Estudio comparativo de simulaciones físicas en GPU y FPGA

Autor
Sánchez, Juan Carlos
Director
Ing. Vodanovic, Gonzalo

Universidad Nacional de Córdoba

Facultad de Matemática, Astronomia y Física
Marzo 2018

Agradecimientos

A mi tutor, Ing. Gonzalo Vodanovic,
por guiarme y estar presente en todo momento.
A Walter Zaninetti por su colaboración.
A mi FAMILIA que ayudo mucho a este logro.
Y amigos, que estuvieron presentes.

Resumen

El problema que se propone resolver es la ecuación de calor de Laplace, un problema paralelizable. Este problema cuenta con operaciones entre números reales que le dan a sus algoritmos un tinte interesante para su estudio comparativo. El trabajo consiste en analizar el rendimiento operacional de punto fijo en las siguientes arquitecturas:

- Implementación del desarrollo en OpenCL para medir su rendimiento en las GPU´s.
- Implementar el desarrollo en VHDL para medir su rendimiento en la FPGA.
- Implementar el desarrollo mixto, híbrido, mediante OpenCL en FPGA (Intel).

Abstact
The problem he proposes to solve is the Laplace heat equation, a parallel problem. This problem has operations between real numbers that give their algorithms an interesting tint for their comparative study. The work consists of analyzing the operational performance of fixed point in the following architectures:

- Implementation of OpenCL development to measure its performance in GPUs0
- Implement the development in VHDL to measure its performance in the FPGA.
- Implement mixed, hybrid development, using OpenCL in FPGA (Intel).

CLASIFICACIÓN: CCS → Computer systems organization → Architectures → Other architectures

PALABRAS CLAVES: GPU - FPGA - OpenCL - Paralelismo - Ecuacion de calor - Laplace

Índice general

Índice de figuras

Índice de cuadros

Objetivos

Los avances tecnológicos en el campo de inteligencia artificial, big data, robótica, entre otros, requieren gran capacidad de procesamiento. Las unidades de procesamiento gráfico (GPU) están cubriendo este campo, sin embargo, una alternativa planteada recientemente, ha sido la introducción de las FPGA. Sintetizando hardware a la medida del problema, incluso, a partir de la codificación de algoritmos en alto nivel.

En principio las GPU surgieron para cubrir una necesidad gráfica en los computadores, y luego, se vio orientada no solo a dichas prestaciones, sino a diferentes soluciones a problemas con altos procesamientos matemáticos. Como consecuencia actualmente las GPU no solo se utilizan a nivel gráfico, sino que nos ayudan a paralelizar algoritmos que requieren una gran cantidad de cálculos.

Por otro lado, surgió la idea de configurar hardware a medida para crear circuitos digitales de todo tipo, estos dispositivos se denominan FPGA. Al igual que las GPU, se pudo ver que esta tecnología podría utilizarse para cálculos específicos con hardware adaptado especialmente al problema.

Recientemente el mercado se vio impulsado a la combinación de las técnicas ya propuestas con lenguajes paralelizables y las FPGA llamados sistemas híbridos, donde se convierte código en alto nivel paralelizable a un hardware específico para el problema.

El presente trabajo tiene como finalidad comparar la implementación de un problema de aplicación real en diferentes tipos de tecnologías: GPU y FPGA; ambas pueden ser utilizadas para resolver problemas altamente paralelizables haciendo uso de distintas técnicas o mecanismos planteados por cada arquitectura. Al implementar una posible solución al problema es fundamental conocer los dispositivos, lo cual, nos brindará una comparativa de rendimiento bajo cierto tipo de problemática.

El problema planteado para resolver en las diferentes arquitecturas consiste en hallar la distribución de calor en una superficie de dos dimensiones, para lo cual, es necesario la resolución de la ecuación de Laplace cuya solución puede ser analítica, o entre otras, una solución con métodos numéricos. Este último enfoque permite llevar la ecuación continua a una forma discreta, lo cual significa que podremos computar los resultados mediante algoritmos computacionales. Para realizar una primer aproximación al lenguaje computacional se realizará una

implementación en alto nivel con el lenguaje python y en los apartados correspondientes a las arquitecturas se llevará el modelo propuesto a los respectivos lenguajes de opencl y vhdl.

Objetivo específico

El problema que se propone resolver es la ecuación de calor de Laplace, un problema altamente paralelizable. Este problema cuenta con operaciones entre números reales que le dan a sus algoritmos un tinte interesante para su estudio comparativo.

El trabajo consiste en analizar el rendimiento operacional de punto fijo en las siguientes arquitecturas:

- Implementación del desarrollo en opencl para medir su rendimiento en las GPU´s.
- Implementar el desarrollo en VHDL para medir su rendimiento en la FPGA.
- Implementar el desarrollo mixto, híbrido, mediante opencl en FPGA (Intel).

Se espera con este trabajo dar a conocer el flujo de desarrollo en las diferentes tecnologías y obtener una comparativa de las distintas implementaciones.

Objetivos

Los avances tecnológicos en el campo de inteligencia artificial, big data, robótica, entre otros, requieren gran capacidad de procesamiento. Las unidades de procesamiento gráfico (GPU) están cubriendo este campo, sin embargo, una alternativa planteada recientemente, ha sido la introducción de las FPGA. Sintetizando hardware a la medida del problema, incluso, a partir de la codificación de algoritmos en alto nivel.

En principio las GPU surgieron para cubrir una necesidad gráfica en los computadores, y luego, se vio orientada no solo a dichas prestaciones, sino a diferentes soluciones a problemas con altos procesamientos matemáticos. Como consecuencia actualmente las GPU no solo se utilizan a nivel gráfico, sino que nos ayudan a paralelizar algoritmos que requieren una gran cantidad de cálculos.

Por otro lado, surgió la idea de configurar hardware a medida para crear circuitos digitales de todo tipo, estos dispositivos se denominan FPGA. Al igual que las GPU, se pudo ver que esta tecnología podría utilizarse para cálculos específicos con hardware adaptado especialmente al problema.

Recientemente el mercado se vio impulsado a la combinación de las técnicas ya propuestas con lenguajes paralelizables y las FPGA llamados sistemas híbridos, donde se convierte código en alto nivel paralelizable a un hardware específico para el problema.

El presente trabajo tiene como finalidad comparar la implementación de un problema de aplicación real en diferentes tipos de tecnologías: GPU y FPGA; ambas pueden ser utilizadas para resolver problemas altamente paralelizables haciendo uso de distintas técnicas o mecanismos planteados por cada arquitectura. Al implementar una posible solución al problema es fundamental conocer los dispositivos, lo cual, nos brindará una comparativa de rendimiento bajo cierto tipo de problemática.

El problema planteado para resolver en las diferentes arquitecturas consiste en hallar la distribución de calor en una superficie de dos dimensiones, para lo cual, es necesario la resolución de la ecuación de Laplace cuya solución puede ser analítica, o entre otras, una solución con métodos numéricos. Este último enfoque permite llevar la ecuación continua a una forma discreta, lo cual significa que podremos computar los resultados mediante algoritmos computacionales. Para realizar una primer aproximación al lenguaje computacional se realizará una

implementación en alto nivel con el lenguaje python y en los apartados correspondientes a las arquitecturas se llevará el modelo propuesto a los respectivos lenguajes de opencl y vhdl.

Objetivo específico

El problema que se propone resolver es la ecuación de calor de Laplace, un problema altamente paralelizable. Este problema cuenta con operaciones entre números reales que le dan a sus algoritmos un tinte interesante para su estudio comparativo.

El trabajo consiste en analizar el rendimiento operacional de punto fijo en las siguientes arquitecturas:

- Implementación del desarrollo en opencl para medir su rendimiento en las GPU´s.
- Implementar el desarrollo en VHDL para medir su rendimiento en la FPGA.
- Implementar el desarrollo mixto, híbrido, mediante opencl en FPGA (Intel).

Se espera con este trabajo dar a conocer el flujo de desarrollo en las diferentes tecnologías y obtener una comparativa de las distintas implementaciones.

1 Ecuación de Calor

La ecuación de Laplace es utilizada en numerosos campos de la física, matemática e ingeniería, modelanndo fenómenos estacionarios. Describe el comportamiento del potencial eléctrico, la gravedad, la conducción del calor, entre otros. En termodinámica, la ecuación de Laplace (ecuación 1.1), recibe el nombre de ecuación de calor y describe la distribución de temperatura en una superficie.

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{1.1}$$

Con esta fórmula el objetivo es simular la distribución de la temperatura en un plano bi-dimensional a lo largo del tiempo. La solución analítica de problemas de transferencia de calor, en muchos casos implica un desarrollo complejo haciendo de la solución un trabajo difícil. Por lo tanto, se recurre a la solución mediante métodos numéricos por computadoras.

Método numérico

Se discretiza la superficie bidimensional dividiéndola en segmentos iguales en ambas direcciones, es decir, coordenadas x e y, como se muestra en la Figura 1.1. Los incrementos en x se designan con la letra m y los incrementos en y se denotan mediante la letra n (todos los puntos ubicados equidistantemente). Para establecer la temperatura en cualquiera de estos puntos dentro del cuerpo bidimensional se utiliza la ecuación 1.1. Las diferencias finitas se utilizan para aproximar incrementos diferenciales en las coordenadas de temperatura y espacio; cuanto más pequeño sean estos incrementos finitos, mayor será la precisión obtenida.

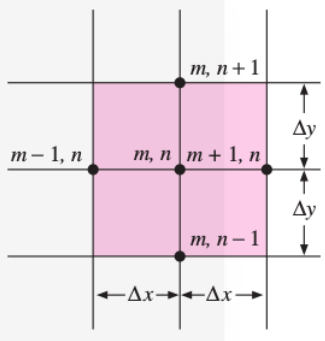

Figura 1.1: Esquema ilustrativo de la nomenclatura utilizada

El gradiente de la temperatura pueden escribirse como se muestra en la ecuación 1.2.

$$\left.\frac{\partial T}{\partial x}\right]_{m+1/2,n} \approx \frac{T_{m+1,n} - T_{m,n}}{\Delta x} \tag{1.2}$$

$$\left.\frac{\partial T}{\partial x}\right]_{m-1/2,n} \approx \frac{T_{m,n} - T_{m-1,n}}{\Delta x}$$

$$\left.\frac{\partial T}{\partial y}\right]_{m,n+1/2} \approx \frac{T_{m,n+1} - T_{m,n}}{\Delta y}$$

$$\left.\frac{\partial T}{\partial y}\right]_{m,n-1/2} \approx \frac{T_{m,n} - T_{m,n-1}}{\Delta y}$$

Por lo tanto, las derivadas parciales de segundo orden quedan como se muestra en la ecuación 1.3.

$$\left.\frac{\partial^2 T}{\partial x^2}\right]_{m,n} \approx \frac{\left.\frac{\partial T}{\partial x}\right]_{m+1/2,n} - \left.\frac{\partial T}{\partial x}\right]_{m-1/2,n}}{\Delta x} = \frac{T_{m+1,n} + T_{m-1,n} - 2T_{m,n}}{(\Delta x)^2} \tag{1.3}$$

$$\frac{\partial^2 T}{\partial y^2}\Big]_{m,n} \approx \frac{\frac{\partial T}{\partial y}\Big]_{m,n+1/2} - \frac{\partial T}{\partial y}\Big]_{m,n-1/2}}{\Delta x} = \frac{T_{m+1,n} + T_{m-1,n} - 2T_{m,n}}{(\Delta x)^2}$$

Por lo tanto, la aproximación de diferencias finitas de la ecuacion 1.1 es:

$$\frac{T_{m+1,n} + T_{m-1,n} - 2T_{m,n}}{(\Delta x)^2} + \frac{T_{m,n+1} + T_{m,n-1} - 2T_{m,n}}{(\Delta y)^2} = 0$$

Considerando que $\Delta x = \Delta y$ y trabajando algebraicamente la expresión, se obtiene la siguiente ecuación:

$$T_{m,n} = \frac{1}{4}(T_{m+1,n} + T_{m-1,n} + T_{m,n+1} + T_{m,n-1}) \tag{1.4}$$

Resolución computacional

A continuación se proporcionará una primer versión expresada en pseudo lenguaje cuya finalidad es realizar los cálculos de la ecuación 1.4. Se llamará a esta versión INICIAL (versión 1).

Para las diferentes versiones se fijarán las siguientes constantes:

- N: dimensión de la matriz (NxN)
- TEMP_BORDE: temperatura de borde
- TEMP_INICIAL: temperatura inicial de la matriz
- TEMP_FUENTE: temperatura de la fuente de calor
- FUENTE_X: posición de la fuente en las ordenadas
- FUENTE_Y: posición de la fuente en las abscisas
- ITERACIONES: cantidad de iteraciones a realizar

Se utilizará punto fijo para los valores de temperatura, por esto es indispensable para la comparativa de eficiencia contra la FPGA, dependerá de la excursión y precisión necesaria donde se ubica la coma decimal dentro de los 32 bits.

VERSION 1: INICIAL

```
1
2  VAR Matriz= matriz[N,N];
3
4  // Inicializa la matriz con los respectivos valores
5  inicializar(Matriz,TEMP_INICIAL,TEMP_FUENTE);
6
7  // Inicializa la matriz con los respectivos valores
8  function esBorde(fila,col){
9      return fila==0 or col==0 or col == N-1 or fila == N-1;
10 }
11
12 // funcion que realiza el calculo en la matriz
13 function calcular(Matriz){
14     VAR fila,col,valor,matrizCalculada = array[N,N];
15     for(fila = 0; fila<N; fila++){
16         for(col = 0; col<N; col++){
17             if(esBorde(fila,col) or (fila==FUENTE_X and col==FUENTE_Y)){
18                 valor = Matriz[fila][col];
19             }else{
20                 valor = Matriz[fila][col-1] +
21                 Matriz[fila][col+1] +
22                 Matriz[fila-N][col] +
23                 Matriz[fila+N][col] ;
24                 Valor = valor/4;
25             }
26         matrizCalculada[fila][col] = valor;
27         }
28     }
29     Return matrizCalculada;
30 }
31
32 VAR It = 0;
33 while(it<ITERACIONES){
34     Matriz = calcular(Matriz);
35     it++;
36 }
```

La versión INICIAL consiste en un recorrido de fila y columnas. A modo de mejora se propone una segunda versión con la finalidad de reducir las operaciones presentadas en la presente versión.

Versión 2: REDUCIDA

Se propone una segunda versión reducida cuyo fin es reducir el costo operacional, para lo cual, se proponen dos vectores con la siguiente finalidad: en un primer vector representará la matriz de valores y en un segundo vector representará los bordes de la matriz. El mapeo de la matriz al respectivo vector se muestra en la figura 2.9.

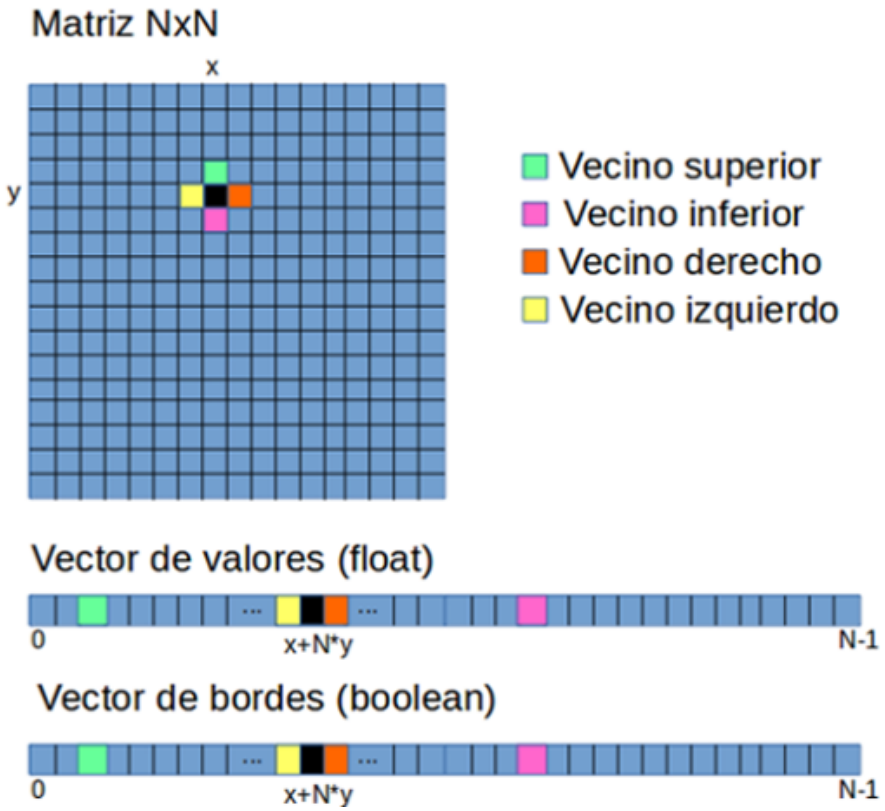

Figura 1.2: Mapeo de matriz NxN a vector[NxN]

Donde la posición $x + N \times y$ corresponde al mapeo posicional de la coordenada (x, y) de la matriz, respectiva ubicación en el vector; finalmente los vecinos están dados por:

- Vecino derecho: $x + 1$
- Vecino izquierdo: $x - 1$
- Vecino superior: $x + N$
- Vecino inferior: $x - N$

Con esto se propone la versión reducida de la siguiente manera:

```
1 VAR Matriz  = vector[N*N];
2 VAR Bordes  = vector[N*N];
3
4 // Inicializa la matriz y los Bordes en los respectivos arrays
5 inicializar(Matriz,Bordes,TEMP_INICIAL,TEMP_FUENTE);
6
7 var It = 0,NxN=N*N,pos;
8 while(it<ITERACIONES){
9     pos = 0;
```

```
10      while(pos<NxN){
11          if(Bordes[pos]){
12              matrizCalculada[pos]=Matriz[pos];
13          }else{
14              matrizCalculada[pos]=(
15                      Matriz[pos-1]+
16                      Matriz[pos+1]+
17                      Matriz[pos-N]+
18                      Matriz[pos-N])*.25;
19          }
20          pos++;
21      }
22      Matriz = matrizCalculada;
23      it++;
24  }
```

Debe notarse que la fuente de calor es contemplada como un borde dado que cumple las mismas condiciones de no ser modificado. La versión REDUCIDA propone reducir el costo operacional a cambio de un ligero incremento de memoria (un segundo vector de valores booleanos).

Verificación

Para verificar los resultados se proponen dos escenarios desarrollados en el lenguaje python:

1. FUENTE DE CALOR SIN TEMPERATURA DE BORDE: Generar una fuente de calor en la matriz sin temperatura de borde para observar cómo se propaga el calor de un punto al resto de la matriz.
2. AUSENCIA DE FUENTE DE CALOR CON TEMPERATURA DE BORDE: se propone el caso de aplicar solo una temperatura en el borde sin fuente de calor para ver su propagación desde los bordes al centro de la matriz.

Los ejemplos, a modo de verificación, se realizaron en python y nos permiten ver a continuación los resultados de manera gráfica de la versión reducida (en el apéndice se puede encontrar la implementación del código propuesto A.1.1). A continuación se observan los resultados de ambos casos.

CASO 1: FUENTE DE CALOR SIN TEMPERATURA DE BORDE

Parámetros:

- N : 15

- TEMP_BORDE : 0
- TEMP_INICIAL : 0
- TEMP_FUENTE : 5
- FUENTE_X : 10
- FUENTE_Y : 3
- ITERACIONES : 10

Resultado: Luego de 10 iteraciones en la figura 1.3 se observa el gráfico mapa de calor generado a partir de las operaciones.

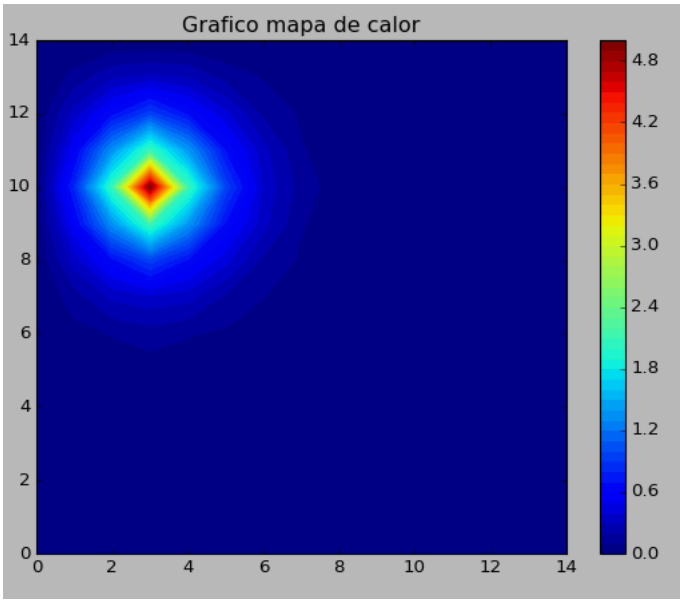

Figura 1.3: Mapa de calor CASO 1

CASO 2: AUSENCIA DE FUENTE DE CALOR CON TEMPERATURA DE BORDE:

Parametros:

- N : 25
- TEMP_BORDE : 10
- TEMP_INICIAL : 0
- TEMP_FUENTE : 10
- FUENTE_X : 0

- FUENTE_Y : 0
- ITERACIONES : 10

Resultado: Luego de 10 iteraciones en la figura 1.4 se observa el gráfico mapa de calor generado a partir de las operaciones.

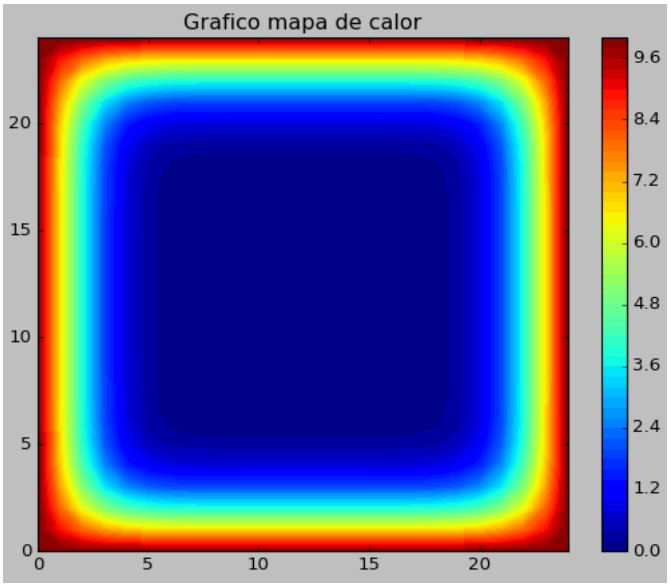

Figura 1.4: Mapa de calor CASO 2

2 FPGA

En 1985 se introdujeron dispositivos de hardware configurable, llamados FPGA (Field Programmable Gate Array), cuya finalidad es proveer al desarrollador un entorno con el cual lograr diseños electrónicos digitales de pequeña y alta complejidad. Su nombre, traducido del inglés, significa matriz de compuertas programables y consisten en una matriz de bloques configurables que se pueden conectar mediante recursos de interconexión. La evolución de diseños electrónicos, a partir de esta tecnología, se ha tornado de mucho interés, por lo que se están desarrollando continuamente nuevas herramientas y arquitecturas.

Las aplicaciones donde más comúnmente se utilizan las FPGA incluyen al procesamiento digital de señales, sistemas aeroespaciales y de defensa, prototipos de ASIC, sistemas de imágenes para medicina, sistemas de visión para computadoras, entre otras.

En esta primera parte se realizará una breve descripción de los conceptos básico del desarrollo en FPGA. Es necesario conocer las herramientas de trabajo como sintetizadores, simuladores, IDE's de desarrollo, lenguajes de descripción de hardware, entre otras. Dichos conceptos/herramientas serán mencionadas en el presente capítulo en el siguiente orden:

- Lenguaje de descripción de hardware: permiten modelar hardware mediante código de alto nivel.
- Arquitectura de las FPGA: conceptos generales del funcionamiento de las FPGA.
- Flujo de diseño: etapas de proceso que convierte el código a hardware en la FPGA.
- Archivos de *constraints*: permite decirle a las FPGA que recursos utilizar entre otras características.

Para finalizar el capítulo se realizará la ia implementación de la ecuación de Calor mencionada en el capitulo 1.

Lenguaje descripción de hardware

La creciente complejidad del desarrollo electrónico llevó a la ingeniería a crear un lenguaje de descripción de hardware (HDL, hardware description language) el cual nos permite el modelado de diferentes circuitos mediante la descripción de código. Existen diferentes lenguajes, en lo que respecta a las FPGA se utilizan VHDL y VERILOG, los cuales se basan un estándar de la IEEE y trabajan bajo el concepto de modularidad (dividir el diseño en componentes más pequeños).

Arquitectura FPGA

La arquitectura de las FPGA varían según el fabricante: se ofrecen dispositivos de diferentes gamas, de pequeño a alto rendimiento, se ofrece también distintos bloques de propiedad intelectual (IP), y prestaciones complejas como USB, conexión a integrados externos de memoria RAM, conexiones Ethernet, etc. De forma general, y simplificada, en la figura 2.1 se observa como esta compuesta la arquitectura de una FPGA con sus respectivos bloques internos:

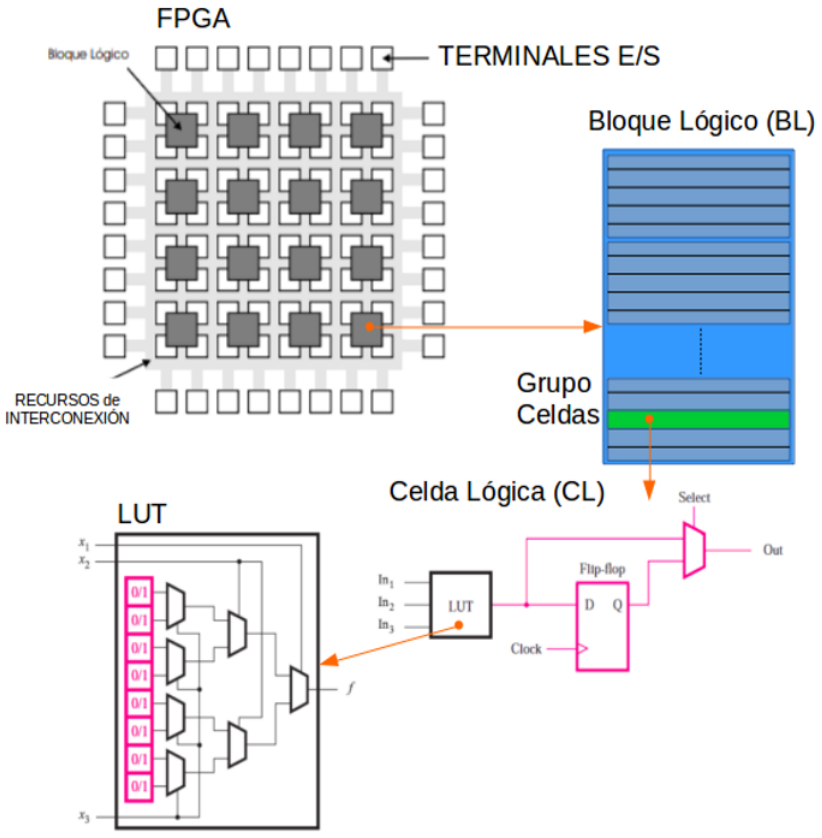

Figura 2.1: Arquitectura general FPGA

Se destacan tres aspectos fundamentales en las FPGA:

- Bloques lógicos
- Recursos de interconexión
- Terminales E/S

Bloques lógicos

Los bloques lógicos (BL) están formados un conjunto de celdas lógicas (CL). Las cuales representan la parte más básica de las FPGA. Las CL implementan funciones booleanas y los BL pueden interconectar varias de ellas para obtener funciones más complejas. Generalmente incluyen registros para la implementación de circuitos secuenciales.

Las CL se implementa utilizando tablas de búsqueda llamadas LUT (Look-Up Table), es decir, funciones lógicas almacenadas en tablas de verdad. La cantidad de entradas disponibles para una LUT determina su tamaño. Una LUT con n entradas comprende 2^n celdas de memoria de un solo bit seguidas por un multiplexor 2^{n-1}: 1. En la figura 2.2 se observa una LUT de tres entradas.

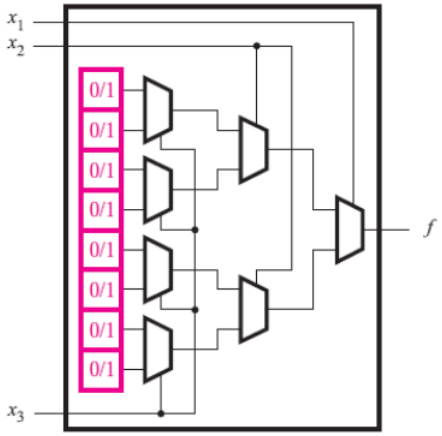

Figura 2.2: LUT de 3 entradas

La CL, además cuenta con un multiplexor y un flip flop. Esto último da la posibilidad de crear logica combinacional o secuencial.

NOTA: Los BL recibe diferentes nombres según el fabricante: CLB (llamados así por Xilinx), o LAB (llamados así por Altera). Del mismo modo la celda lógica reciben diferentes nombre: celda lógica (CL=logic cell, en Xilinx) o elemento lógico (LE=logic element, en Altera)

Recursos de interconexión

Las FPGA cuentan con una variedad de recursos, además de los bloques lógicos: bloques de memoria, multiplicadores, etc. Mediante la interconexión se provee la forma de conectar

los recursos suministrados por el fabricante.

Terminales E/S

Para el intercambio de señales digitales con el exterior las FPGA requiere de entradas y salidas (E/S) llamados terminales que permiten la interacción fuera del integrado.

Flujo de diseño

El modelado a partir del código HDL utilizado, cumple la función de representar al hardware requerido. Mediante las diferentes etapas de trabajo (flujo de diseño) se realiza el proceso de de conversión de dicho modelo al hardware dentro de una FPGA. El flujo de diseño se muestra en la figura 2.3, cuyas etapas más importantes son:

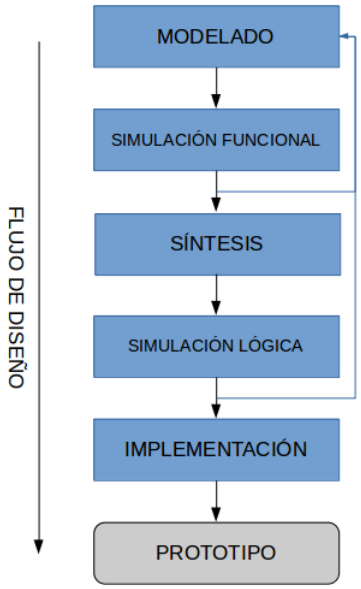

Figura 2.3: Flujo de diseño

- MODELADO: diseño y modelado del hardware en lenguaje alto nivel HDL.
- SIMULACIÓN FUNCIONAL: consiste en comprobar que el funcionamiento del sistema es el correcto. En las etapas de simulación se utilizan herramientas que permite ver las señales digitales generadas por el modelo propuesto.
- SÍNTESIS: se convierte el código HDL a un circuito digital implementable en el dispositivo. La herramienta de síntesis buscará la forma óptima para el dispositivo.
- SIMULACIÓN LÓGICA: se utiliza para comprobar el código sintetizado.

- IMPLEMENTACIÓN: consiste en realizar la distribución del diseño en el dispositivo elegido (place) y realizar la interconexión de los recursos (route).

Arquitectura FPGA ALTERA

El siguiente apartado tiene como finalidad mostrar la arquitectura de una FPGA comercial. Dado que ALTERA posee rasgos más sencillos e intuitivos que otros fabricantes y que el código HDL desarrollado puede sintetizarse e implementarse para ambos dispositivos (Altera y Xilinix), se reflejaran los conceptos mencionados anteriormente en la FPGA de ALTERA. Los conceptos a tener en cuenta son:

- Bloques lógicos (elemento lógico)
- Recursos de interconexión
- Terminales E/S

Bloques lógicos

En la figura 2.4 se observa el elemento lógico (LE) de una FPGA de Altera, el cual está compuesto principalmente por una LUT, un registro y otras compuertas básicas.

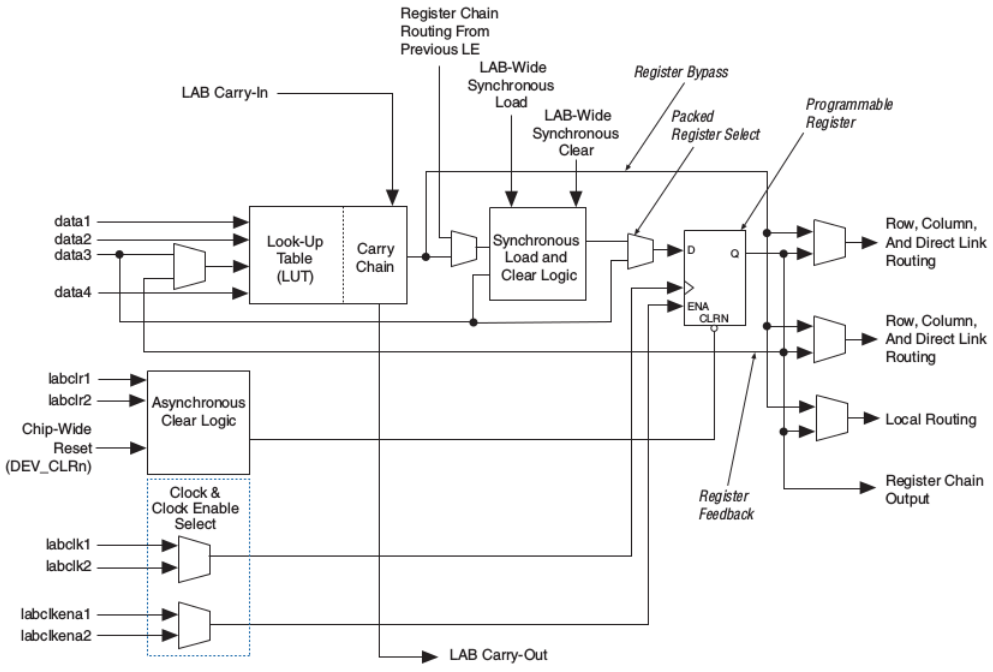

Figura 2.4: Elemento lógico

El registro LE puede configurarse como flip flop tipo D, T, JK o SR. Cada LE tiene señales de datos, reloj, habilitación de reloj y control. Cuenta con salidas de datos y control que

pueden dirigirse a los recursos globales de interconexión, al recurso local de interconexión o a otros LEs. Existen múltiples señales para la interconexión con otros LABs, lo que permite al sintetizador contar con mayor flexibilidad en los recursos y obtener así funcionalidades más complejas.

Recursos de interconexión

Los Bloques de arreglos lógicos son grupos de LE´s llamados LAB (estudiado como bloque lógico en la sección anterior), los cuales se hallan interconectados como se muestra en la siguiente figura 2.5.

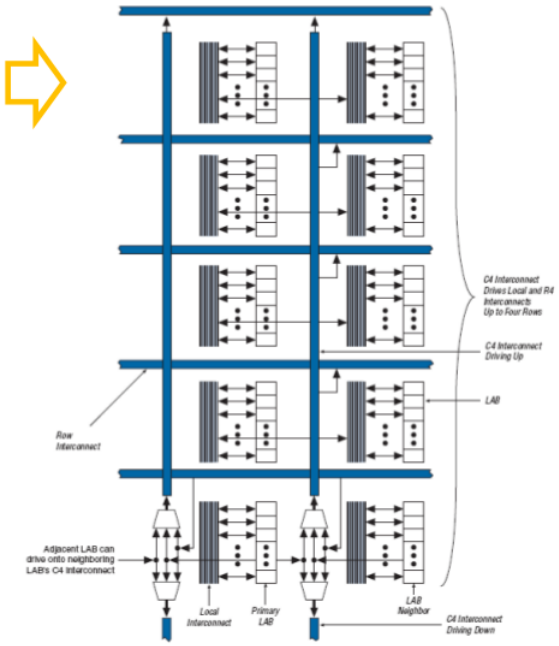

Figura 2.5: Bloques lógicos

La interconexión nos permite vincular LES vecinas o varios niveles de grupo de LES con la finalidad de sintetizar funciones más complejas, pero, la interconexión entre LES lejanas resulta más escasa. Finalmente, en la figura 2.6 se observa ver la interconexión generalizada de la FPGA.

Además de los LAB, la FPGA provee recursos exclusivos para su utilización, con lo que es posible potenciar más el uso de las mismas: por ejemplo, bloques de memoria y multiplicadores. En la figura 2.7 se observa el diagrama general que representa a la FPGA con todos sus recursos.

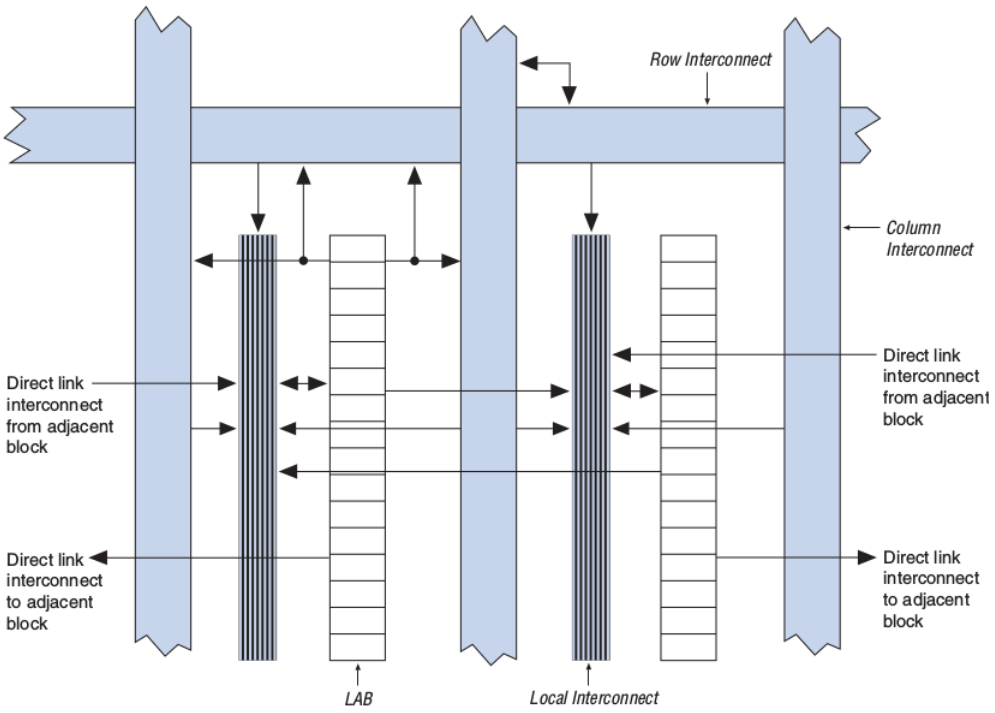

Figura 2.6: Interconexión

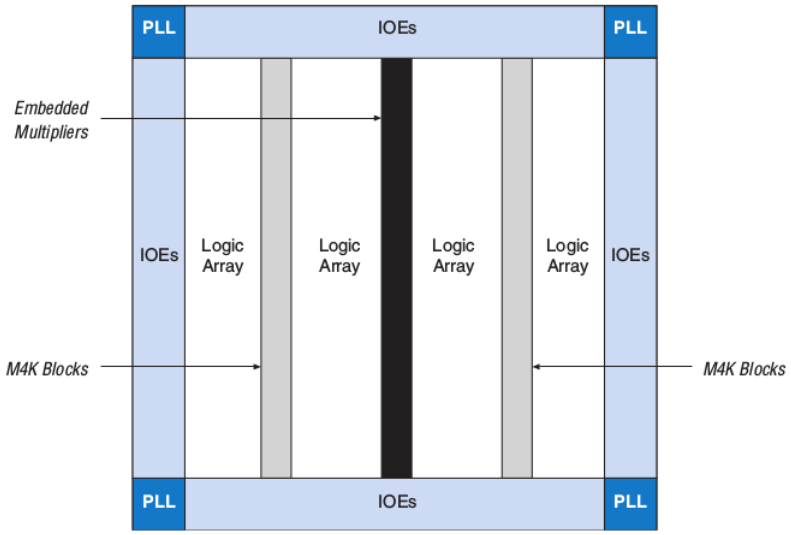

Figura 2.7: Interconexión

Nos resta decir que los PLL se utilizan para modificar la frecuencia de los relojes, IOEs representa las entradas y salidas.

Implementación ecuación de calor en FPGA

En el presente apartado se detalla la implementación de la ecuación de calor mediante el lenguaje de descripción VHDL. El objetivo no es detallar el código propuesto en VHDL, sino, comprender la arquitectura del diseño. Se distingue una forma genérica de cómo se abordó el problema y una segunda forma que explique detalles técnicos.

Implementación

Implementación de la matriz: la implementación, a grandes rasgos, consiste en realizar una $celda$ con la cual se construye una pequeña matriz de 2 filas por 2 columnas, y a su vez, a partir de ella, construir una matriz N x N (matrices múltiplo de dos). La $celda$ cuenta con la lógica combinacional para realizar la operación de punto fijo (sumar los cuatro valores vecinos y dividirlos por cuatro) y se le indica si debe o no realizar la operación; Cada $celda$ combinada en un módulo de 2x2 ($macrocelda2X2$) para que finalmente se construya el módulo matrizNxN. En la figura 2.8 se muestra la secuencia seguida para la construcción.

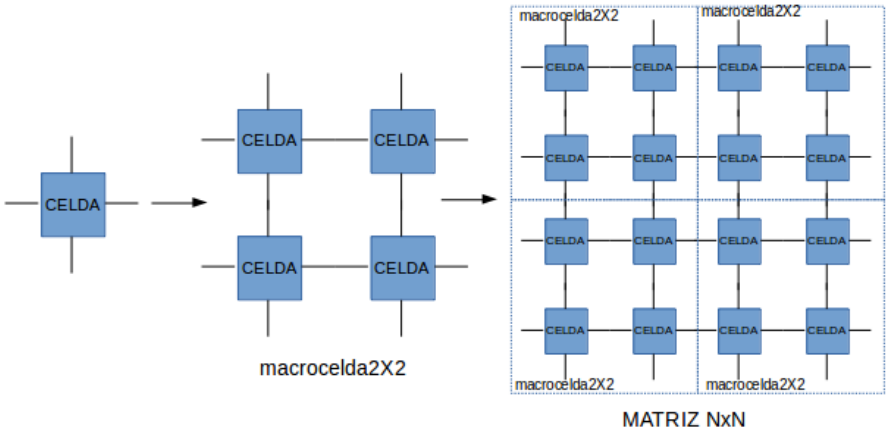

Figura 2.8: Celdas y Matriz

Inicialización de la matriz: Los datos iniciales de la matriz de calor se encuentran almacenados en un bloque de memoria interna de la FPGA. Cada posición de memoria contiene el dato de una $macrocelda2x2$, dado que el valor de temperatura de cada celda se representa en un número de punto fijo de 32 bits, el ancho de la memoria es de 128 bits.

Cada $celda$ se mapea a una $macrocelda2x2$ específica, es decir, la posición 0 de la memoria es mapeada a la $macrocelda2x2[0]$ de la matriz que a su vez particiona los valores de memoria 0...128, asignando un valor a cada celda de la macrocelda, la posición 1 al la $macrocelda[1]$ y así sucesivamente. La dirección de dichas posiciones se genera a partir de un contador como se ve en la figura 2.9. El objetivo de utilizar $macroceldas2x2$ es de disminuir el tiempo de lectura y escritura por un factor de 4, ya que, por cada ciclo de clock se escriben o leen cuatro celdas.

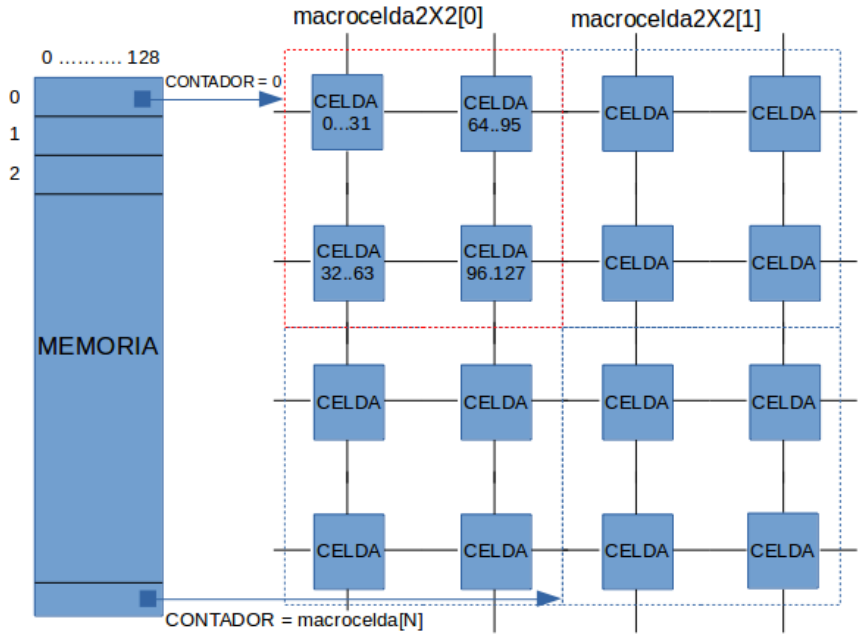

Figura 2.9: Mapeo de memoria a matriz

Sincronización de tareas: Finalmente, se coordina las diferentes tareas mediante una máquina de estado: iniciar la carga de valores de la matriz, iniciar el cálculo de n iteraciones, finalizar y retornar los valores a la memoria. En la imagen 2.10 se observa la interacción realizada entre la máquina de estado y los diferentes módulos.

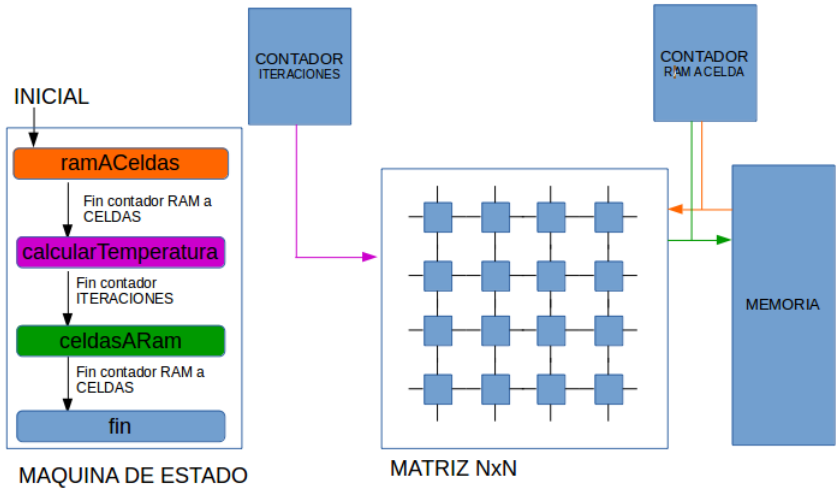

Figura 2.10: Interconexion maquina de estado

Cada estado (en un color distinto) activa las respectivas señales (indicadas con su mismo color).

Detalles de la implementación

A continuación se analiza la descripción propuesta con detalles técnicos con algunos de los esquemáticos respectivos. Los modelos propuestos se basan en los siguientes objetivos:

calculoDeCelda

Implementa una celda, con su respectiva operación de cálculo. Dado cuatro valores de N bits realiza la suma y división por 4; además, es posible configurar un valor inicial e indicarle cuándo debe operar; el resultado es asignado a las salidas respectivas de la celda (superior, inferior, derecha e izquierda). En la figura 2.11 y figura 2.12 se muestra el esquemático de *calculoDeCelda*.

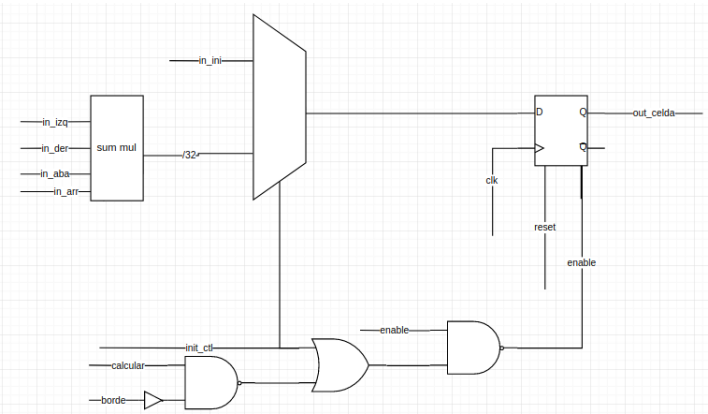

Figura 2.11: calculoDeCelda

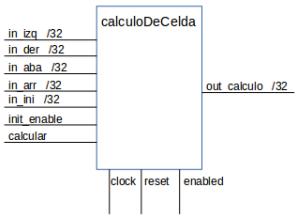

Figura 2.12: calculoDeCelda

macroCelda2x2

Instanciación de 4 bloques *calculoDeCeldas* formando la matriz de 2x2. A partir de 8 valores de temperatura de *macroceldas2x2* vecinas, se calculan 4 resultados, uno por cada celda.En la figura 2.13 se muestra el esquemático de la *macroCelda2x2* con las respectivas conexiones internas.

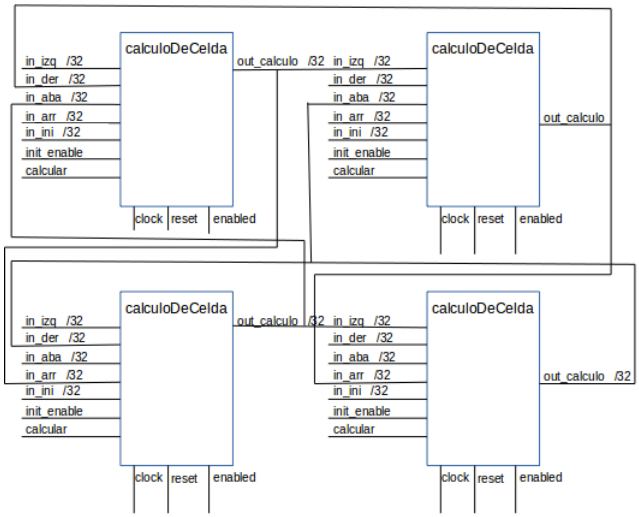

Figura 2.13: macroCelda2x2

macroCelda2x2

Matriz formada por el módulo $macroCelda2x2$ para generar una matriz NxN.

counter

El módulo counter se utilizar para contar desde el valor inicial cero hasta un valor MAX_COUNT. El mismo contiene un flip-flop que mantiene el valor de salida del contador. Se genera una señal $counter\_end$ para indicar la finalización del mismo; el contador se inicializa cuando la señal $load$ lo indica. En la figura 2.14 se puede visualizar el diagrama correspondiente.

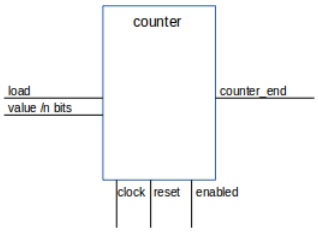

Figura 2.14: counter

memory

Bloque de memoria que contiene los datos de la matriz al momento de iniciar el trabajo y el resultado de la misma al finalizar.

Control_machine

La máquina de control se basa en una máquina de estados y es la encargada de sincronizar las tareas que deben realizar cada módulo. Dado que el esquemático no aporta información relevante, se representa el diseño mediante un diagrama de estado que se muestra la figura 2.15.

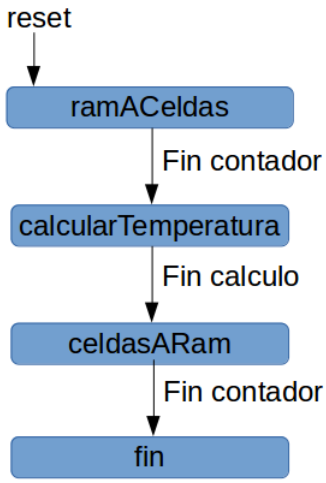

Figura 2.15: Control_machine

La máquina de estado realiza las siguientes tareas en los respectivos estados:

- $ramACeldas$: se trasladan los valores de la memoria a la matriz (celdas). Para tal caso es inicializado el contador en cero y se habilita la memoria en modo lectura para obtener el respectivo valor cuya dirección se indica en el contador. Durante este proceso, la matriz de celdas se encuentra deshabilitada para el cálculo.
- $calcularTemperatura$: Se habilita la señal de cálculo para iniciar el proceso, además, se inicia un segundo contador cuya finalidad es controlar la cantidad de iteraciones (cantidad de cálculos). En este estado, la memoria y el contador asociado a ella, se encuentran desabilitados. Una vez finalizada las iteraciones se indica el fin de cálculo.
- $celdasARam$: se traslada los valores de la matriz (desde cada $macroCelda$) a la memoria. El proceso consiste en desabilitar el cálculo de la matriz de celdas, iniciar el contador, colocar la memoria en modo escritura y trasladar los datos de la matriz a la memoria.
- fin: se deshabilitan todas las señales de los módulos poniendo contadores y otras señales a cero indicando el fin del proceso de cómputo.

NbitDecoder

Genera un decodificador de N bits para la activación secuencial de las macroCeldas.

CalculoMatrix

Integra la totalidad del diseño (top), instancia los bloques *counter*, *memory*, *matrixDeCeldas*, *NbitDecoder* y *control_machine*, realizando los cálculos de la ecuación de calor. En la figura 2.16 se observa la interconexión final entre cada bloque teniendo como principal "coordinador" la maquina de control. En este bloque se hallan las constantes correspondientes al problema: dimensión de la matriz (NxN), temperatura de borde, temperatura inicial de la matriz, cantidad de iteraciones a realizar, etc. Dichos valores son propagados a los módulos correspondientes.

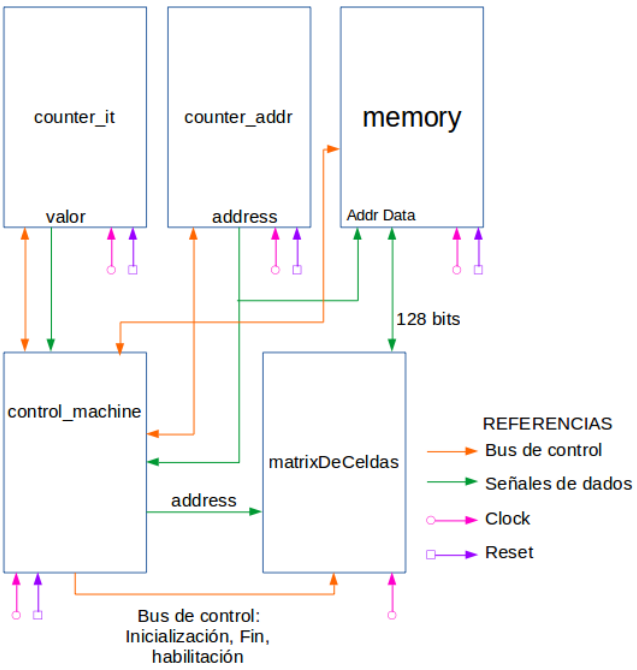

Figura 2.16: CalculoMatrix

Mediciones

Para las mediciones se dispone de una placa Xilinx Artix-7 la cual posee las siguientes características:

PLACA DISPONIBLE: Xilinx Artix-7 FPGA XC7A100T-1CSG324C

- 15,850 logic slices, each with four 6-input LUTs and 8 flip-flops
- 4,860 Kbits of fast block RAM

- Six clock management tiles, each with phase-locked loop (PLL)
- 240 DSP slices
- Internal clock speeds exceeding 450 MHz
- 16Mbyte CellularRAM^R
- Two 4-digit 7-segment displays

Tiempos de cálculo

Mediante la simulación post implementación se analizaran los tiempos que se requiere para llevar adelante el procesamiento (cantidad de ciclos de reloj por cada estado). En la imagen 2.17 se proporcionan los valores por estado:

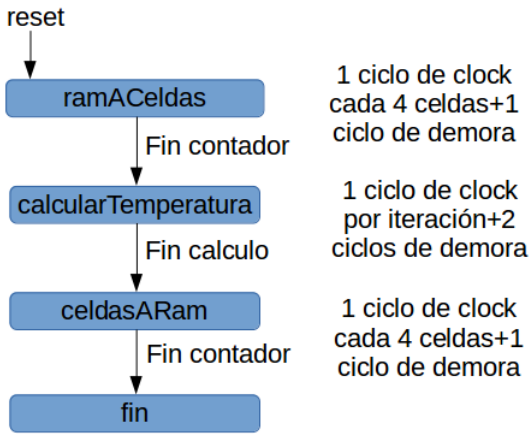

Figura 2.17: Ciclos de reloj por estado

Los ciclos de demora (+1, +2) se deben a la lógica de control generada por la máquina de estado. Se puede deducir el tiempo de análisis para una matriz $2Mx2M$ en un periodo P (en $[\mu s]$) con IT iteraciones, por:

Tiempo $ramACeldas = P \times (M * M + 1)$ $[\mu s]$

Tiempo $celdasARam = P \times (M * M + 1)$ $[\mu s]$

Tiempo $calcularTemperatura = P \times (IT + 2)$ $[\mu s]$

Tiempo de cálculo $= P \times [2 \times (M * M + 1) \times (IT + 2)]$ $[\mu s]$

Las pruebas se han realizado sobre el siguiente conjunto de matrices: 8x8, 16x16, 24x24, 32x32 con iteraciones de 5,10,50,100,500,1000,5000 y 10000. La elección de la matriz 32x32 como valor mayor se debe a las características de dicha FPGA mientras que las iteraciones varían en un rango de pocas iteraciones a una cantidad considerable de casos.

Tiempo de cómputo en ms

	8	16	24	32
5	0.00548	0.02084	0.04644	0.08228
10	0.00568	0.02104	0.04664	0.08248
50	0.00728	0.02264	0.04824	0.08408
100	0.00928	0.02464	0.05024	0.08608
500	0.02528	0.04064	0.06624	0.10208
1000	0.04528	0.06064	0.08624	0.12208
5000	0.21048	0.2412	0.2924	0.36408
10000	0.40528	0.42064	0.44624	0.48208

Tabla 2.1: Tiempo de computo

En la figura 2.18 se observa el resultado interpretado gráficamente, las iteraciones en función del tiempo por cada matriz propuesta NxN:

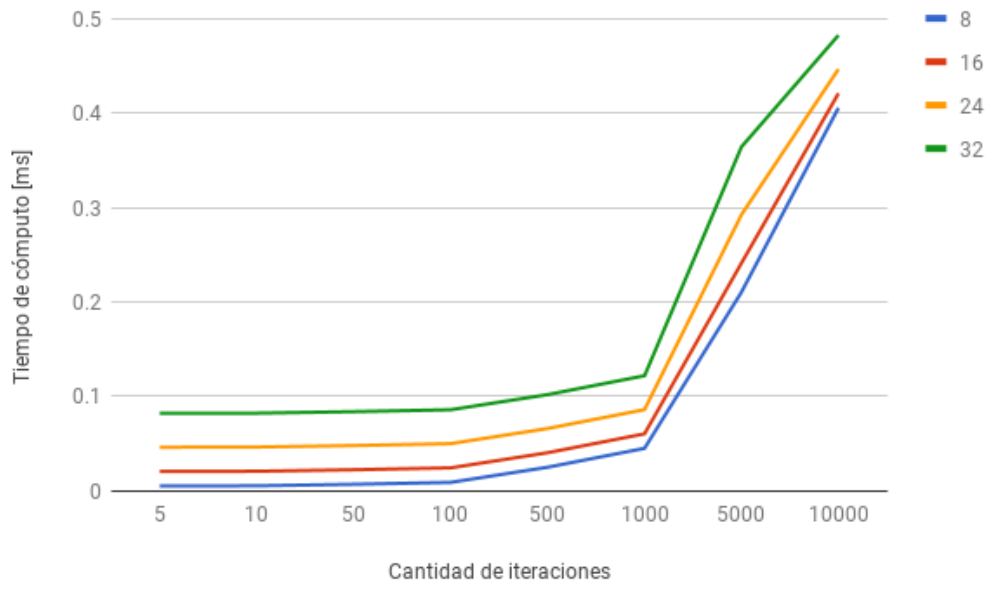

Figura 2.18: Tiempo de computo

Consumo de potencia

En la tabla 2.2 se muestra el consumo de potencia de la FPGA para los diferentes tamaños de matriz, estos fueron calculados mediante la herramienta Vivado. Para obtener estos valores se realizó la simulación post-implementación con la que se analizó la actividad de conmutación de todos los transistores de la FPGA. Como resultado de esta simulación se generó un archivo .saif utilizado para calcular los valores de potencia.

Dimensión	Potencia (W)
8	0.099
16	0.11
24	0.364
32	0.564

Tabla 2.2: Consumo de potencia

En la figura 2.19 y figura 2.20 se muestra un gráfico generado por la herramienta Vivado donde puede verse las componentes del valor final de potencia para una matriz de 8x8 y de 32x32. Como puede verse el consumo de potencia de una matriz de 8x8 está dado mayoritariamente por la potencia estática siendo la dinámica menor al 10 %, esto se debe a que la utilización de la FPGA es muy baja, en cambio, en el cálculo de una matriz de 32x32 el consumo está dado por la potencia dinámica ya que la utilización de la FPGA es cercana al 100 %.

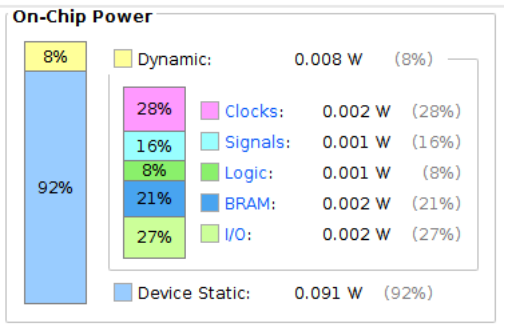

Figura 2.19: Medición potencia matriz 8x8

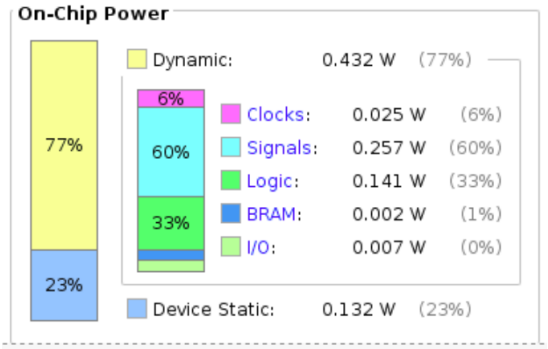

Figura 2.20: Medición potencia matriz 32x32

Conclusiones

- El tiempo de cálculo se ve poco afectado por la dimensión de la matriz, permitiendo calcular matrices tan grande como el dispositivo lo permita sin penalizar el tiempo considerablemente. Sin embargo, el número de iteraciones incide directamente sobre el tiempo de cómputo.
- El consumo de potencia se ve afectado significativamente con el tamaño de la matriz. Por otro lado, un número de iteraciones mayor implica mayor tiempo de cómputo, esto no incide en el consumo instantáneo, pero el dispositivo se mantendrá más tiempo consumiendo la potencia calculada.
- El tiempo de lectura y escritura no pudo ser calculado debido a que la placa de desarrollo únicamente permite conectividades de baja velocidad (usb). Es evidente que estos

tiempos son muy superiores si no se cuenta con puertos de alta velocidad (PCIex). Las placas con FPGA de alto desempeño utilizan interfaces de alta velocidad.

- El tamaño máximo de matriz que se logró sintetizar en este dispositivo fue de 16x16 celdas, sin embargo, se logró obtener matrices más grandes utilizando dispositivos más grandes de la misma gama.

Limitantes

- La dimensión máxima de la matriz depende del dispositivo a utilizar. En este caso se está utilizando uno de la gama mas baja disponible de Xilinx, permitiendo gran escalabilidad.

3 OpenCL

Por más de dos décadas, la industria de la computación se inspiró y motivó en la observación realizada por Gordon Moore (también conocida como la Ley de Moore) que establece que la densidad de transistores en un chip se duplica cada 18 meses. Esto creó la predicción de que las performance de los procesadores se duplicará cada dos años. Entonces, la nueva generación de procesadores podía tener el doble de densidad de transistores, para ganar el 50 % de velocidad y mantener el mismo nivel de potencia. A mediado del 2000 el tamaño del transistor era tan chico que la "física de los dispositivos pequenos"tomó el control de la caracterización del chip. Por lo tanto, los aumentos de frecuencia y densidad de transistores no se pudo lograr sin aumentar significativamente el consumo de potencia.

Para cumplir con la expectativa de duplicar la performance se debieron realizar dos grandes cambios:

- En lugar de aumentar la frecuencia, se aumentó el número de cores en cada pastilla de silicio. A consecuencia, se debió realizar nuevas aplicaciones que saquen provecho de las arquitecturas multicore.
- En el diseño de las nuevas arquitecturas, el consumo de potencia y la distribución de temperatura se volvieron de gran importancia.

En el año 2005, la mayoría de los grandes fabricantes de microprocesadores habían decidido que el camino para el aumento del desempeño, caía en la dirección del paralelismo. En lugar de continuar desarrollando procesadores monolíticos cada vez más rápidos, se comenzó a integrar múltiples procesadores completos en un único circuito integrado. Desarrollar software para sistemas paralelos y distribuidos es considerada una tarea no trivial, aunque los desarrollos utilicen paradigmas y lenguajes conocidos.

OpenCL (.ºpen Computing Language") fue diseñado para suplir esta necesidad. Este lenguaje y su entorno tomaron prestados varios conceptos básicos de otros entornos para hardware específicos como CUDA, CAL, CTM y se los modificó para crear un entorno de

desarrollos de software independientes del hardware. Soporta distintos niveles de paralelismo y mapea el trabajo en forma eficiente en dispositivos homogéneos, heterogéneos, simples o múltiples como CPUs, GPUs, FPGAs. Además, OpenCL permite manejar los recursos existentes y combinar distintos tipo de hardware en el mismo entorno de ejecución.

CPU y GPU

En un comienzo la CPU se ocupó de realizar todo el computo, limitando a la GPU (unidad procesamiento gráfico) únicamente al procesamiento gráfico. A partir del 2007, cuando Nvidia introduce CUDA, la GPU se comenzó a utilizar para cómputo de propósito general (GPGPU), ya que, esta unidad permite realizar el procesamiento de los datos con un elevado grado de paralelismo, mejorando el rendimiento significativamente para cierto tipo de aplicaciones.

La arquitectura de la GPU difiere en gran medida de la del CPU, como puede verse en la figura 3.1. En forma simplificada, la diferencia es que los GPU cuentan con cientos (o miles) de cores más sencillos, en cambio, el CPU cuenta con pocos cores pero de alta complejidad y optimización. De esta forma, la GPU es el dispositivo correcto a utilizar cuando el problema requiere aplicar procesos sencillos a una gran cantidad de datos en forma independiente de ellos. En cambio, el CPU se recomienda cuando se deben realizar procesos complejos sobre baja cantidad de datos.

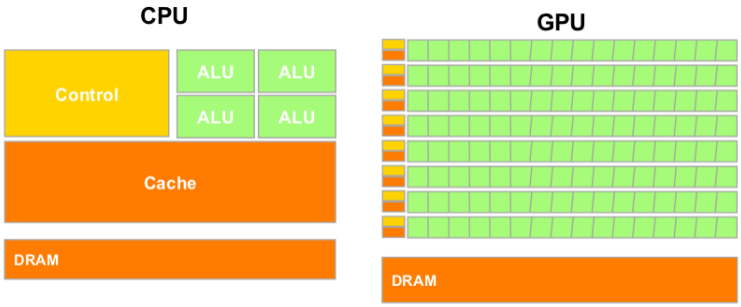

Figura 3.1: Arquitectura CPU y GPU

Arquitectura GPU

Las GPUs tienden altamente al multiproceso con hardware sofisticado para el manejo de tareas debido al tipo de cómputo que implica la parte gráfica. Estas tareas son muy paralelizables, lo que implica una gran cantidad de trabajos independientes a procesar por dispositivos con múltiples cores y multiprocesos altamente tolerante a latencias. Es importante resaltar que a pesar de los sofisiticados mecanismos para el manejo de tareas, o para esconder la ejecución SIMD (del inglés Single Instruction, Multiple Data) detrás del hardware, las GPUs son simplemente procesadores multiprocesos con la parametrización apuntada a procesar grandes números de pixeles muy eficientemente.

La GPU de escritorio de gama alta y sus derivados para la informática de alto rendimiento apuntan al rendimiento y no a la eficiencia energética. Para lograr gran un ancho de banda de memoria, una muchos puertos están dedicados al tráfico de la memoria y, por lo tanto, se pueden utilizar protocolos de memoria de gran ancho de banda por pin (posiblemente de latencia más baja), como GDDR5. Estos dispositivos utilizan una combinación de características para mejorar la tasa de procesamiento, como por ejemplo, amplios arreglos SIMD para maximizar la tasa de procedimientos aritméticos.

Figura 3.2: Arquitectura GPU NVIDIA

La arquitectura de la GPU NVIDIA GeForce GTX 980 (que se muestra en la figura 3.2) está compuesta por cuatro *Graphics Processing Clusters* (GPCs), 16 Maxwell *Streaming Multiprocessors* (SMMs) y cuatro controladores de memoria. Cada SMM tiene 128 cores CUDA, al estar compuesta por 16 SMMs, la GeForce GTX980 contiene 2048 cores CUDA en total. Por otro

lado, esta GPU utiliza cuatro controladores de memoria de 64 bits (256 bits en total), atado a cada controlador de memoria hay 512Kb de caché L2, el chip completo contiene 2048KB de cache L2.

OPENCL

OpenCL consta de una interfaz de programación (API) y un lenguaje de programación abierto mantenido por Khronos. El modelo establecido permite crear programas portables, independientes del proveedor y del dispositivo que son capaces de ser ejecutados en muchas plataformas de hardware diferentes. La API OpenCL está definida en términos del lenguaje de programación C. El código que se ejecuta en un dispositivo OpenCL, que puede ser o no el mismo dispositivo host, está escrito en el lenguaje OpenCL C, el cual es un versión restringida del lenguaje C99 con extensiones apropiadas para ejecutar código paralelo en una variedad de dispositivos.

Estructura OpenCL

La especificación OpenCL se define en cuatro partes, llamadas modelos, que se pueden resumir de la siguiente manera:

1. **Plataforma**: se especifica el modelo de ejecución entre el procesador (cliente, en inglés host) y los dispositivos con los gpu (dispositivos, en inglés devices). Mediante un código escrito en C (estándar c99) se detalla el comportamiento del programa tanto en el cliente como en el/los dispositivo/s; el código correspondiente al comportamiento del dispositivo recibe el nombre de kernel.
2. **Ejecución**: define cómo se configura el entorno OpenCL en el host y cómo se ejecutan kernels en el dispositivo. Esto incluye configurar un Contexto OpenCL en el host, que proporciona mecanismos para la interacción entre el servidor y el dispositivo, y definir un modelo de simultaneidad utilizado para la ejecución del kernel en dispositivos.
3. **Memoria**: define la jerarquía de memoria abstracta que usan los kernels.
4. **Programa**: describe como es mapeado físicamente el paralelismo en el hardware.

A continuación ampliaremos esta división de trabajos realizados en la arquitectura para comprender mejor su funcionamiento:

Plataforma

La plataforma se refiere a la parte vinculada al hardware, cuenta con un dispositivo central (la computadora host con arquitectura x86 o similares) y los diferentes tipos de dis-

positivos acoplados al host (GPUs o DSPs). Cada dispositivo funciona con su contador de programa y su memoria independiente. En la figura 3.3 se observa el diseño de la plataforma.

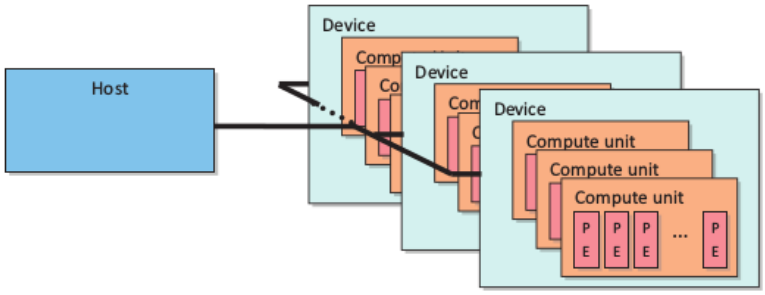

Figura 3.3: Plataforma

Ejecución

La ejecución hace referencia a cómo se contextualiza y ejecuta un programa en la arquitectura. Opencl provee un espacio para manejar sus recursos y objetos el cual recibe el nombre de contexto. Este se relaciona con los siguientes conceptos: dispositivos (las GPUs), el kernel (función que corre en el dispositivo, el código fuente del mismo), memoria tanto del host como de los dispositivos, la sincronización de los datos del programa y la implementación del programa de cómputo.

Los contextos disponen de una lista de dispositivos para ser asociados y el $host$ se comunica con ellos por medio de una cola de comandos (command queues). Cada dispositivo cuenta con su cola de comandos los cuales pueden estar sincronizados, o no, y ejecutarse en orden o no. Mediante dicha cola el host solicita al dispositivo realizar determinadas acciones en el dispositivo.

Memorias

El manejo de información se realiza por medio de las memorias disponibles, local y/o global, las cuales se vinculan al contexto antes de realizar la ejecución. Se clasifican en buffer (memoria contigua que puede leerse y escribirse) e imagenes accesible solo por funciones de imágenes vinculadas. La información será tranferida desde el $host$ al $divice$ al momento de realizar la ejecución del kernel. Como podemos ver en la figura 3.4 se definen cuatro tipos de memorias:

- Memoria global: accesible a todos los *work-items*.
- Memoria constante: solo lectura y global.
- Memoria local: local a el *work-group*.
- Memoria privada: privada a un *work-item*.

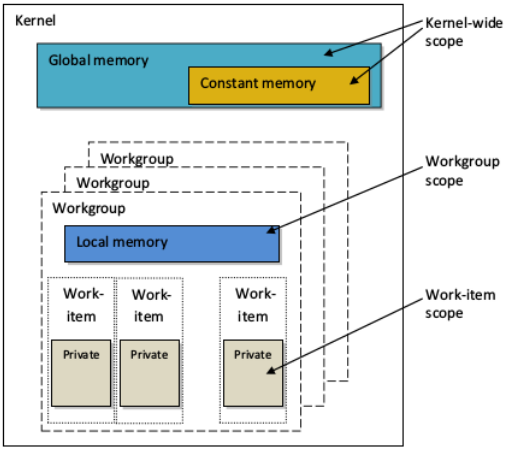

Figura 3.4: Memoria OpenCL

Programa

Un programa es una secuencia de instrucciones compiladas, o no, el cual puede contener constantes y funciones. Se requiere especificar el dispositivo asociado y los parámetros de compilación requeridos.

Ecuación de calor en OpenCL

El modelo de OpenCL cuenta con una gran cantidad de opciones de configuración, tanto para el host como para el device. A continuación, se describirán las principales características del mismo para dos versiones del Kernel, una basada en memoria global y otra basada en memoria local. En el apéndice A.1.2 se muestra la implementación en lenguaje C, la cual fue paralelizada a OpenCL.

En el algoritmo paralelo de la ecuación de calor todas las celdas de la matriz se procesan simultáneamente, asignadas cada una a un *work-item* (paralelismo de datos). La cantidad de celdas (y por tanto de *work-items*) depende del tamaño de la matriz a calcular. Todos los *work-items* deben pertenecer al mismo *workgroup*, ya que, es necesario sincronizarlos luego de cada iteración. Para una ejecución óptima, el número máximo de *Work-items* está determinado por la arquitectura de la GPU, en este caso 1024 *work-items*. Como se explicó en el capítulo 1 se decidió recorrer la matriz con la forma de un vector de una dimensión, por lo tanto, el espacio indexado debe ser de 1 dimensión. La paralelización del código propuesto, consiste en reemplazar el ciclo while que recorre la matriz en forma de vector del código en C [A.1.2] con paralelismo.

Las diferentes iteraciones son llevadas adelante en la misma instancia de kernel evitando el traslado de información entre el *host* y el *device* o entre *devices*; El manejo de datos en memoria se realizó en dos versiones, global y local, como puede verse a continuación:

Kernel global

```
__kernel void calculo_calor(
        __global float * current,
       __global bool * bordes,
       int N,
       int iteraciones)
{
    int p = get_global_id(0);
    while(iteraciones--){
        bordes[p] = bordes[p]?current[p]:
       (current[p+1]+current[p-1]+current[p+N]+current[p-N])*0.25;
        barrier(CLK_LOCAL_MEM_FENCE);
    }
}
```

Kernel local

```
__kernel void calculo_calor(
    __global float * matrix,
        __global bool * bordes,
        int iteraciones)
{
    __local float current[24*24];
    int p = get_global_id(0);
    current[p] = (float)matrix[p];
        barrier(CLK_LOCAL_MEM_FENCE);
    while(iteraciones--){
        current[p]  = bordes[p]?current[p]:
                (current[p+1]+current[p-1]+current[p+N]+current[p-N])*0.25;
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    matrix[p] = (float)current[p];
}
```

La versión final correspondiente a las mediciones fue utilizar el kernel local dado que su tiempo de cómputo es inferior al kernel con memoria global.

Mediciones

Para las mediciones se dispone de una PLATAFORMA GeForce GTX 980, la cual posee las siguientes características:

INFORMACION

- Max work items dimensions: 3
- Max work items[0]: 1024

- Max work items[1]: 1024
- Max work items[2]: 1024
- Max work group size: 1024

- Number of Platforms: 2
- Platform size: 8
- Platform Name: NVIDIA CUDA
- Platform Vendor: NVIDIA Corporation
- Platform Version: OpenCL 1.2 CUDA 9.1.84
- Platform Profile: FULL_PROFILE

- DEVICE INFORMATION:
- Number of Devices: 2
- Device size: 8
- Device Name: GeForce GTX 980
- Device Vendor NVIDIA Corporation
- Device Version: OpenCL 1.2 CUDA
- Device Profile FULL_PROFILE
- Compute Units 16
- Clock Frequency 1215
- Performance 19440

Mediciones Cálculo

Las pruebas se han realizado con el kernel local aplicando a los siguientes conjuntos de matrices: 8x8, 16x16, 24x24, 32x32 con iteraciones de 5,10,50,100,500,1000,5000 y 10000. La elección de la matriz 32x32 como valor mayor se debe a las características del GPU (1024 *work-items* disponibles), mientras que las iteraciones varían en un rango de pocas iteraciones a una cantidad considerable de casos.

Se obtuvieron los siguientes resultados expresados en milisegundos:

Tiempo de lectura y escritura:

	8	16	24	32
5	0.0036	0.0023	0.0027	0.003
10	0.0026	0.0027	0.0029	0.0029
50	0.0025	0.0026	0.0026	0.0029
100	0.0023	0.0023	0.0025	0.003
500	0.0023	0.0024	0.0025	0.0029
1000	0.0023	0.0023	0.0025	0.0029
5000	0.0023	0.0023	0.0025	0.0029
10000	0.0023	0.0023	0.0025	0.0029

Tabla 3.1: Tiempo de escritura en ms

	8	16	24	32
5	0.0013	0.0013	0.0014	0.0016
10	0.0013	0.0013	0.0014	0.0016
50	0.0013	0.0015	0.0014	0.0016
100	0.0013	0.0013	0.0014	0.0016
500	0.0013	0.0014	0.0014	0.0016
1000	0.0013	0.0013	0.0014	0.0016
5000	0.0013	0.0013	0.0014	0.0016
10000	0.0013	0.0013	0.0014	0.0015

Tabla 3.2: Tiempo de lectura en ms

	8	16	24	32
Lectura	0.0013	0.0013375	0.0014	0.0015875
Escritura	0.002525	0.0024	0.0025875	0.002925

Tabla 3.3: Tiempo medio de lectura/escritura

En la figura 3.5 podemos observar el tiempo medio de lectura/escritura gráficamente para las diferentes dimensiones de la matriz.

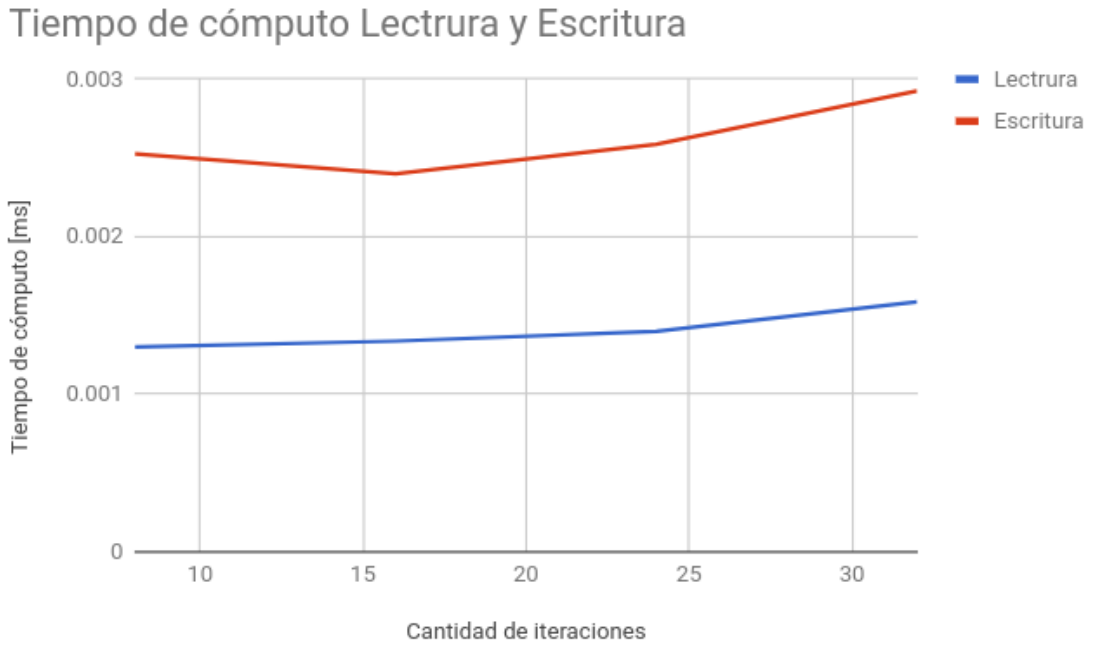

Figura 3.5: Tiempo medio lectura/escritura en ms

	8	16	24	32
5	0.0136	0.011	0.013	0.016
10	0.0131	0.014	0.017	0.022
50	0.032	0.035	0.056	0.079
100	0.055	0.065	0.104	0.146
500	0.24	0.28	0.485	0.696
1000	0.476	0.56	0.96	1.385
5000	2.34	2.77	4.77	6.88
10000	4.688	5.53	9.27	13.1

Tabla 3.4: Tiempo de cómputo kernel

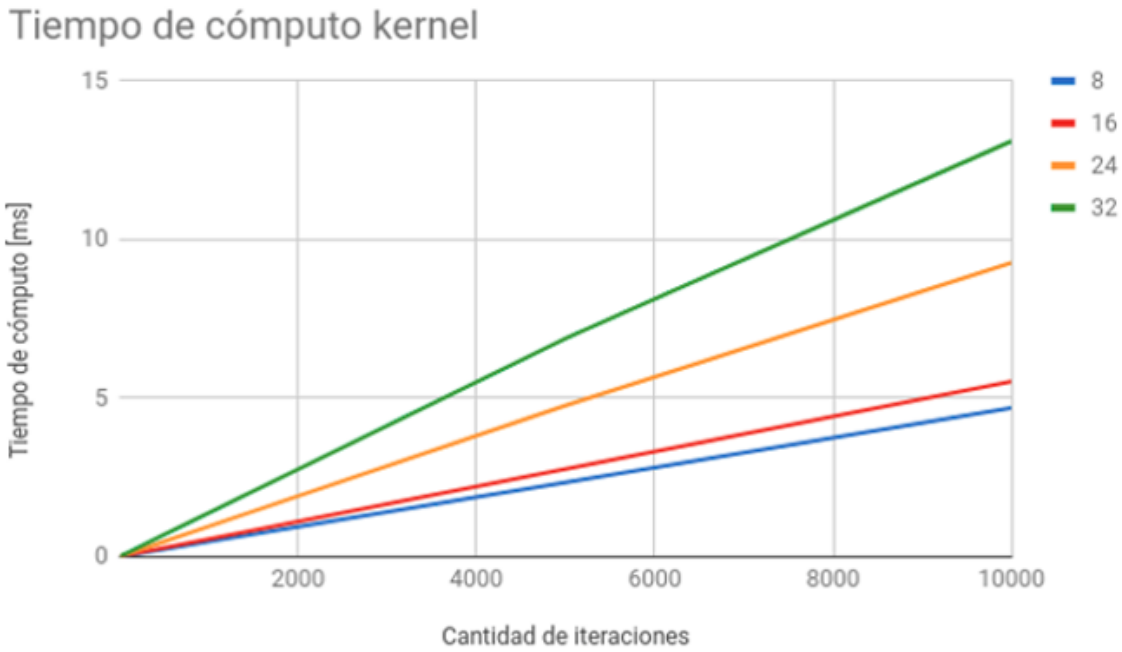

Figura 3.6: Tiempo de cómputo kernel en ms

Tiempo total de cómputo (kernel+lectura+escritura):

	8	16	24	32
5	0.0185	0.0146	0.0171	0.0206
10	0.017	0.018	0.0213	0.0265
50	0.0358	0.0391	0.06	0.0835
100	0.0586	0.0686	0.1079	0.1506
500	0.2436	0.2838	0.4889	0.7005
1000	0.4796	0.5636	0.9639	1.3895
5000	2.3436	2.7736	4.7739	6.8845
10000	4.6916	5.5336	9.2739	13.1044

Tabla 3.5: Tiempo total de cómputo (kernel+lectura+escritura)

En la figura 3.7 se observa el resultado interpretado gráficamente, las iteraciones en función del tiempo por cada matriz propuesta NxN:

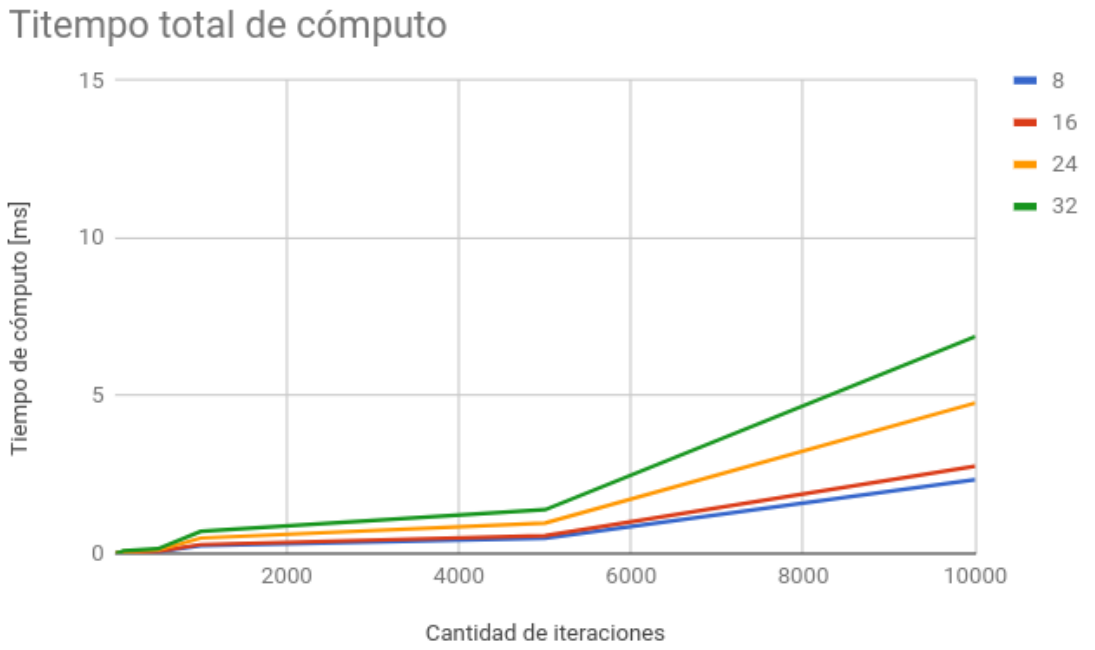

Figura 3.7: Tiempo total de cómputo en ms

Consumo de potencia

Las mediciones de potencia se realizaron en el cálculo de 10.000 iteraciones, utilizando matrices dimensionadas en: 32x32, 24x24, 16x16 y 8x8. Los resultados se observan en la tabla 3.6.

Matriz	Potencia (W)
32	62.02
24	59.9
16	57.87
8	57.26

Tabla 3.6: Consumo de potencia en ms

La potencia inicial cuando la GPU no está trabajando es de 22.5W, es decir, que la potencia se incrementa aproximadamente un 260 % al ejecutar el algoritmo de cálculo de calor.

Conclusiones

- El tiempo de cómputo se ve incrementado por la dimensión de la matriz de forma no proporcional.

- El tiempo de lectura y escritura no depende de la cantidad de iteraciones y depende levemente del tamaño de la matriz.
- El consumo de potencia aumenta un 8 % entre la matriz de 8x8 y 32x32 celdas.
- El tamaño máximo de la matriz que se logró obtener es de 32x32 celdas.

4 FPGA basada en sistemas con OpenCL

Los FPGA son una solución rentable en comparación con los ASIC (Circuitos Integrados de Aplicación Específica). Como vimos en el capítulo 3, los lenguajes de descripción de hardware se han utilizado para programar FPGA. Para mejorar el rendimiento, el programador debe tener un elevado grado de conocimiento y experiencia sobre diseño de hardware. Además, los diseños basados en HDL requieren simulaciones en varios niveles. Como resultado, el tiempo de diseño es muy grande. Una alternativa creciente planteada para resolver estos problemas ha sido introducida recientemente, la cual se ocupa de la resolución de problemas mediante OpenCL y FPGA.

Sistemas heterogéneos con FPGA

En la figura 4.1 se muestran dos tipos de sistemas informáticos heterogéneos basados en CPU y FPGAs. La Figura 4.1(a) muestra un sistema informático de tipo SoC (sistema en chip, system on chip), donde una CPU y un FPGA están integrados en el mismo encapsulado. La transferencia de datos entre la CPU y el FPGA se realiza a través de un bus interno (en chip). Esta arquitectura es la que se utilizará en el presente trabajo. La Figura(b) muestra otro tipo de sistemas de computación con CPU y un FPGA. En este sistema, la FPGA está conectado a la CPU a través de un bus externo como PCI-express (PCIe). La transferencia de datos entre la CPU y el FPGA se realiza a través de dicho bus. Este tipo de sistemas generalmente se usan en aplicaciones de alto rendimiento .

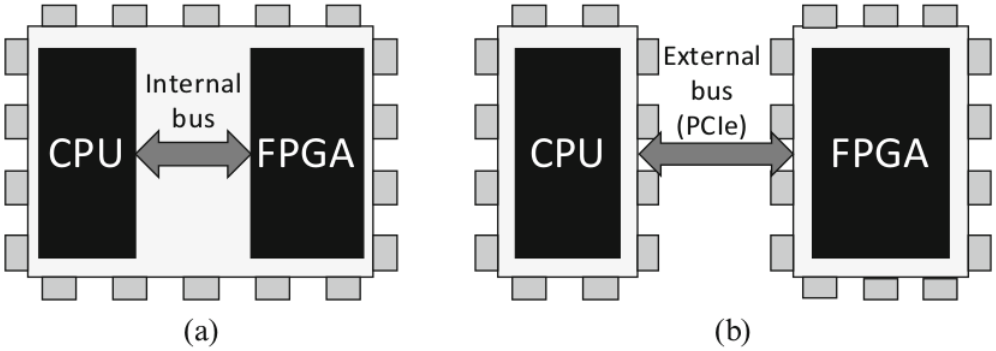

Figura 4.1: Sistemas heterogeneos

Hay muchas placas FPGA diferentes con diferentes entradas/salidas (E/S), memoria externa, PCI Express, USB, etc. Además, se pueden encontar FPGA diseñadas para diferentes propósitos, como los de aplicaciones de baja potencia y computación de alto rendimiento. Los recursos de lógica y memoria de esos FPGA son significativamente diferentes. Como resultado, generalmente es difícil usar el mismo código en diferentes placas FPGA, sin conocer las E/S, lógica y recursos de memoria disponible. La Figura 4.2 muestra cómo se resuelve este problema.

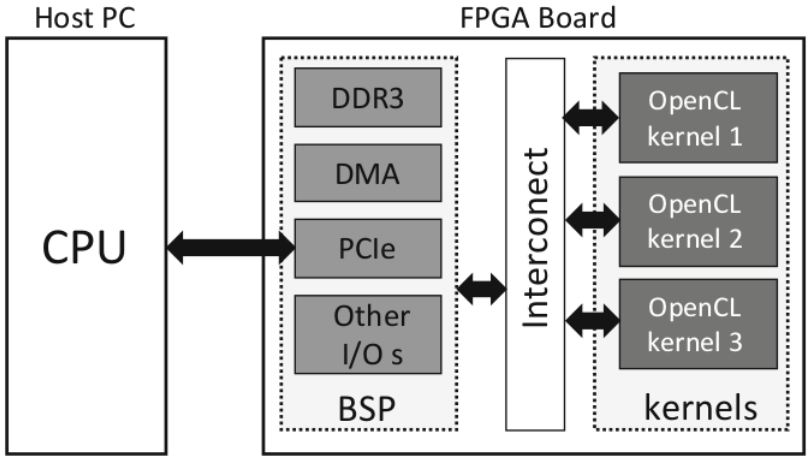

Figura 4.2

OpenCL para FPGA utiliza una paquete de soporte de placa (BSP) que contiene información de los bloques lógicos, memoria y controladores de E/S (DDR3, PCI, etc). Durante la compilación se utiliza la información contenida en el BSP y se las combina con los kernels. Como resultado, los kernel puede acceder a las E/S a través del BSP. Dado que el BSP se ocupa de las E/S y los recursos de la FPGA, el mismo código de kernel se puede utilizar en diferentes placa FPGA compatibles con OpenCL.

HDL vs. OpenCL

Es posible mencionar las siguientes ventajas y desventajas de opencl para FPGA:

Ventajas

- Diseño con lenguaje C: esto nos brinda una aceleración en cuanto al diseño.
- Soporte para entradas/salidas: no se requiere configurar los puertos de entrada ni de salida.
- Compatibilidad y re-usabilidad de diferentes tipos de FPGA: por medio de la descripción de BSP es posible compilar nuestro código a diferentes tipos de placas ajustando los requerimientos al tipo de placa.
- Fácil de debuggear: se utilizan métodos de debugeo de alto nivel.

Desventajas

- La arquitectura está oculta para el programador: el compilador convierte OpenCL en HDL con lo cual el programador no tiene forma de conocer la arquitectura implementada.
- No se puede diseñar para una frecuencia de reloj especifica: el compilador determina automáticamente la frecuencia.
- Dificultad de controlar los recursos utilizados: el compilador busca la mayor performancia por lo que puede utilizar demasiados recursos para un problema sencillo.

FPGA orientado a la programación en paralelo

OpenCL para FPGA utiliza dos tipos de kernels: kernel NDRange y el kernel Single-Work-Item. El kernel NDRange tiene muchos work-items mientras que el kernel Single-Work-Item tiene solo un work-item. La primera diferencia entre ellos está en los estilos de programación, los kernels NDRange usan un estilo de programación similar a GPU, donde se procesan múltiples elementos de trabajo en paralelo usando identificadores de elementos de trabajo globales y locales. Por otro lado, los kernels Single-Work-Item usan una CPU donde solo se procesa un work-item en todo el kernel. La segunda diferencia está en los métodos de intercambio de datos. Los kernel en NDRange comparten datos entre múltiples work-items utilizando memoria local, mientras que el kernel Single-Work-Item comparten datos entre múltiples iteraciones mediante el uso de una memoria privada.

Kernel NDRange

Un kernel NDRange es ejecutado por múltiples work-items, los cuales son son lanzados uno tras otro separados por un ciclo de reloj y son procesados en forma de pipeline. El modo de ejecución es bastante diferente al de las GPU, aunque el código kernel NDRange es muy similar a una GPU. En las GPU, los work-items se lanzan simultáneamente y se ejecutan de una sola vez, es decir, múltiples work-items se ejecutan simultáneamente para diferentes datos. Por otro lado, en FPGAs, los work-items se inician y se ejecutan uno tras otro de forma pipeline. El comportamiento del kernel NDRange es similar a múltiples instrucciones, múltiples datos(MIMD), y muy diferente al de la GPU.

Cuando hay dependencia de datos en los work-items, los datos deben ser compartidos a través de memoria local o global. Es necesario utilizar barreras de sincronización para asegurar el orden de ejecución de instrucción correcto. Todos los work-items dentro de un work-group deben haber alcanzado este punto de sincronización antes de continuar con la ejecución del programa. Las FPGA usan pipelines y la penalidad por usar barreras de sincronización en los work-items es grande a diferencia de los GPU.

Cuando existe dependencias de datos, la metodología single-work-item es más sencilla de utilizar y probablemente se obtenga mejor performancia que el kernel NDRange.

Loop Unrolling

Loop Unrolling es un método para incrementar el paralelismo reduciendo el número de iteraciones a expensas de incrementar la utilización de los recursos. Para su utilización debe considerarse:

El loop unrolling no solo incrementa la utilización de recursos, también requiere mayor ancho de banda de memoria. La utilización de factores altos de unrolling reduce la frecuencia de reloj. Cuando existen loops anidados, aplicar esta metodología al loop externo puede incrementar la utilización de recursos significativamente. Por lo tanto es recomendable primero aplicar al loop interno.

Descripción general de la memoria

OpenCL para FPGA, al igual que para las GPU, usa una estructura de memoria jerárquica. La memoria global es una DRAM ubicada fuera del chip FPGA. La memoria constante ocupa una porción de la DRAM que se usa como memoria global. Los datos de la memoria constantes están almacenados en la memoria, en el chip FPGA, como así también, los recursos privados y locales. La memoria local se implementa utilizando Bloques de RAM y la memoria privada se implementa utilizando tanto bloques RAM como registros.

La la figura 4.3 muestra la ruta de datos entre la memoria global y un kernel. El kernel ac-

cede a la memoria global a través de un controlador de memoria. Un controlador de memoria proporciona una interfaz simple para el kernel mientras se ocupa del complicado control de la memoria global.

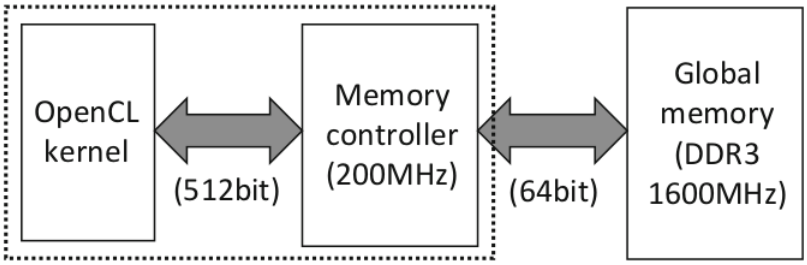

Figura 4.3: Tipos de implementacion memoria

Mediciones

Para las mediciones se dispone de una placa DE1-SoC integrada con una FPGA Cyclone V SoC 5CSEMA5F31C6. Las características de la plataforma pueden verse a continuación:

INFORMACIÓN

- Number of Platforms: 1
- Platform size: 4
- Platform Name: Altera SDK for OpenCL
- Platform Vendor: Altera Corporation
- Platform Version: OpenCL 1.0 Altera SDK for OpenCL, Version 16.0
- Platform Profile: EMBEDDED_PROFILE

DEVICE INFORMATION:

- Number of Devices: 1
- Device size: 4
- Device Name: de1soc_sharedonlyCyclone V SoC Development Kit
- Device Vendor Altera Corporation
- Device Version: OpenCL 1.0 Altera SDK for OpenCL, Version 16.0
- Device Profile EMBEDDED_PROFILE

- Compute Units 1
- Clock Frequency 1000
- Performance 1000

Se utilizó el kernel local propuesto en el capítulo 3 y las pruebas se han realizado sobre el siguiente conjunto de matrices: 8x8, 16x16, 24x24, 32x32, 40x40 y 44x44 con iteraciones de 5,10,50,100,500,1000,5000 y 10000.

Mediciones Cálculo

Se obtuvieron los siguientes resultados expresados en milisegundos:

	8	16	24	32	40	44
5	0.006	0.017	0.036	0.062	0.104	0.123
10	0.006	0.017	0.037	0.062	0.102	0.124
50	0.006	0.018	0.044	0.063	0.104	0.121
100	0.006	0.018	0.036	0.07	0.104	0.121
500	0.006	0.017	0.036	0.071	0.104	0.122
1000	0.006	0.017	0.036	0.07	0.097	0.123
5000	0.006	0.017	0.036	0.07	0.105	0.124
10000	0.006	0.017	0.037	0.071	0.104	0.123
	0.006	0.01725	0.03725	0.067375	0.103	0.122625

Tabla 4.1: Tiempo de lectura

	8	16	24	32	40	44
5	0.008	0.014	0.026	0.039	0.061	0.08
10	0.008	0.014	0.025	0.048	0.061	0.072
50	0.008	0.015	0.025	0.04	0.069	0.072
100	0.008	0.015	0.026	0.04	0.068	0.072
500	0.007	0.015	0.025	0.04	0.061	0.72
1000	0.008	0.015	0.025	0.04	0.06	0.071
5000	0.008	0.014	0.025	0.04	0.06	0.071
10000	0.008	0.014	0.026	0.039	0.061	0.071

Tabla 4.2: Tiempo de escritura

	8	16	24	32	40	44
Lectura	0.006	0.01725	0.03725	0.067375	0.103	0.122625
Escritura	0.007875	0.0145	0.025375	0.04075	0.062625	0.153625

Tabla 4.3: Tiempo medio Lectura - Escritura

En la figura 4.4 se observa el tiempo medio de lectura/escritura gráficamente para las diferentes dimensiones de la matriz:

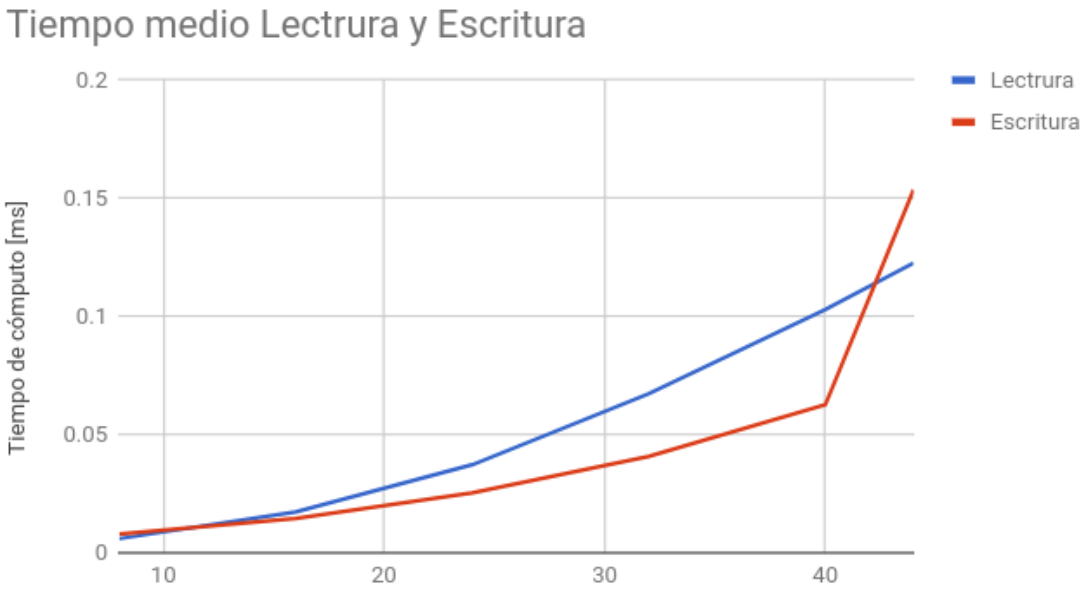

Figura 4.4: Tiempo medio de lectura/escritura

Tiempo de kernel:

	8	16	24	32	40	44
5	0.1	0.103	0.117	0.14	0.167	0.19
10	0.1	0.112	0.143	0.183	0.247	0.279
50	0.131	0.203	0.343	0.515	0.708	0.857
100	0.17	0.34	0.557	0.881	1.307	1.551
500	0.507	1.198	2.366	3.976	6.038	7.24
1000	0.904	2.283	4.595	7.7811	11.966	14.336
5000	4.16	11.007	22.492	38.459	59.074	71.079
10000	8.197	21.91	44.851	76.777	118.012	142.027

Tabla 4.4: Tiempo de kernel

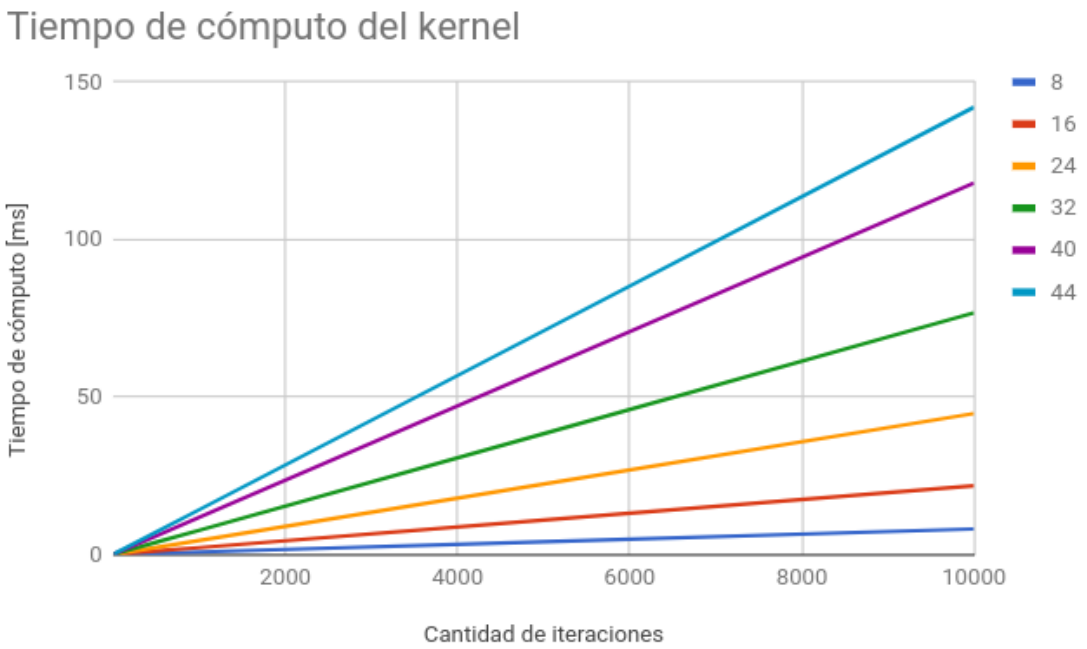

Figura 4.5: Tiempo de kernel

Tiempo total de computo (kernel + lectura + escritura)

	8	16	24	32	40	44
5	0.114	0.134	0.179	0.241	0.332	0.393
10	0.114	0.143	0.205	0.293	0.41	0.475
50	0.145	0.236	0.412	0.618	0.881	1.05
100	0.184	0.373	0.619	0.991	1.479	1.744
500	0.52	1.23	2.427	4.087	6.203	8.082
1000	0.918	2.315	4.656	7.8911	12.123	14.53
5000	4.174	11.038	22.553	38.569	59.239	71.274
10000	8.211	21.941	44.914	76.887	118.177	142.221

Tabla 4.5: Tiempo total de computo

En la figura 4.6 se observa el resultado interpretado gráficamente, las iteraciones en función del tiempo por cada matriz propuesta NxN:

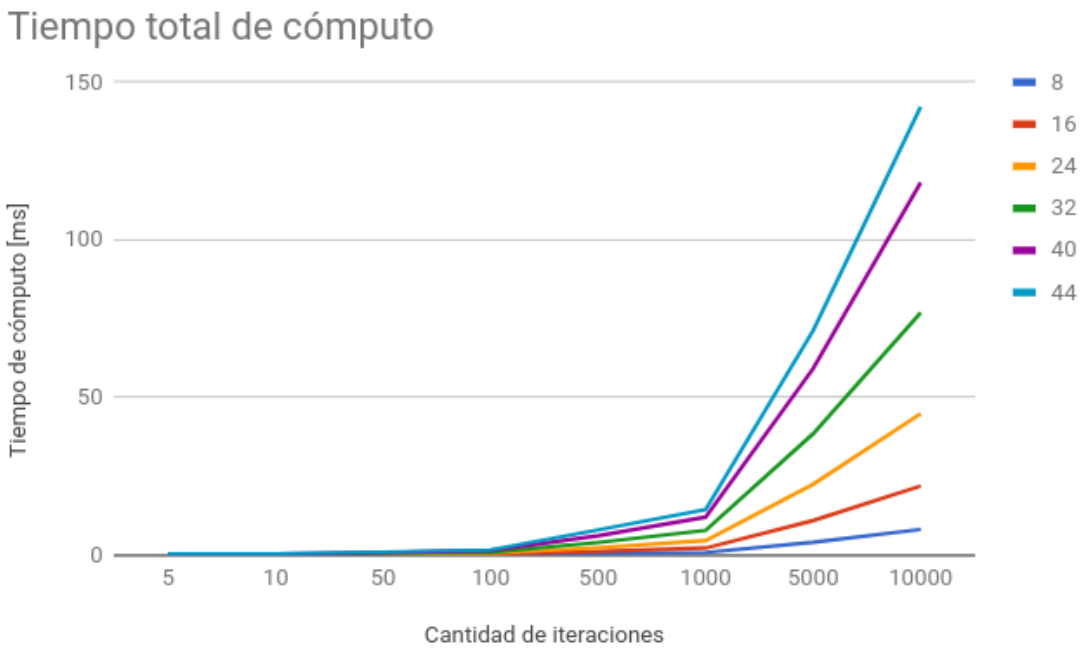

Figura 4.6: Tiempo de kernel

Consumo de potencia

En la tabla 4.6 se muestra el consumo de potencia de la FPGA para los diferentes tamaños de matriz, estos fueron calculados mediante la herramienta Altera, para obtener los valores se realizó la simulación post-implementación para obtener la actividad de switcheo de todos los transistores de la FPGA.

Matriz	Potencia (W)
44	1.03
32	1.06
24	1.05
16	0.8
8	0.74

Tabla 4.6: Consumo de potencia

Conclusiones

- El tiempo de cómputo se observa muy dependiente de la dimensión de la matriz y la cantidad de iteraciones.
- Los tiempos de escritura/lectura aumentan respecto al tamaño de la matriz.
- El consumo de potencia aumenta un 40 % entre la matriz de 8x8 y 32x32 celdas.

- El tamaño máximo de la matriz que se logró obtener es de 44x44 celdas.

Conclusiones

Para cerrar el presente trabajo se realiza una comparativa de los diferentes resultados obtenidos en cada capítulo, se analiza los tiempos tanto de lectura/escritura, tiempo total de procesamiento y potencia consumida, superponiendo en una gráfica los tiempos obtenidos en las diferentes arquitecturas (FPGA, GPU y OPENCL-FPGA). Para las comparaciones se tomarán las matrices 8x8, 16x16, 24x24 y 32x32; para el tiempo total de procesamiento se analizará la matriz 32x32 común a todas las arquitecturas.

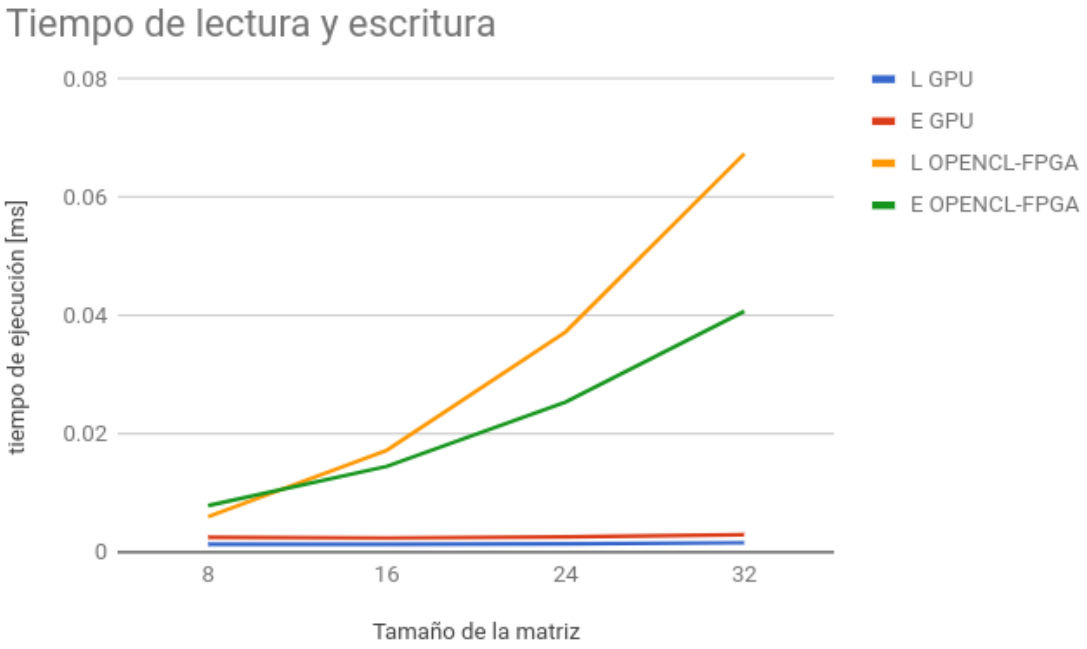

Figura 4.7: Tiempo de lectura y escritura

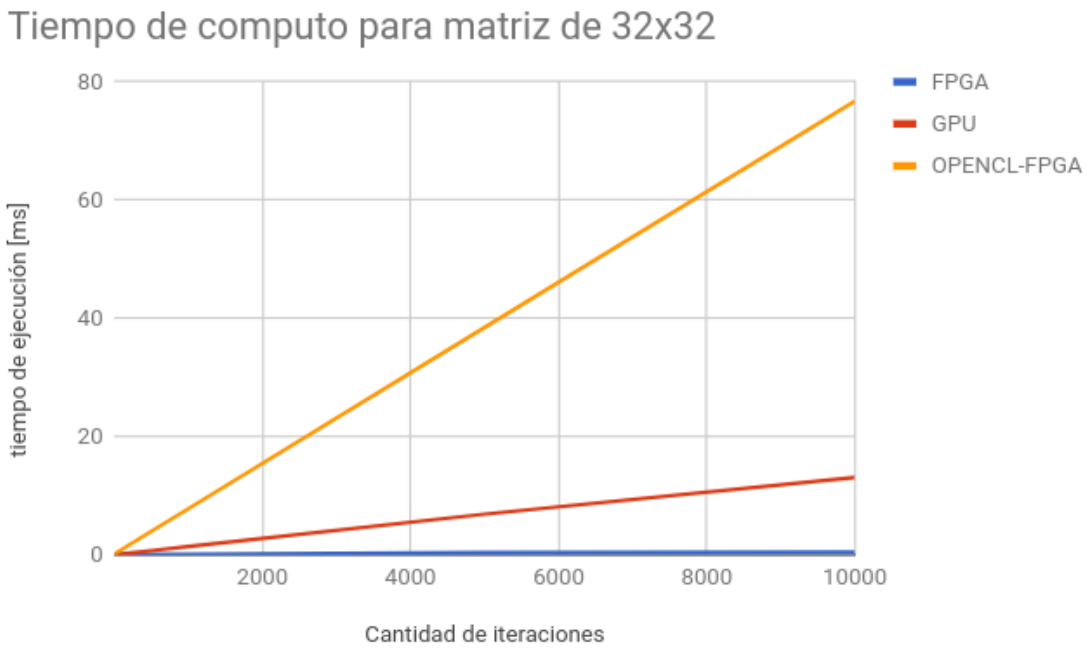

Figura 4.8: Tiempo total de procesamiento matriz 32x32

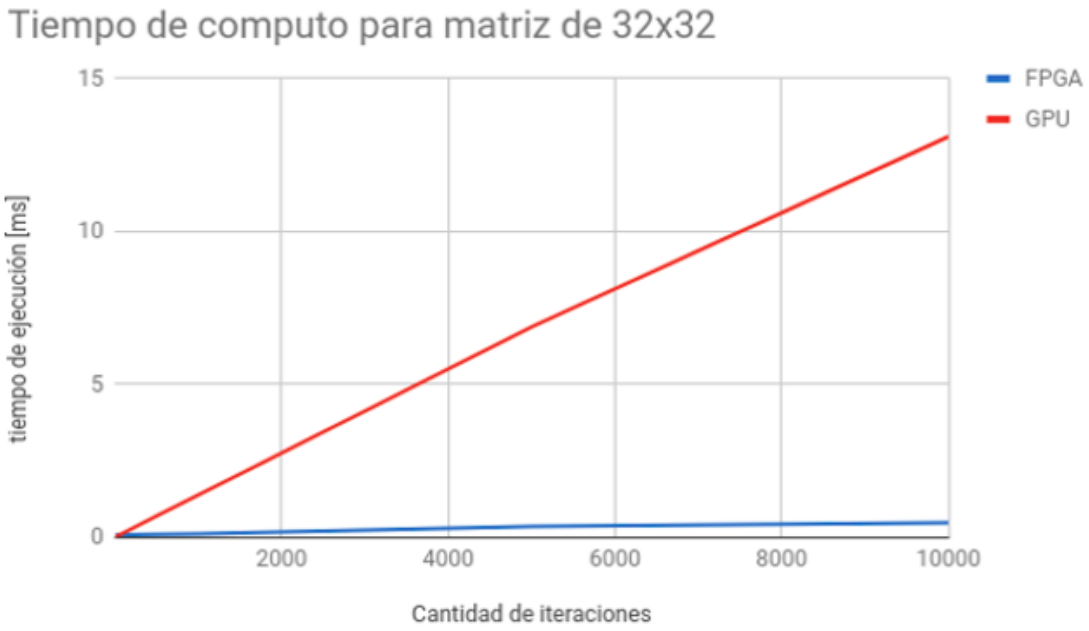

Figura 4.9: Tiempo total de procesamiento matriz 32x32. FPGA - GPU

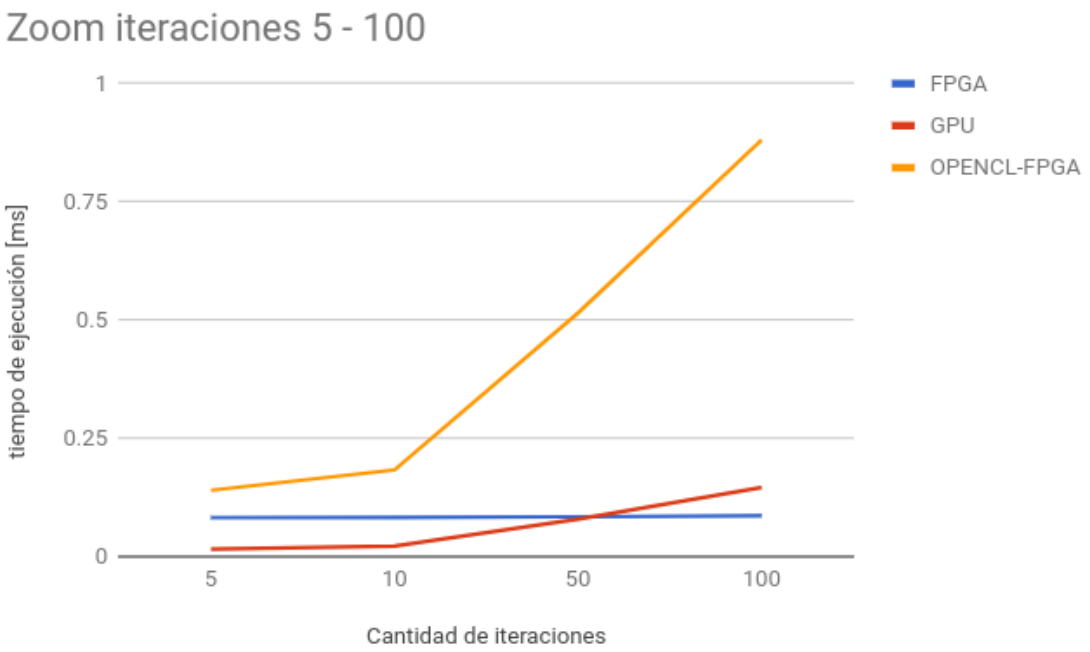

Figura 4.10: Zoom iteracion 5-100

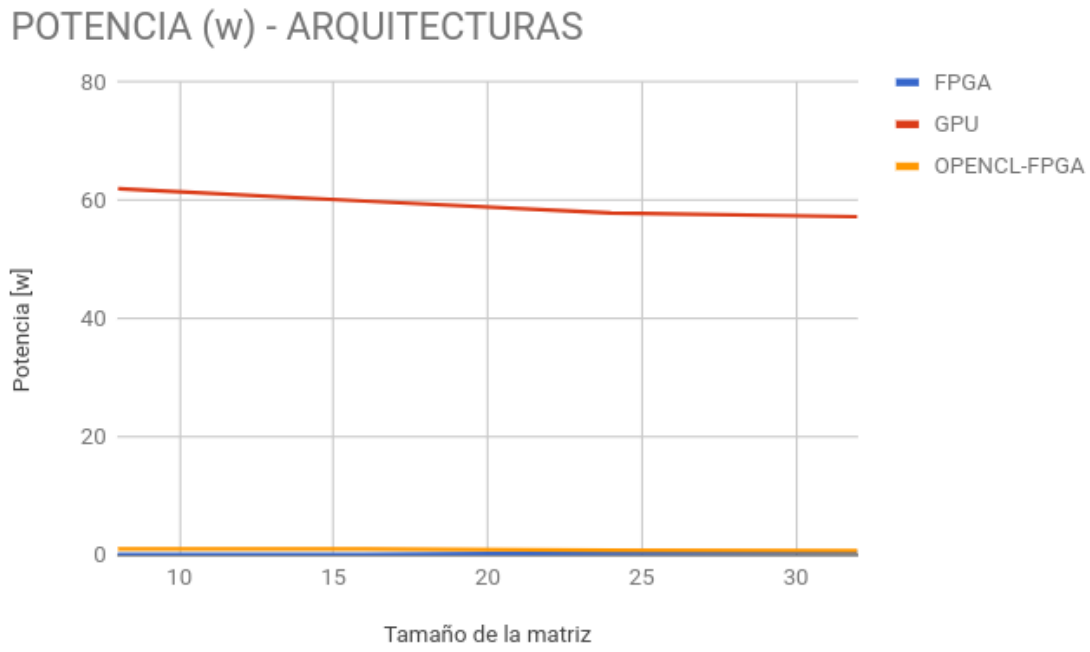

Figura 4.11: Consumo de potencia

Comparación de plataformas

- Para una cantidad muy baja de iteraciones la GPU computa en menor tiempo que la FPGA.
- El tiempo de cómputo de las FPGA es significativamente inferior, al de OpenCL tanto en la GPU como en la FPGA, para gran cantidad de iteraciones. El nivel sobre el que se resuelve el problema en las FPGA es una solución a medida (un hardware específico) y esto se ve claramente reflejado en la velocidad de procesamiento.
- El consumo de potencia en la GPU es notablemente superior al de las FPGA y su proporción en relación a la dimensión de la matriz se ve más afectado en las GPU (un orden de 2 a 3 watt de crecimiento, mientras que en la FPGA no supera 1w) .
- Tamaño total de la matriz a implementar en la FPGA y la GPU fue de 32x32 celdas, sin embargo, en la FPGA utilizando openCL se logró implementar matrices de mayor tamaño

Conclusión general

A lo largo del trabajo se analizaron tres metodologías de trabajo distintas para resolver la llamada ecuación de calor de Laplace computacionalmente. La primer implementación fue realizada en VHDL sobre una FPGA de gama baja obteniendo un bajo consumo con bajo tiempo de ejecución, sin embargo, se requirió dedicar la mayor parte del desarrollo del trabajo para lograrlo. Más aún, considerando que el desarrollador es de la rama computacional y no electrónica. Luego, la implementación llevada adelante en OpenCL obtuvo tiempo de ejecución mayores a los ya obtenido en la FPGA, sin embargo, el tiempo de desarrollo y la complejidad fueron insignificantes frente al desarrollo en HDL para la FPGA. Dadas las amplias diferencias entre ambas metodologías se decidió utilizar la reciente implementación de OpenCL a la FPGA, la cual, intenta obtener las ventajas de las dos anteriores. Sin embargo, aunque la potencia se mantuvo en el orden de magnitud de la primer implementación, los tiempos de ejecución aumentaron significativamente. La razón de esto se debe a:

- Esta metodología de trabajo no cuenta con la trayectoria y el desarrollo que si presentan las otras tecnologías impactando en la cantidad de documentación disponible y características necesarias que aún no se han implementado [10].
- La placa de desarrollo utilizada no está diseñada para esta tarea sino para desarrollos de bajo consumo de potencia.
- Se requiere de una gran comprensión de cómo la herramienta compila y sintetiza el diseño, ya que, es sencillo escribir el código OpenCL, sin embargo, la optimización para alta performance es difícil. Es por esto que se considera necesario contar con mayor tiempo de desarrollo y experiencia.

Dado que las FPGA se están volviendo cada vez más grandes, se torna difícil diseñar un código en el cual se utilice eficientemente todos los recursos de la FPGA en un tiempo de diseño limitado. Por lo tanto si consideramos el tiempo de diseño como una limitante, la performance de los diseños basados en OpenCL (para las FPGA) van a dominar los diseños basados en HDL. Incluso hoy, la performance de estos diseños no están muy atrasados respecto aquellos basados en HDL [10].

Trabajo futuro

Se recomienda como trabajo futuro la implementación donde el tamaño de la matriz no esté limitada por el dispositivo, y además, continuar con optimizaciones del código OpenCL sobre las FPGA.

Experiencia personal del autor

El trabajo ha servido para generar una fuerte experiencia desde un punto de vista electrónico y computacional. Algunas apreciaciones personales que percibí durante la elaboración del trabajo son opiniones y no se tratan de conclusiones técnicas, sino de una experiencia que puede sumar al contexto:

- No creo que alguna arquitectura reemplazará a otra, sino que se complementarán; las fpga no vendrán para reemplazar ni al cpu ni a la gpu sino más bien complementar su potencia como lo fue en su momento la gpu y cpu.
- La FPGA es una tecnología con mucho desarrollo por delante que requiere involucrar diferentes diciplinas, tanto a ingenieros electrónicos como licenciados en computación.
- Los estándares en los lenguajes de modelados HDL no son del todo rigurosos haciendo que cada empresa opte por su forma de implementación.
- La sintaxis de los lenguajes HDL está muy sobrecargada.
- Los sistemas heterogéneos son muy recientes y se espera obtener mejores resultados con el tiempo.
- No se cuenta con las ventajas de generar IDEs abiertos; Las empresas generar demasiado esfuerzo en generar todo un paquete propio, pero como entorno de desarrollo estos no logran evolucionar como los de un lenguaje, en el cual una comunidad genera ideas que mejoran la eficiencia y tiempo de desarrollo.
- Para quienes provienen del desarrollo computacional se genera una dificultad mayor sobre el desarrollo en FPGA dado que no se cuenta con muchos conocimientos digitales/electrónicos para mejorar optimizaciones, y otros factores, que hacen a un mejor desarrollo.

- Para la resolución de matrices 1000x1000 en FPGA implica un mayor desarrollo comparativamente al de la GPU; y los sistemas híbridos aún no generar la suficiente velocidad de cálculo para superar o igualar la GPU.

A Apendice

Algoritmo ecuación de calor

Python

Algoritmo ecuación de calor realizado en lenguaje python (versión reducida).

```
1  import grafico as gr
2
3  # Parametros de la simulacion
4  N             = 25
5  TEMP_BORDE    = 10.0
6  TEMP_INICIAL= 0.0
7  TEMP_FUENTE = 10.0
8  FUENTE_X      = 0
9  FUENTE_Y      = 0
10 ITERACIONES = 10
11
12 # Posicion de la fuente de calor y temperatura inicial en la matrix
13 def iniciar(matrix):
14
15     # Temepratura inicial
16     bordes=[False for i in range(N*N)]
17
18     # Posicion fuente de calor
19     matrix[FUENTE_X+FUENTE_Y*N] = TEMP_FUENTE
20     bordes[FUENTE_X+FUENTE_Y*N] = True
21
22     # Bordes
23     for f in range(N) :
24         for c in range(N) :
25             if f==0 or c==0 or c==N-1 or f==N-1:
26                 p = f*N+c
27                 bordes[p] = True
28                 matrix[p] = TEMP_BORDE
29
30     return bordes
```

```
31
32 # Calculo de una iteracion
33 def calcular(current,bordes) :
34     x=N;next=[0 for i in range(N*N)]
35     for p in range(N*N) :
36         if bordes[p] :
37             next[p] = current[p]
38         else :
39             next[p] = (current[p+1]+current[p-1]+current[p+N]+current[p-N])*0.25
40     return next
41
42 # valores iniciales
43 current = [TEMP_INICIAL for i in range(N*N)]
44
45 bordes = iniciar(current)
46
47 # Calculo de las iteraciones
48 for it in range(ITERACIONES) :
49     current = calcular(current,bordes)
50
51 gr.graficar(current,N)
52
53 #print(current)
54
55 print("")
```

C

Algoritmo ecuación de calor realizado en lenguaje C (versión reducida).

```
1  #define _GNU_SOURCE
2  #include <math.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <time.h>
7  #include <sys/time.h>
8  #include <stdbool.h>
9  #include "netpbm.h"
10
11 // Parametros de simulacion
12
13 static int N                    = 5;
14 static const float TEMP_BORDE   = 0.0f;
15 static const float TEMP_INICIAL = 0.0f;
16 static const float TEMP_FUENTE  = 12.0f;
17 static const float TEMP_AMBIENTE= 8.0f;
18 static const int FUENTE_X       = 2;
19 static const int FUENTE_Y       = 2;
20 static const int ITERACIONES    = 50;
21
```

```
22 static void inicializar(float * matrix,bool * bordes) {
23     // init
24     int p,f,c;
25 
26     // inicializamos las matrices
27     memset(matrix, TEMP_AMBIENTE, N * N * sizeof(float));
28     memset(bordes, false, N * N * sizeof(bool));
29 
30     // colocamos la fuente de calor
31     matrix[FUENTE_X+FUENTE_Y*N] = TEMP_FUENTE;
32     bordes[FUENTE_X+FUENTE_Y*N] = true;
33 
34     // seteamos los bordes
35     for(f = 0; f < N; ++f) {
36         for(c = 0; c < N; ++c) {
37             if (f==0||c==0||c==N-1||f==N-1){
38                 p = f*N+c;
39                 bordes[p] = true;
40                 matrix[p] = TEMP_BORDE;
41             }
42         }
43     }
44 
45 }
46 
47 // Calcula un paso en la matriz de calor
48 static void calcular(const float * current, float * next, bool * bordes) {
49     int p = N * N;
50     while(p--){
51         next[p] = bordes[p]?current[p]:(current[p+1]+current[p-1]+current[p+N]+current[
              p-N])*0.25;
52     }
53 }
54 
55 
56 int main() {
57 
58     int it = ITERACIONES;
59     size_t array_size = N * N * sizeof(float);
60     float * matriz = malloc(array_size);
61     float * next = malloc(array_size);
62     bool * bordes = malloc(N * N * sizeof(bool));
63 
64     inicializar(matriz,bordes);
65 
66     while(it--){
67         calcular(matriz, next,bordes);
68         float * swap = current;
69         matriz = next;
70         next = swap;
71     }
72 
```

```
73      free(matriz);
74      free(next);
75      free(bordes);
76
77      return 0;
78  }
```

Bibliografía

[1] McGinnis III, F. K., & Holman, J. P. (1969) Individual droplet heat-transfer rates for splattering on hot surfaces. International Journal of heat and mass transfer, 12(1), 95-108.

[2] Waidyasooriya, H. M., Hariyama, M., & Uchiyama, K. (2018). Design of FPGA-Based Computing Systems with OpenCL. Springer.

[3] Gaster, B., Howes, L., Kaeli, D. R., Mistry, P., & Schaa, D. (2012). Heterogeneous Computing with OpenCL: Revised OpenCL 1. Newnes.

[4] Pedroni, V. A. (2010). Circuit design and simulation with VHDL. MIT Press.

[5] Nurvitadhi, E., Venkatesh, G., Sim, J., Marr, D., Huang, R., Ong Gee Hock, J., ... & Boudoukh, G. (2017, February). Can FPGAs beat GPUs in accelerating next-generation deep neural networks?. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (pp. 5-14). ACM.

[6] Khronos Group, Open Computing Language (OpenCL),http://www.khronos.org/opencl

[7] Using Intel OpenCL with DE1-SoC

ftp://ftp.altera.com/up/pub/Intel_Material/16.1/Tutorials/OpenCL.pdf

[8] GTX, N. G. (2014). 980 Whitepaper. NVIDIA Corporation.

[9] Waidyasooriya, H. M., Takei, Y., Tatsumi, S., & Hariyama, M. (2017). OpenCL-based FPGA-platform for stencil computation and its optimization methodology. IEEE Transactions on Parallel and Distributed Systems, 28(5), 1390-1402.

[10] Hill, K., Craciun, S., George, A., & Lam, H. (2015, July). Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA. In Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on (pp. 189-193). IEEE.