada con una planificación algo ajustada y con fines en parte didácticos, el objetivo principal era obtener una aceleración visible. No existía un objetivo de rendimiento fijo como el que inspiró Catapult, ni una posibilidad de reutilización como objetivo principal (aunque fue algo que con el tiempo se consiguió de forma natural). Sólo se buscaba realizar un acelerador desde cero.

Lo primero que necesitábamos para esto era un caso de estudio. Este caso nos vino dado por un trabajo previo de otro alumno[31] que había conseguido aceleración al pasar código a hardware, pero lo había hecho en SoC Xilinx Zynq. Este pequeño detalle cambia enormemente el proceso por varias razones:

- El SoC Zynq[25] tiene integrado en el mismo chip el microprocesador (Un ARM Cortex) y una pequeña lógica programable (En el caso del modelo 7Z020 “Zedboard”, la lógica programable es similar a una placa Xilinx Artix de gama media[25]). Esto significa que comparten interconexiones y el acceso a memoria es compartido facilitando la transferencia de datos.
- De la misma manera que existe SDAccel para sistemas parecidos al nuestro (host CPU con una FPGA conectada en el bus PCIe), Xilinx tiene un software llamado SDSoC[26] que se encarga de hacer un trabajo similar sobre chips Zynq - esto es, tomar el código de alto nivel y convertir las partes seleccionadas en un circuito integrado para FPGA. Esto no significa que no requiriese ningún trabajo, sólo que ese estilo de trabajo no nos servía para nuestro sistema.
- Incluso aunque no se tuviese SDSoC, el sistema operativo en el Zynq tampoco es igual a un Linux normal y corriente[27]. Entre otras cosas contiene integrados drivers para gran cantidad de IPs de interconexión en la parte FPGA y comandos para emplearlos. Esto no nos serviría en

una plataforma Linux de escritorio o servidor, en parte porque el IP de interconexión formaría parte de una placa FPGA.

El objetivo era, por tanto, tomar ese código que en origen corría solo sobre CPU y que fue portado a un SoC Zynq y emplear ese conocimiento para volver a portar el código para funcionar en una plataforma híbrida CPU/FPGA, obteniendo en el proceso la aceleración que fuese posible. Veremos un poco más adelante que el resultado fue ampliamente satisfactorio, con una aceleración cercana al 1000%.

Sin más preámbulos, pasaré a presentar el caso de estudio en sí.

El caso de estudio

Lo que se busca es ver qué aceleración se puede alcanzar al pasar el cálculo del coste para comprimir un video a formato x265 de una CPU a una FPGA. Para ello tenemos:

- Un video de referencia tipo que nos servirá para ver el comportamiento del algoritmo en múltiples escenarios.
- El algoritmo que realiza el cálculo del coste escrito en un lenguaje de alto nivel (en este caso, C++)
- Una secuencia de valores de referencia obtenidos por CPU que nos servirán para saber cuándo la FPGA está devolviendo valores correctos.
- La tasa de frames por segundo que calcula la CPU como valor de rendimiento base, así como una prueba realizada sobre SoC Zynq que demuestra la viabilidad de la aceleración vía FPGA[31].

La ventaja de empezar con un trabajo anterior es que aunque no todo sea utilizable sí hay una parte que se puede utilizar. En este caso nos valimos del análisis del código realizado[31], en el que se podía ver que la mayor parte del tiempo de ejecución se ocupaba en un bucle de multiplicaciones y acumulaciones, perfecto para este tipo de aceleradores.

También utilizamos la separación de código que realizó, tomando la parte de código que había decidido acelerar. Esto lo hicimos así porque para crear el IP que haría el cómputo nos valimos del software HLS incluido en Vivado, pudiendo sintetizar directamente desde el código C tras unas modificaciones en forma de pragmas indicando la arquitectura de memoria y otras optimizaciones, aunque nos estamos adelantando un poco aquí.

Una vez ya sabemos cuál es la base sobre la que vamos a realizar este proyecto, podemos pasar a hablar del proceso en sí. Empezando por la

estrategia para poder conseguir la máxima aceleración dentro de unos parámetros establecidos.

Adaptando el algoritmo

Ya sabemos cual es el algoritmo que queremos acelerar, pero obviamente no podemos simplemente cogerlo y pasarlo por un software HLS para obtener el circuito acelerador. Necesitamos definir de qué forma va a acceder a los datos que necesita (de momento no vamos a tratar el problema de llevarlos a la FPGA), qué clase de optimizaciones vamos a efectuar sobre el algoritmo a nivel de paralelizado y cómo nos va a devolver los resultados. Además, tenemos que hacerlo de tal forma que no nos ocupe absolutamente toda la lógica, porque tendremos que conectar partes de interconexión al sistema.

El algoritmo, sin tocarlo, tiene más o menos la siguiente apariencia:

```
void cost(in int framesize, window, w, sizeCorr, h,
char* image,
int*  hist;
out int* cost)
memset(hist, 0, all)
*xcor_val = malloc(framesize * sizeof double)
for(i, j:2, 1:h) do
      //Generar buffer para cada cosa
      *even_lines = image+ i*w +window
      *odd_lines = even_lines + w
      memset(xcor_val, 0, all)
      //Correlacion cruzada por linea
      for(k=-window, r: 1, 1: window) do
            *buf = *even_lines + k
            for(n:1:sizeCorr) do
                  even2= buf[n] * buf[n]
                  odd2 = odd_lines[n]*odd_lines[n]
                  corr = buf[n] * odd_lines[n]; done
            xcor_val[r] = corr / sqrt(even2*odd2); done //fin ccl
      for(r=1:1:numCorr) do
            MaxCorr = xcor_val[r] > MaxCorr ? xcor_val[r] : MaxCorr
            MaxX = xcor_val[r] < MaxCorr ? MaxX : r;done
      hist[j] = MaxX – window; done
cost=0
for(l:1:h) do
      cost+=hist[l]*hist[l]; done
end
```

Como vemos, tiene 5 entradas de tipo entero y un puntero a un buffer de caracteres. Sabiendo el objetivo, sabemos que ese puntero es el frame a analizar, y que por tanto va a representar al frame. La codificación de la que disponemos es de 8 bits por píxel, por lo que tras hacer un cálculo rápido del tamaño del frame podemos pensar que necesitaremos unos 2MB para la imagen (1920x1080x8b ~16Mb). Salidas tiene 1, también de tipo entero.

Cuando se hace por software se llama 3 veces a este algoritmo: una vez para el frame entero y una por cada campo (lineas pares / lineas impares), siendo el coste de frame el obtenido con la llamada de frame y el coste de campo el obtenido por la suma de los costes de ambos campos. Para el HW se decidió cambiar esto por una única llamada que diera directamente ambos resultados, dado que se podría hacer que los cálculos intermedios se escribieran en dos registros, uno para cada llamada relevante, puesto que el frame no cambiaría. La llamada por HW necesitaría de esta manera tener 2 salidas, una por coste.

Lo más lógico, por tanto, es poner 5 registros de lectura y escritura para las entradas y 2 para las salidas. Dado que no va a funcionar de manera aislada, es lógico añadir un par de registros de control y estado, para poder saber cuándo se hace qué. Dichos registros también servirán para saber cuándo se dispone de los resultados (más adelante se explicará por qué no se utilizaron interrupciones) para poder presentarlos en pantalla.

Queda por tanto decidir dos cosas:

- Cómo se va a organizar el buffer. Tenemos dos posibilidades principales de colocación, esto es, conectarlo en el bloque DDR3 que tiene el módulo sobre el que va la FPGA al que tiene acceso o aprovechar que no es demasiado grande para añadirlo como cache en el IP. La primera opción tiene la ventaja de que solo necesitas cachear en IP el numero de líneas que se vayan a calcular a la vez y los resultados intermedios, pero necesita una entrada de bus particular. Cachear el frame completo en el IP simplifica el modelo de bus y permite mayor paralelismo al poder dividirlo tanto como se quiera, pero al mismo tiempo penaliza la capacidad de cómputo de la FPGA al gastar una gran cantidad de transistores en un buffer que va a ser limpiado cuando el cálculo finalice.
- La precisión de los cálculos. Cuanto mayor número de bits se necesite para cada sumador y multiplicador, más espacio ocuparán en la FPGA y más tiempo necesitarán para realizar los cálculos, pero el algoritmo CPU empleaba valores de doble precisión y esto puede resultar en errores de redondeo.

A la hora de la verdad, decidimos colocar el buffer en el IP. Se podía dividir en múltiples buffer de menor tamaño (aunque dificultase la transferencia de datos) y acceder a todos a la vez, y así poder realizar más cálculos a la vez. Aunque en primera instancia se planteó como un buffer único, pero la aceleración alcanzada no era demasiado impresionante.

En cuanto a la precisión de cálculos, originalmente se planteó como números de simple precisión para ahorrar tiempo y espacio, pero los errores de redondeo llegaban a ser tan grandes que nos planteamos si el resultado obtenido era el correcto. Esto es debido a que los resultados son el producto de millones de multiplicaciones y acumulaciones, y el error acumulado al finalizar todo podía llegar a ser de más del 20% en algunos casos. Visto eso, se optó por pasar a doble precisión que, aunque ralentizaba mínimamente el cómputo, obtenía los mismos resultados que la CPU.

La organización del buffer en el IP se hizo en 9 bloques de 120 líneas no contiguos, separados a distancias uniformes dentro del espacio de direcciones del IP. De esta manera podía calcularse sin ningún problema el coste parcial de 9 líneas al mismo tiempo, y las 1080 iteraciones se reducían a 120. Además así no era necesario hacer nada fuera de lo común a nivel de accesos a los buffers, solo se lee una línea de cada y para ello solo se necesita un bus de entrada (para cargar los datos) y uno de salida. De esta manera además se elimina la necesidad de conectar de manera específica a DDR.

De cara a las optimizaciones estándar que se pueden hacer en este tipo de desarrollos, realizamos “loop unrolling” sobre el bucle interno de multiplicación y acumulación, consiguiendo hacer todo en una cantidad de ciclos mucho menor. También se aplicó en cierto nivel sobre el bucle externo, lanzando 9 iteraciones al tiempo al acceder a 9 líneas de la imagen a la vez.

Más adelante discutiremos qué otras mejoras se podrían haber realizado y qué beneficios nos traerían, además de por qué no se realizaron.

Acoplamiento a la CPU

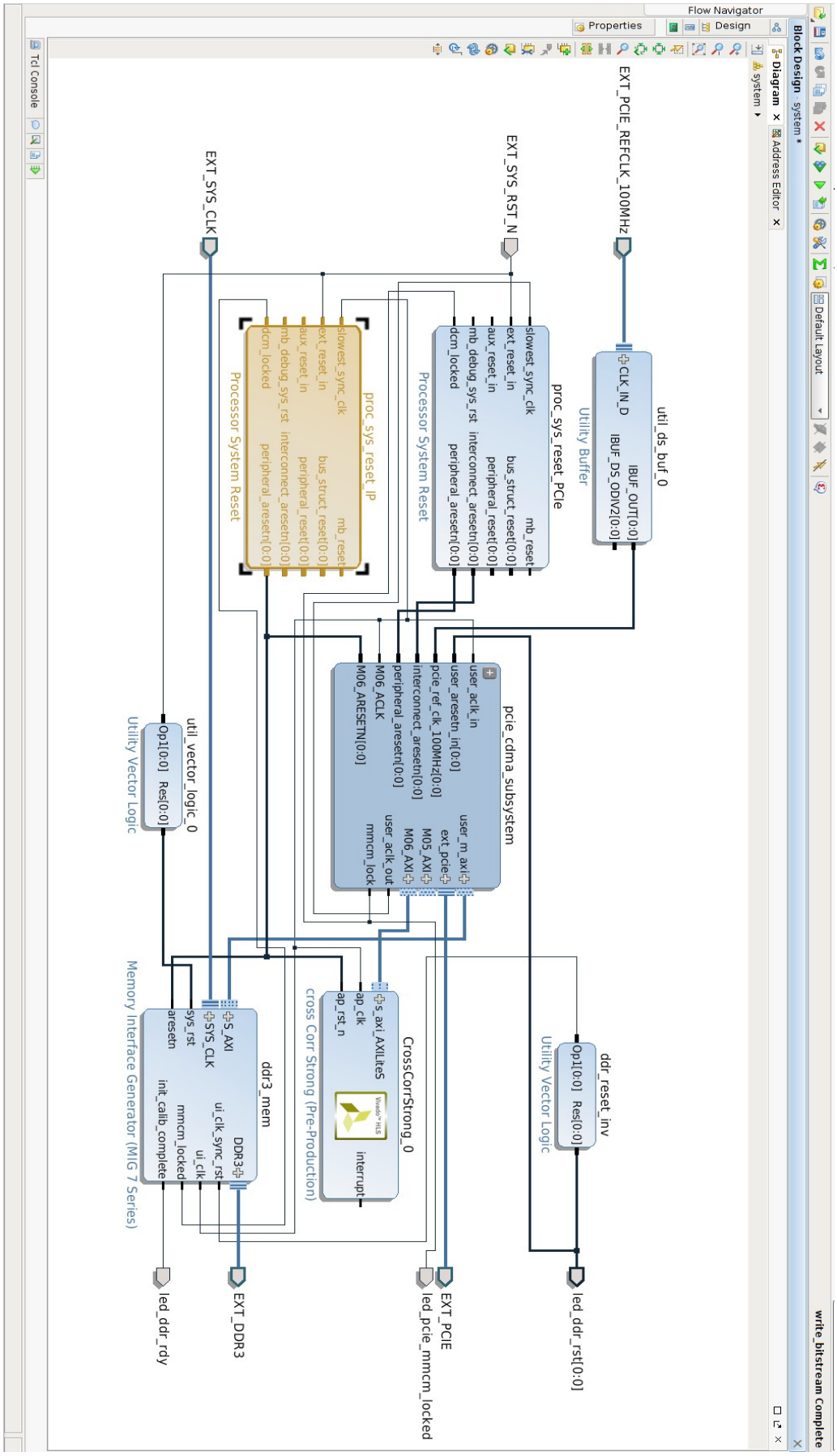
Como ya hemos dicho en repetidas ocasiones, este acelerador de código tiene formato de tarjeta PCIe. Esto significa que en la parte del hardware se requerirá de lógica que se encargue de gestionar los diversos requerimientos de PCIe[13], además de las posibles traducciones.

Para el movimiento de datos entre CPU y FPGA empleamos el bloque CDMA de Xilinx[28], modificado para trabajar con buffers de datos de 4MB en lugar de 8MB debido a ciertos problemas que surgieron en la construcción del driver. Las razones por las que empleamos este sistema de DMA en el acelerador son

su versatilidad (puede direccionar un espacio AXI completo, incluyendo conexiones a DDR en el caso de emplearse) y la extensa documentación para su programación en estilo simple y estilo scatter-gather.

Para la conexión PCIe en sí empleamos el puente AXI-Memory mapped-to-PCIe[29], también por su versatilidad, capacidad de funcionamiento y documentación del bloque, ademas de su sinergia con AXI CDMA. Se podría haber utilizado el bloque Hard-IP, pero su interacción con los distintos sistemas de DMA en FPGA es más costosa que con el puente AXI-PCIe. Además, el bloque Hard-IP maneja peor las interrupciones tipo legacy, lo que habría complicado la construcción del driver.

En la página siguiente se puede observar el diagrama final de interconexiones. Nótese que algunas líneas, como la línea de interrupciones del IP, no están conectadas. Esto se debe a que a la hora de presentar el funcionamiento de este acelerador no se había conseguido obtener el funcionamiento deseado de las MSI en el puente AXI-PCIe, así que se tomó la decisión de dejar la línea de interrupción legacy, la del DMA para controlar las transferencias, y obtener los resultados por encuesta. En una iteración posterior del sistema se pueden implementar interrupciones MSI para el DMA y el IP propio, teniendo un manejador para cada uno.



Captura 1: Diagrama de bloques del acelerador. El IP encargado del cálculo es CrossCorrStrong_0

Fases de construcción del Driver

La otra parte necesaria para el acelerador en este paradigma es un driver para un dispositivo PCIe. El driver se fue construyendo paso a paso sobre versiones reducidas del hardware hasta llegar a una versión final con una funcionalidad útil, no exentos de problemas en el proceso.

Carga del driver

La primera funcionalidad que se implementó en el driver fue la simple existencia del driver. Como se puede ver en la guía de referencia Linux Device Drivers, los dispositivos PCIe requieren un poco más de trabajo que otros dispositivos[30]. Por suerte, gran parte de todo lo que se tiene que hacer está implementado en la API de construcción de módulos en Linux.

Aun así, la secuencia de carga sigue requiriendo de cierto proceso y llamadas a funciones. Si utilizamos el estilo clásico de búsqueda, como se hizo en este driver por ser el estilo en el que estaban basados ciertos drivers de muestra de Xilinx, tenemos que realizar una llamada a la función `struct pci_dev *pci_get_device(unsigned int vendor, unsigned int device, struct pci_dev *from)` para obtener la estructura que represente al dispositivo, donde *vendor* es el ID del fabricante (el de Xilinx es 10EE) y *device* es el ID del dispositivo que puede ser modificado en el puente AXI-PCIe. **from* representa al puntero del primer dispositivo con esos ID de fabricante y dispositivo desde el que buscar (para cuando se quieren tomar varios dispositivos idénticos), pero en nuestro caso, al solo existir un dispositivo, se ponía NULL. Emplear el estilo clásico requiere “soltar” el dispositivo posteriormente con la función `pci_dev_put(struct pci_dev *dev)`, a la que se llama en la función para descargar el módulo[30].

Lo siguiente que se debe hacer (esto ya en estilo clásico o actual) es dar acceso al dispositivo con la llamada a `int pci_enable_device(struct pci_dev *dev)`, que da acceso a las diversas regiones de I/O e interrupciones del dispositivo. Esto es importante para los siguientes pasos que hay que seguir en la función de arranque, como mapear las regiones I/O en memoria (las que se necesiten) o asignar una subrutina de interrupción a la(s) línea(s) de interrupciones que tome el dispositivo.

Nos queda finalizar el empaquetado de los bloques de llamadas de carga y descarga del driver, que deberemos posteriormente definir como *module_init* y *module_exit* respectivamente. Dado que estamos fabricando el driver al estilo clásico y una llamada a lspci describirá la placa como un dispositivo de carácter, introducimos las llamadas `register_chrdev(major_number, *name, *file_ops)` en *init* y `unregister_chrdev(major_number, *name)` en *exit*.

Estas funciones se encargarán de mantener el estado del driver (en tanto que registrado o no registrado) en el kernel. En ambos casos, *major_number* se refiere al número mayor del driver, una suerte de identificador que puede o bien pedirse al kernel o fijar en un número elevado como hicimos nosotros (en nuestro caso, 241. No es una práctica recomendada, pero para tener un driver funcional se puede hacer). **name* se refiere al nombre del driver, que está dado por el nombre del nodo que representa al dispositivo que controla (en nuestro caso, pcie_10). **file_ops* es una estructura que contiene las funciones de las operaciones que se pueden realizar, y se definirá algo más adelante.

Con esto ya tenemos el mínimo absoluto para un driver PCIe. Empaquetando las llamadas dentro de sus correspondientes module_init y module_exit ya podríamos lanzar el make y cargarlo, aunque no tendría ninguna funcionalidad.

Lectura y escritura mediante MMIO

Después de tener ya un driver que arranca pasamos a darle la primera funcionalidad básica, que sería la lectura y escritura en mmio. Para ello, es necesario añadir al módulo de arranque (donde se activa el dispositivo) las siguientes dos llamadas:

```
unsigned long pci_resource_start(struct pci_dev *dev, int bar);
unsigned long pci_resource_len(struct pci_dev *dev, int bar);
```

Cada una en una variable, representan espacios de IO (el que esté en el numero BAR, de 0 a 5 ambos incluidos). Además es necesario realizar un `struct resource *request_mem_region(unsigned long start, unsigned long len, char *name)` para pedir un espacio de memoria sobre el que mapear el espacio IO, donde start y len representan el inicio y su longitud (obtenidos cada uno en las respectivas llamadas anteriores) y name es una cadena de caracteres cualquiera.

En caso de que ninguna de estas llamadas falle, se puede realizar la llamada a `void *ioremap(unsigned long phys_addr, unsigned long size)`, donde *phys_addr* representa la dirección física del inicio de la región a mapear y *size* el tamaño a mapear. Al mismo tiempo se debe añadir al módulo de salida, antes de la llamada a *unregister_chrdev*, la llamada a *iounmap* con el puntero obtenido con *ioremap* y la llamada *release_mem_region* con los mismos parámetros de inicio y longitud empleados para la petición. Hecho esto ya tenemos acceso a esas regiones, pero para poder utilizarlas necesitamos definir una serie de operaciones. Para nuestro driver solo definiré las operaciones más básicas: *open / close* para obtener acceso, *ioctl* para mandar comandos y *write / read* para acceder a los datos. Estas operaciones se juntarán en una estructura de operaciones de driver para el registro, como hemos visto hace poco. La estructura tiene el siguiente aspecto:

```
struct file_operations pci_fops = {
    read:           _read,
    write:          _write,
    unlocked_ioctl: _unlocked_ioctl,
    open:           _open,
    release:        _close,
}
```

Las funciones de *open / close* realmente no hacen nada más que notificar que han sido utilizadas, puesto que lo más importante se hace en la cadena de llamadas y de-referencias que llevan a la función definida en el driver. En ioctl en primera instancia solo se incluyen los comandos de leer y escribir registros de configuración que llaman a las funciones pertinentes de la APÎ pci de linux.

Las funciones *write / read* son las que más van a cambiar. De momento van a ser envoltorios para las funciones auxiliares *wr_reg / rd_reg* que en un futuro implementarán el comando de “leer registro de IP”.

Una vez programadas esas funciones y comprobar que funcionaban, pasamos a darle la capacidad de manejar interrupciones.

Soporte para interrupciones

El soporte para interrupciones se hizo sólo pensando en las interrupciones de tipo legacy, aunque el soporte en el driver para interrupciones MSI no es mucho más complejo y se detallarán las diferencias en los pasos.

En el software el soporte requiere tres fases: preguntarle al dispositivo cuál es su línea de interrupción, asignar a esa línea un manejador y liberar la línea cuando se descargue el driver. Como habréis supuesto por el apartado anterior, eso requiere de añadir llamadas en los módulos de entrada y salida del driver.

Por suerte, al obtener el dispositivo pci (en *pci_get_device*) tenemos la línea de interrupción en el struct que lo representa, basta con de-referenciar esa variable y asignarla a una variable, como `irq_num = device->irq`. Asignarle un manejador se hace con la llamada `int request_irq(unsigned int irq, irqreturn_t (*handler)( ), unsigned long flags, const char *dev_name, void *dev_id)`, siendo *irq* la línea a la que se asigna, **handler* la función que maneja la interrupción y *flags* una máscara que define varias opciones sobre la interrupción, como si es compartida o no. Los otros dos parámetros son identificadores para el dispositivo, teniendo en cuenta que **dev_name* ha de ser igual al **driver_name* anterior puesto que es el dispositivo el que da nombre al driver.

La diferencia en el caso de las interrupciones MSI es mínima. El número que está en el dispositivo es la primera, que se actualiza con una llamada a *pci_enable_msi* o *pci_enable_msi_block*, recibiendo esta segunda el número de interrupciones que se quieren habilitar además del dispositivo (la primera solo habilita una). Una vez hecha esta llamada, simplemente se asignan tantos manejadores como interrupciones tengamos, y las líneas de interrupciones variarán entre `device->irq` y `(device->irq)+(count-1)`.

Obviamente, **handler* debe apuntar a la función que maneja la interrupción. El manejador se tiene que encargar únicamente de limpiar la interrupción y lanzar cualquier evento que deba ser disparado por ésta en el host. Como esto es una acción que cambia según el hardware que haya debajo, no vamos a explicar mucho más del manejador. Sí voy a explicar que el retorno (*irqreturn_t*) es un envoltorio para un entero, tal que 0 significa "sin acción", 1 significa "manejada" y 2 significa "deferida". Los demás valores son ilegales.

En el módulo de salida se tiene que añadir la llamada `void free_irq(unsigned int irq, void *dev_id)` para borrar el manejador de este driver del registro, lo que acelerará el manejo de líneas compartidas. Si se utiliza MSI, además de realizar esa llamada para cada interrupción, también se tendrá que llamar a `void pci_disable_msi(struct pci_dev *dev)` para desactivar la funcionalidad MSI, siempre después de descargar las interrupciones que funcionaban de esta manera.

Una vez vimos que el manejador funcionaba, pasamos a controlar el DMA de la FPGA y añadir la entrada/salida bloqueante vía DMA por bloques. El estilo scatter/gather no se implementó por ciertos problemas con el bloque CDMA que explicaremos en el siguiente sub-apartado.

Lectura y escritura mediante DMA

Para esta parte ya tuvimos que acoplar el sistema CDMA, que nos ocupaba el recurso BAR0. Por tanto, antes de hacer nada, tuvimos que añadir un recurso BAR1 para el IP que se conectaría a todo el sistema. Al añadir el recurso BAR1 se ha de proceder igual que con la llamada que se describió en la sección de lectura y escritura por mmio - en aquél momento solo se suponía el BAR0 pero ahora se tiene que hacer para BAR0 y BAR1. No es más que repetir la misma secuencia.

También añadimos varios comandos a ioctl: lectura / escritura de registros del IP, lectura / escritura de registros CDMA y modificar dirección de la transferencia DMA. Creamos los dos buffer pertinentes (uno para la lectura y otro para la escritura) y las colas para I/O bloqueante (una para lectura y otra para escritura).

Una vez hecho esto pasamos a programar las nuevas versiones de write / read. Para ello empezamos trabajando con los buffer. Para poder utilizar un buffer con DMA ha de ser físicamente contiguo o emplear la técnica de scatter/gather. Optamos por realizar la transferencia por bloque contiguo por limitaciones del bloque CDMA e incompatibilidades con el kernel linux.

En lo referente a estas incompatibilidades, en los kernel 3.5 y posteriores se añadió a la API de DMA código para reservar “grandes bloques de memoria físicamente contiguos” (de hecho ese es el nombre del parche, “contiguous memory allocation”)[32], concretamente más de 4MB (lo máximo que, sin este código, se puede reservar mediante *kmalloc* o *dma_alloc_coherent*). Este código funciona sin problemas en kernel de 32 bit sin extensiones pae[1], pero en otro tipo de kernel está limitado por la tabla de bounce buffers que mapea las direcciones de bus de 32 bit en el espacio de direcciones de sistema[2]. Curiosamente, se rompe justo en el momento que se tratan de reservar más de 4MB físicamente contiguos. La implementación estándar de CDMA requiere que o bien los buffer de sistema de entrada y salida sobre los que actúa el DMA sean de 8MB cada uno o que el buffer con los descriptores para DMA scatter-gather sea de 8MB[28], y en cualquier caso deben ser contiguos. Debido a la cantidad de espacio que se desperdiciaba con los descriptores por un lado, la complejidad de programar por scatter-gather por otro y el hecho de que era más fácil modificar el requerimiento de los buffer de sistema que el del buffer de descriptores se optó por modificar el bloque CDMA para aceptar buffers de transmisión de 4MB.

Una vez hecho eso, se modificaron las definiciones de los buffer para ser de 4MB y se pasó a modificar el código de write / read, que ahora sigue la siguiente secuencia de acciones:

1. Si se van a enviar datos a la FPGA, copiar al buffer los datos que se quieren enviar desde el espacio de usuario.
2. Mapear el buffer de lectura o escritura según sea el tipo de transmisión.
3. Comprobar que CDMA está libre
4. Programar CDMA dándole las direcciones de origen y destino de la transmisión, así como el vector de traducción para el puerto PCIe, si queremos activar interrupciones y el número de bytes a transferir.

1 Las extensiones pae son módulos del kernel que permiten instalar más de 4GB de memoria RAM en un sistema de 32b

2 Lanzando un error “swiotlb: buffer full”. Muchas otras aplicaciones de DMA se han encontrado con este mismo problema como se puede ver en una búsqueda rápida en google.

        1. Una vez escrito el número de bytes a transferir, CDMA empieza a trabajar.
    5. Esperar la interrupción. Para ello ponemos el proceso en la cola adecuada.
    6. Una vez llega la interrupción el manejador despierta a la cola.
    7. Comprobar que no hay errores. Si es así, devolver el control del buffer mapeado a la CPU.
        1. Si además estamos leyendo de FPGA, copiar los datos que han llegado al buffer de lectura al buffer del espacio de usuario.
    8. Finalizar la transmisión

No se llegó a implementar el uso de semáforos que impedirían condiciones de carrera por causas ajenas que provocaron un retraso en la implementación de las transmisiones por DMA, además del problema de los bounce buffer que se tardó una semana en diagnosticar y otra en solucionar. El semáforo se comprobaría / tomaría antes de comprobar que CDMA está libre, y se soltaría al devolver el control del buffer mapeado a la CPU.

Una vez hemos conseguido este nivel de funcionalidad, el driver ya está listo para trabajar con nuestro programa. Una vez hemos unido todas las partes en el hardware y construimos el driver, podemos pasar a realizar las pruebas.

Pruebas

En este apartado vamos a hablar de las pruebas a dos niveles: tanto el rendimiento del propio sistema Puente AXI2PCIe más CDMA para las transferencias vía PCIe como de la aceleración conseguida en la ejecución del programa.

Para probar el rendimiento de las transferencias realizamos un pequeño programa que simplemente hacía envíos y recepciones de bloques de datos numéricos generados aleatoriamente. Era una modificación con relojes del programa que empleamos para probar que los envíos y recepciones eran correctos.

El programa probaba dos casos concretos: Uno era el envío de 4KB y otro el envío de 4MB, para poder mostrar el contraste en el overhead que cada transmisión podía tener. El tiempo que se mide es la llamada a write / read, para dar una idea del tiempo real de transmisión (esto es, el tiempo que se está en el contexto de transmisión de datos).

Los resultados muestran velocidades dispares en lectura y escritura, siendo la transmisión hacia la FPGA más rápida que la transmisión desde la FPGA y las transmisiones de grandes bloques más rápidas que las transmisiones de bloques más pequeños, siendo esta última observación un resultado esperado debido al overhead de programación del dispositivo DMA.

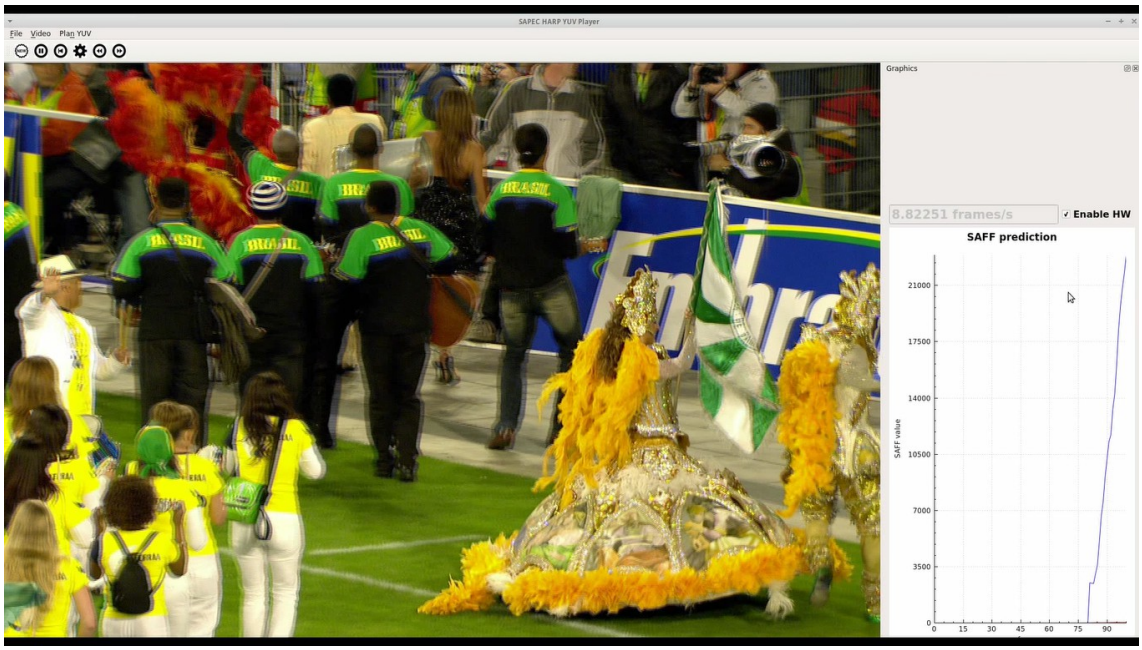
También se comparó con transmisiones de los mismos bloques de datos utilizando mmio (para ello creamos un buffer de tamaño considerable y le dimos una región de memoria en la FPGA) para comparar la eficiencia de las transmisiones, y vimos que las lecturas podían llegar a ser 400 veces más rápidas por DMA en el caso de grandes escrituras. En la tabla que mostramos a continuación se pueden ver los datos:

Tipo de transferencia		**Lectura**	**Escritura**
CDMA	4 kB	2Gbps	1.15Gbps
	4 MB	10.67Gbps	4Gbps
MMIO	4 kB	~24Mbps	~520Mbps
	4 MB	~24Mbps	~458Mbps

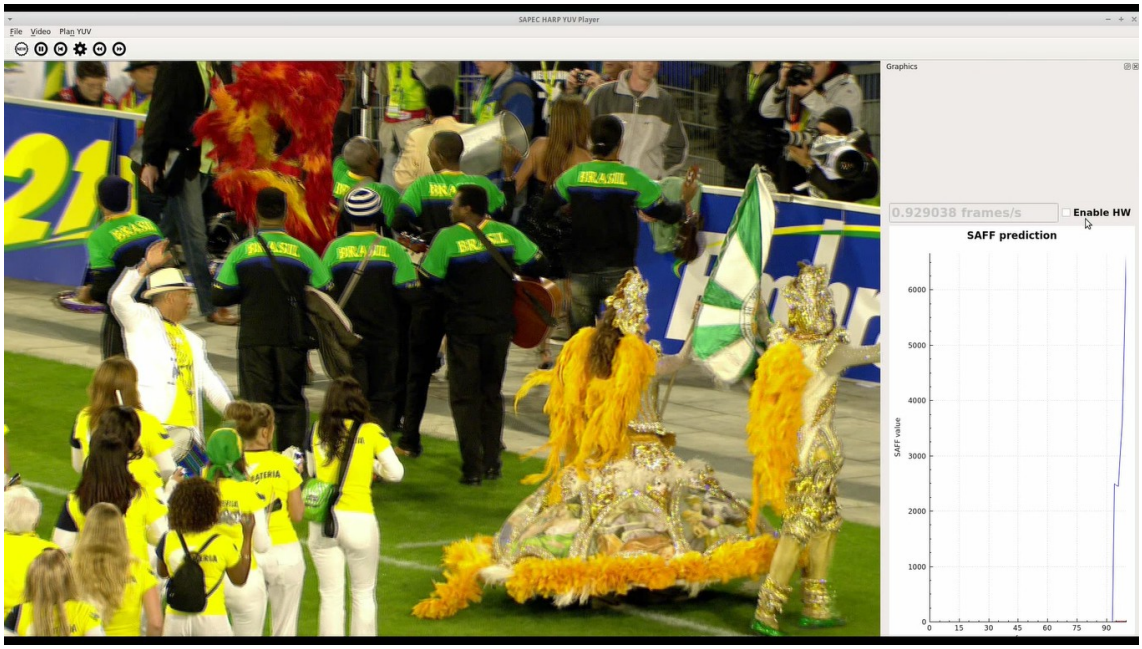
Tabla 1: Diferencia entre velocidades de transmisión según la tecnología empleada y tamaño del bloque.

En cuanto al rendimiento del conjunto en sí, la prueba consistió en cambiar la llamada a la función que realizaba el cálculo en cada frame por una petición a la FPGA para que realizase ese mismo cálculo. Una vez la FPGA estuvo lista, se pudo alcanzar un speed-up cercano al 1000% comparado con la CPU, siendo la tasa de cálculo en CPU de ~0.9 frames/s y la tasa alcanzada con ayuda del acelerador de aproximadamente 8.5 frames/s, empleando el sistema que se había construido desde cero con las optimizaciones de acceso a memoria. Un resultado satisfactorio y que superaba con creces las expectativas que se tenían tras el trabajo original sobre Zynq (speedup aproximado de 500%). En la página siguiente se muestran imágenes del programa final, en el que se podía activar y desactivar la aceleración a voluntad para ver el cambio en el rendimiento.

Vistos los resultados es lógico pensar que a investigación ha sido un éxito, pero comparando con otros trabajos siempre se puede pensar en qué podría haber mejorado. Por ello voy a hablar de ciertas mejoras que se podrían haber realizado si se hubiese dispuesto de más tiempo o de una mayor experiencia con estos sistemas.



Captura 2: Funcionamiento solo por software. Vemos que el rendimiento no llega a 1fps



Captura 3: Una vez se activa la aceleración por FPGA llegamos a alcanzar los 9fps (8.8 en esta captura)

Posibles mejoras

Ya existe una pequeña mejora en la propia llamada a la FPGA. Por comodidad, la llamada se hizo de tal forma que para cada frame se escribían los parámetros del vídeo que estábamos analizando. Esas cinco llamadas a escribir registro son un tiempo ínfimo, pero sigue siendo una redundancia innecesaria y a lo largo de un vídeo se puede acumular. Se podría dejar aparte, en una función que cargara los parámetros una vez cargado el vídeo.

De forma similar puede pensarse en la programación del DMA, al menos los vectores de traducción. Aunque en nuestro caso esto era más complicado por trabajar con un buffer cortado en 9 partes en la FPGA pero contiguo en el sistema, sigue siendo una posibilidad en sistemas cuyos buffers no tengan estas particularidades. En una vena similar, podríamos haber explotado la DDR de la FPGA y utilizar un sistema de acceso a memoria distinto, dentro del hardware.

Otra mejora, aunque sería más en el tema de la transmisión que en nuestro caso particular (dado que nuestras transmisiones eran menores a 4MB) habría sido aumentar el tamaño de los buffer. Aun mejor si viniera acompañado de alguna implementación de memoria de más alto nivel, como un scatter-gather (si se hubiera encontrado una solución distinta al problema de los bounce buffers), una implementación mmap (le daría más versatilidad al driver), o incluso ambas soluciones para conseguir a la vez facilidad de uso (con mmap) y rendimiento en las transmisiones (con scatter-gather).

El IP de cálculo tenía soporte para interrupciones, pero no las utilizaba realmente porque no disponíamos de una línea que darle al IP de cálculo. Una implementación en hardware correcta del soporte para MSI habría permitido disponer de una segunda línea de interrupción para dar soporte a las interrupciones del IP, ahorrandonos así un sistema de encuesta que indudablemente reduce el rendimiento del sistema. También habría requerido otro manejador que funcionaría de forma similar al manejador de interrupciones del DMA - limpiarla y despertar a un proceso (en este caso el de usuario), que sería el que recogería resultados.

Entre todas estas mejoras podríamos haber conseguido un rendimiento ligeramente superior en nuestro sistema, además de una mejora en la robustez, por lo que sería una alternativa aún más interesante a las librerías mencionadas hace unas cuantas páginas. Visto eso, veamos qué conclusiones se pueden extraer de este trabajo.

Conclusiones

Este trabajo se originó en una oferta de beca para investigar Microsoft Catapult en particular y la viabilidad futura de las FPGA en centros de datos en general. Pese a que el trabajo final poco o nada tiene que ver con este objetivo, los resultados no engañan: Si sabes qué partes de tu lógica de negocio pasar a un sistema hardware, puedes conseguir grandes beneficios tanto en rendimiento como en eficiencia energética, que es lo importante en los tiempos que corren en los que se necesitan resultados tan rápidos o más que antaño pero consumiendo la menor energía posible[5].

También me ha permitido ver más de cerca cómo funciona el propio kernel de Linux al escribir un driver funcional y utilizable que usar en él. Añado que escribir drivers en Linux está al alcance de cualquiera dispuesto a aprender y es una magnífica experiencia, y todo ello es debido a la facilidad que existe para acceder a la API de llamadas al sistema de Linux[30]. Desde aquí animo a cualquier alumno a trastear con su Linux, es muy divertido y muy gratificante cuando se hace bien. Debido a que escribí un driver para un hardware PCIe decidí investigar un poco sobre la escritura de drivers en Windows. Eso ya es menos entretenido, por decirlo amablemente.

Puede que la primera conclusión a la que llegué mientras realizaba este trabajo sea “mide dos veces y corta una”. Aunque parece de sentido común, el proyecto se “retrasó” casi dos meses debido a no tomarse en serio esta afirmación. Digo “retrasó” entre comillas porque ese tiempo no se gastó por completo. Aprendí importantes nociones de C y C++ que me permitieron seguir con el trabajo cuando se reanudó, pude investigar otros trabajos similares y llegar además a los dos curiosos casos de acelerador de los que hablo en el apartado de “trabajos similares” y leer sobre ellos. Pude resolver ciertas dudas sobre el desarrollo de drivers en Linux y encontrar el libro que me ha servido de referencia para el desarrollo. Es un poco antiguo (de hecho la siguiente edición saldrá al público el año que viene y se basa ya en un kernel más moderno) pero sigue siendo útil para crear drivers no muy complejos.

En lo referente a lo que se puede extraer del proceso completo, es cierto lo que en la presentación de Catapult dijo el representante de Microsoft, el desarrollo para FPGAs no es sencillo y el empujón que realmente necesita para entrar en el mercado de los aceleradores de cómputo de forma real es una herramienta que haga el paso de código a hardware algo relativamente indoloro. Existen frameworks de integración y software HLS que ayuda, pero sigue siendo complejo. Y SDAccel a día de hoy sigue sin tener el soporte necesario como

para ser algo que emplear en entornos de producción. Aun así, se tienen las esperanzas puestas en herramientas como ésa para el futuro.

La extensa sección de posibles mejoras me hace pensar que este no es un proyecto acabado. Es una buena prueba de concepto, una demostración de que se puede hacer un acelerador hardware efectivo como un proyecto didáctico, pero necesita unas mejoras que, si bien es cierto que fuimos limitados por el tiempo, serían claves para poder utilizarse en entornos más exigentes. No sé si estaré aquí para implementarlas, pero sería maravilloso verlo terminado algún día.

Bibliografía

1. “Enhanced Intel® SpeedStep® Technology for the Intel® Pentium® M Processor” - Whitepaper de Intel, Marzo 2004
2. Andrew Putnam, Adrian Caulfield, Eric Chung, et. al - “A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services”, 2014
3. Jian Ouyang, et. Al - “SDA: Software-defined Accelerator for Large Scale DNN Systems”, 2014
4. 47th Top500 list, Junio 2016
5. Qing Zhang - “Heterogeneous Computing In HPC and Deep Learning”, Septiembre 2015
6. Intel Xeon Phi product overview - https://software.intel.com/en-us/xeon-phi/mic
7. Andrew Putnam - “Large-Scale Reconfigurable Computing in a Microsoft Datacenter”, 2014
8. Xilinx, Inc. - http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm
9. PCI-SIG - “PCI Express Base Specification Revision 3.0”
10. Xilinx Vivado HLS product page - http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
11. Xilinx SDAccel product overview - http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html
12. “Microsoft Extends Commitment to Open Compute Project”, 2015–03-10. https://blogs.technet.microsoft.com/server-cloud/2015/03/10/microsoft-extends-commitment-to-open-compute-project/
13. Xilinx SDAccel Backgrounder - http://www.xilinx.com/support/documentation/backgrounders/sdaccel-backgrounder.pdf
14. Xilinx Inc, “Xilinx SDAccel Development Environment User Guide (UG1023)”, revision 2015.4
15. Jian Gong, Jason Cong, Tao Wang, et. al - “An Efficient and Flexible Host-FPGA PCIe Communication Library”, 2014

16. Matthew Jacobsen, Dustin Richmond, Matthew Hogains, y Ryan Kastner - “RIFFA: A Reusable Integration Framework for FPGA Accelerators”, 2015
17. Kentaro Sano, Yoshiaki Hatsuda y Satoru Yamamoto - “Multi-FPGA Accelerator for Scalable Stencil Computation with Constant Memory-Bandwidth”, 2013
18. Cray XD1 Datasheet, 2004 - http://www.carc.unm.edu/~tlthomas/buildout/Cray_XD1_Datasheet.pdf
19. Sourcetech 411: “Top FPGA Companies for 2013” - http://sourcetech411.com/2013/04/top-fpga-companies-for-2013/
20. Xillybus - http://xillybus.com
21. RIFFA Software API: http://riffa.ucsd.edu/node/[7-10]
22. Avnet Mini-Module Plus Product Brief - https://products.avnet.com/opasdata/d120001/medias/docus/7/AES-MMP-7K410T-G-product-brief.pdf
23. Xilinx Vivado product page - http://www.xilinx.com/products/design-tools/vivado.html
24. Avnet Mini-Module Plus assembly guide
25. Zynq-7000 All-Programmable SoC Overview - http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf
26. Xilinx SDSoc product page - http://www.xilinx.com/products/design-tools/software-zone/sdsoc.html
27. Repositorio del Kernel Linux modificado de Xilinx para aplicaciones sobre Zynq - https://github.com/Xilinx/linux-xlnx
28. Xilinx Product Guide PG034: AXI Central Direct Memory Access - http://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v4_1/pg034-axi-cdma.pdf
29. Xilinx Product Guid PG055: AXI Memory Mapped to PCI Express - http://www.xilinx.com/support/documentation/ip_documentation/axi_pcie/v2_8/pg055-axi-bridge-pcie.pdf
30. Jonathan Corbet, Greg Kroah-Hartman y Alessandro Rubini - “Linux Device Drivers”, tercera edición, 2005, O'Reilly Media

31. Álvaro Díaz Suárez, Alejandro Nicolás, Iñigo Ugarte Olano y Pablo Pedro Sánchez Espeso - “Designing embedded HW/SW systems with OpenMP”, septiembre 2016

32. Michal Nazarewicz, LWN.net - “A Deep Dive into CMA”, Marzo 2012 https://lwn.net/Articles/486301/