Trabajo de Laboratorio 2:

Programación paralela en OpenCL

Objetivos

- Plantear diversos problemas reales de forma concurrente y paralela.
- Diseñar programas basados en los modelos de plataforma, memoria, programación y ejecución del paradigma OpenCL.
- Desarrollar códigos y algoritmos capaces de explotar los beneficios del planteo concurrente dentro de un mismo dispositivo, y dentro de un sistema heterogéneo.
- Ejercitar los contenidos teóricos brindados en clases a encontrar en la bibliografía de la materia.[1]

Condiciones

- Realizar el trabajo práctico en grupos de **2 ó 3 personas**.
- Subir el trabajo resuelto a la carpeta de Moodle “EntregaLab2”. La fecha límite de entrega es el **miércoles 17 de noviembre** (inclusive). Los trabajos entregados después de esa fecha se consideran desaprobados.
- Quienes estén en condiciones de promocionar, deben defender en un coloquio el trabajo presentado. Deben presentarse todos los integrantes del grupo que cumplan con las condiciones y responder preguntas acerca del trabajo realizado. El coloquio es el **viernes 19 de noviembre**. Ese día deben conectarse a las 9hs al Meet del práctico y allí coordinaremos el orden de las presentaciones. En caso de no poder asistir al coloquio en esa fecha, coordinar **previamente** con los docentes.
- En caso de cambiar su condición a “promocionada/o” luego de rendir los recuperatorios, deberán defender el trabajo el **viernes 26 de noviembre**.

Formato de entrega

Se debe entregar un archivo comprimido cuyo nombre debe seguir el siguiente formato “Lab2_ApellidoNombre1_ApellidoNombre2.tar.gz”. El cual deberá incluir tres *jupyter notebook,* por cada uno de los ejercicios resueltos y una exportación de cada uno de ellos a pdf. Además, las personas del grupo deben encargarse de tener en su cuenta de colab todos los notebooks disponibles para ejecutarse. Los grupos **no** pueden compartir código entre sí. Se verificará la similitud de los códigos entre grupos.

Todas las decisiones de diseño deben justificarse mediante celdas de texto en el mismo *notebook.*

[1] "Heterogeneous Computing with OpenCL", By Benedict R. Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry & Dana Schaa

Calificación

El ejercicio 1 es obligatorio. Su resolución debe estar aprobada para obtener la regularidad de la materia. Quienes además resuelvan el ejercicio 2 estarán habilitados para promocionar, si cumplen con los demás requisitos. Quienes resuelvan el ejercicio 3 (no es necesario que hayan hecho el 2) obtendrán 1 punto extra para el segundo parcial. En caso de rendir recuperatorio del parcial 2, el punto extra se suma a la nota obtenida en el recuperatorio.

Aunque no se califica: estilo de código, simpleza, elegancia, comentarios adecuados y velocidad; si el código está muy por fuera de los parámetros aceptables, se podrá desaprobar el trabajo aunque sea funcionalmente correcto.

Plataforma de trabajo

Para desarrollar el trabajo se utilizará la plataforma de google denominada colaboratory research, en la cual se utiliza el entorno de programación *Jupyter Notebook*, el lenguaje Python y el *wrapper* de OpenCL denominado *pyopencl*. La plataforma de investigación de google dispone de una GPU tesla 80k marca nvidia, 2 procesadores core Xeon 2.2GHz y 13GB RAM (puede variar de acuerdo a la disponibilidad). Considerar que el tiempo de vida máximo de una máquina virtual es de 12 horas, o 90 minutos sin utilizarse. En todos los ejercicios resueltos, el programa en OpenCL deberá seleccionar la plataforma donde se encuentra la GPU de nvidia.

Ejercicio 1: Ejecución secuencial de Kernels (Obligatorio para regularizar)

Realizar un programa en OpenCL que espeje verticalmente, horizontalmente y rote una imagen alrededor de un punto de origen, un ángulo determinado.

1. Descripción general

El problema deberá dividirse en tres partes (kernels) donde cada una realiza distintos procesos, rotación, espejado vertical y espejado horizontal. Dependiendo de los procesos que se deseen realizar, se deberán ejecutar uno o más kernels. Los tres procesos, a partir de la posición de un pixel deberán calcular la nueva posición. Para realizar el espejado, se debe trazar una línea imaginaria (vertical u horizontal dependiendo el tipo de espejado a realizar) e intercambiar cada píxel de un lado con su correspondiente del otro. En el caso de una rotación θ alrededor de un punto origen (x_o, y_o) se puede lograr mapeando cada pixel (x_i, y_i) a su nueva posición (x_f, y_f) de la siguiente manera:

$$x_f = cos(\theta)\,(x_i - x_o) - sin(\theta)\,(y_i - y_o) + x_o$$
$$y_f = sin(\theta)\,(x_i - x_o) + cos(\theta)\,(y_i - y_o) + y_o$$

2. Modelado del problema

La imagen debe ser leída y convertida a un array de dos dimensiones, siendo cada elemento

de la matriz un píxel de la imagen. Recordar que esta matriz se convierte en un array de una dimensión para escribirla en la memoria del dispositivo. No se realiza ningún procesamiento con el contenido de cada uno de los pixeles, solo se relocalizan a la nueva posición.
La imagen de entrada y de salida deben ser representadas por matrices de iguales tamaños. Esto implica que al rotar la imagen, algunos pixeles de la imagen original pueden quedar por fuera de la imagen resultante, y por lo tanto no se mostrarán. Además, algunos pixeles de la imagen de salida no recibirán ningún píxel de la imagen original, en este caso, dichos pixeles deberán mostrarse en blanco.

3. Requerimientos y Sugerencias de Implementación

Este problema es altamente paralelizable, ya que la nueva posición de cada píxel es independiente del cálculo de la posición de los demás pixeles. Por lo tanto, cada work-item deberá calcular la nueva posición de un píxel de entrada. El tipo de descomposición de datos que se utiliza, (respecto a la entrada o respecto a la salida) deberá analizarse y justificarse (en una celda de texto del *notebook*). El problema sugiere de manera directa un tratamiento bidimensional del espacio indexado de kernel. Donde cada *work-item* tendrá un identificador único del tipo (x_i, y_i) obtenible con las funciones get_global_id(0) y get_global_id(1). Dada la naturaleza del problema, no es necesario utilizar sincronismo entre los work-items, sin embargo, **es requerimiento utilizar un command-queue out-of-order** y establecer el orden de ejecución de los tres kernels mediante eventos.

3. Formatos de entrada/salida

Se debe determinar una celda de código del notebook para configurar las especificaciones de ejecución con el siguiente formato estándar:

- Una lista donde se determine el orden y los procesos que se desean realizar.
- El número de procesos (*p*) a realizarse puede ser mayor a 3.
- En el caso de la rotación debe indicarse mediante una lista el centro de rotación y el ángulo a rotar: [ROT, C_x, C_y, ángulo]
- El ángulo de rotación en formato *float*.

Ej: [EV,EH,[ROT, 300, 300, 56.3], [ROT, 0, 300, 90.0]].

La salida del programa mostrará la imagen de entrada y de salida.

Ejercicio 2: Simulación física de flujo de calor en superficies (Obligatorio para promocionar)

Bajo el paradigma de programación OpenCL se pide diseñar e implementar la simulación física de flujo de calor descrita a continuación.

1. Descripción general

Se dispone de una placa cuadrada plana de un material uniforme (ej: una plancha de metal). En algunos puntos de esta placa se aplica calor con una fuente de calor (ej: una llama) de modo de mantener la temperatura constante en ese punto. Si bien la aplicación de calor es

puntual, el calor tiende a distribuirse alrededor de la fuente y, luego de cierto tiempo, los lugares de la placa cercanos a la fuente estarán más calientes que los lejanos. En esta simulación modelamos como varía la temperatura en distintos puntos del material a lo largo del tiempo.

2. Modelado del problema

Si bien la temperatura varía continuamente a lo largo de la placa, nuestro modelo va a ser discreto, es decir, separar la placa en una cantidad finita de áreas, donde mediremos la temperatura promedio en cada una de esas áreas. La división de la placa se hará en una grilla uniforme de $N \times N$ bloques, donde para cada área se almacenará la temperatura en un *double*. La placa está inicialmente a temperatura ambiente (Tambiente). Llamaremos j a la cantidad de fuentes de calor. La posición de la i-ésima fuente de calor (con $0 \leq i < j$) estará dada por valores enteros (x_i, y_i), donde $0 \leq x_i < N$ y $0 \leq y_i < N$. El valor x representa la columna de la grilla donde está situada la fuente y el valor y representa la fila. La temperatura de la i-ésima fuente de calor será denominada t_i, y será un valor de punto flotante (de tipo *double*).

Si T es la matriz de temperaturas, su estado inicial será:

- $T_{pq} = t_i$ cuando es una fuente de calor
- $T_{pq} = Tambiente$ en caso contrario

La simulación a realizar será iterativa, en k pasos ($k \geq 0$). En cada paso se construye un T' a partir de T, que al final de la iteración sustituye al *T original*:

- $T'_{pq} = t_i$ cuando es una fuente de calor
- $T'_{pq} = promedio([T_{pq}] ++ vecinos_T(q,p))$ caso contrario

En la definición anterior, *promedio* tiene su definición usual (*promedio(xs) = sum(xs) / len(xs)*). Los vecinos concretamente son los elementos inmediatamente adyacentes a una posición *(x, y)* (arriba, abajo, izquierda, derecha), pero teniendo cuidado con los bordes. Por ejemplo, si $N = 10$:

- $vecinos_T(0, 0)$ tiene 2 vecinos: $[T_{1\,0}, T_{0\,1}, Tambiente, Tambiente]$
- $vecinos_T(9, 3)$ tiene 3 vecinos: $[T_{8\,3}, T_{9\,2}, T_{9\,4}, Tambiente]$
- $vecinos_T(6, 8)$ tiene 4 vecinos: $[T_{6\,7}, T_{7\,8}, T_{6\,9}, T_{5\,8}]$

Considerando que Tambiente = 25, N = 9, las celdas en amarillo son el borde cuya temperatura es Tambiente, pero con la particularidad de que el valor no es modificable. Además, las fuentes de calor se encuentran resaltadas en negrita. Se muestra a continuación el resultado para las primeras dos iteraciones.

Iteracion 1:

	70.9	36.47								
	36.47									
					42.05					
				42.05	**93.2**	42.05				
					42.05					
							34.4			
						34.4	**62.6**	34.4		

Iteracion 2:

	70.9	36.47	27.87							
	36.47	30.73								
	27.87				29.26					
				33.52	42.05	33.52				
			29.26	42.05	**93.2**	42.05	29.26			
				33.52	42.05	33.52				
					29.26		27.35			
						29.7	34.4	29.7		
					27.35	34.4	**62.6**	34.4	27.35	

En resumen, y desde muy alto nivel, el problema a calcular sin paralelismo y en pseudocódigo es:

```
Leer datos de entrada N, k, j, (x[0], y[0], t[0]) ... (x[j-1], y[j-1], t[j-1])
   Construir T inicial
   repetir k veces:
         Dado T, construir T'
         T := T'
   Mostrar T
```

3. Requerimientos y sugerencias de Implementación

En general, el problema resulta altamente paralelizable dado que el cálculo de la temperatura de cada celda de la matriz *T,* se puede calcular independientemente leyendo los valores de los

vecinos en el estado anterior *k-1*. Sin embargo plantea un desafío de sincronismo en el tiempo (estado a estado) dado que se debe garantizar que las temperaturas de *Tk-1* deben estar calculadas antes de calcular las de *Tk*.

El usuario espera sólo el resultado final de la matriz temperatura *T*. Se deben recorrer todos los *k* pasos directamente hasta llegar al último, el cual será finalmente impreso en pantalla.

Es evidente que el planteo permite que el *Kernel* OpenCL itere internamente a lo largo de los pasos *k*. En otras palabras, se hace una sola copia de memoria (matriz resultado *Tk*) desde el dispositivo al host. Debe quedar claro a nivel conceptual que se requiere de **sincronismo local** (a nivel del kernel) entre cada iteración. Recordar que para que una barrera sea válida los *work-items* deberán estar agrupados en el mismo *work-group*.

Para explotar sistemas capaces de paralelización masiva (GPU), se pide diseñar un programa OpenCL en el que cada celda de la grilla *N×N* sea tratada por un *work-item*. Las naturaleza bidimensional del problema sugiere de manera directa que las dimensiones del espacio de kernel deberá ser de 2, por lo que cada *work-item* tendrá un identificador único del tipo (x_i, y_i) obtenible con las funciones get_global_id(0) y get_global_id(1).

Finalmente, se pide que el programa tome los valores de tiempos asociados a las transferencias de memoria host-device y viceversa, así como los intervalos de tiempo de ejecución de kernels. Para esto se deberán usar los modos de profiling explicados en clase.

Formatos de entrada/salida

Se debe determinar una celda de código para configurar el modo de ejecución de la simulación con el siguiente formato estándar:

- Un entero *N* (tamaño en filas/columnas de la grilla). La grilla es de NxN elementos.
- Un entero *k* (cantidad de iteraciones).
- Un double Tambiente donde se defina la temperatura de borde
- Una lista de listas donde para cada fuente de calor se detalla la posición en X, la posición en Y y la temperatura de la misma. Con el siguiente formato:

$$[\ [x_0, y_0, t_0], [x_1, y_1, t_1], \ldots ,[\ x_{j-1}, y_{j-1}, t_{j-1}]\ ]$$

La salida del programa mostrará una fila de la matriz por línea, en orden. Cada fila será una lista de valores separados por espacios, con cada valor impreso usando el formato "%.2f" de C.

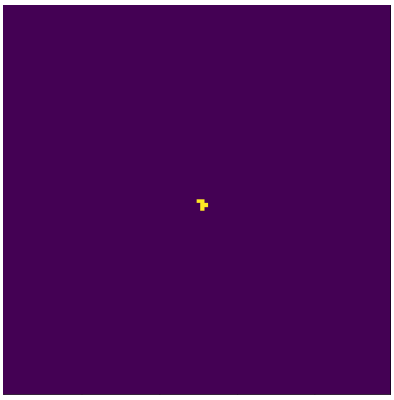
Ejercicio 3: Juego de la vida de John Conway (1 punto extra)

Se pide diseñar e implementar en OpenCL el juego de la vida creado por el matemático John Conway.

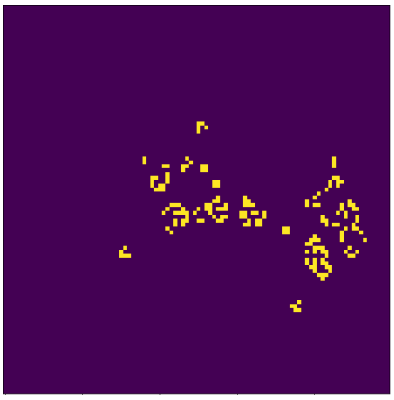
1. Descripción general

El juego de la vida de Conway es de cero participantes, donde el desarrollo del mismo está únicamente determinado por su estado inicial. El juego de la vida no es un típico juego de

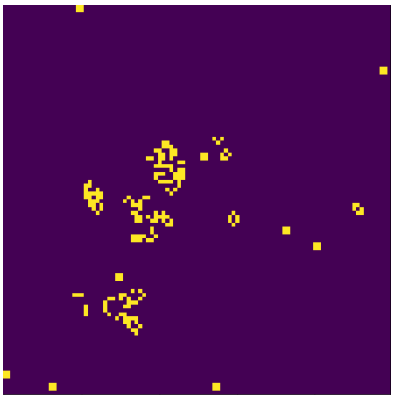
computadoras, es un autómata celular donde las células pueden sobrevivir, reproducirse o morir. Es un excelente ejemplo sobre como estructuras complejas pueden generarse a partir de reglas sencillas.



Generación: 0
Población: 5



Generación: 150
Poblacion: 214



Generación: 600
Población: 197

2. Modelado del problema

El juego consiste en una grilla de NxN donde cada celda puede o no estar ocupada por una única célula. Donde la supervivencia, reproducción o deceso está condicionado por el siguiente conjunto de reglas:

- Para un espacio donde ya existe una célula:
  + Cada célula con uno o cero vecinos muere por soledad.
  + Cada célula con cuatro o más vecinos muere por sobrepoblación.
  + Cada célula con dos o tres vecinos sobrevive.
- Para un espacio vacío:
  + Cada celda con exactamente tres vecinos se ocupa por una nueva célula.

La grilla estará representada por una matriz de dos dimensiones de valores enteros, donde un uno indica una celda ocupada por una célula viva y un cero indica una celda desocupada. Una celda donde hay una célula muerta se la considera igual que una celda desocupada. En este caso, se considera vecino a las células que se encuentran en cualquiera de las 8 celdas adyacentes.

3. Requerimientos y Sugerencias de Implementación

El grado de paralelizable del problema es muy alto dado que la existencia o no de una célula en cada celda se puede determinar de forma independiente, leyendo los valores de los vecinos en el estado anterior *t-1*. Sin embargo, es necesario asegurarnos que antes de empezar a generar la matriz de t +1, se haya finalizado la generación de la matriz de t. Para esto, se deberá realizar un sincronismo del lado del host (utilizando herramientas como colas de comandos en-orden o eventos). Dado que el sincronismo se realiza del lado del *host*, no es necesario que todos los work-items estén en el mismo work-grup.

Para explotar sistemas capaces de paralelización masiva (GPU), se pide diseñar un programa OpenCL en el que cada celda de la grilla *N×N* sea tratada por un *work-item* (descomposición de

datos de salida). Las naturaleza bidimensional del problema sugiere de manera directa que las dimensiones del espacio de kernel deberá ser de 2, por lo que cada *work-item* tendrá un identificador único del tipo (x_i, y_i) obtenible con las funciones get_global_id(0) y get_global_id(1).

Formatos de entrada/salida

Se debe determinar una celda de código del notebook para configurar el modo de ejecución del juego de la vida con el siguiente formato estándar:

- Un entero *N* (tamaño en filas/columnas de la grilla). La grilla es de NxN elementos.
- Un entero *k* (cantidad de iteraciones).
- Una lista de listas donde se determine la posición en X y en Y para cada celda que contenga una célula viva en el t = 0. Con el siguiente formato:

$$[\ [x_0, y_0], [x_1, y_1], \ldots ,[\ x_{j-1}, y_{j-1}]\ ]$$

La salida del programa deberá mostrar *k* imágenes de dos colores, un color para celdas que contienen células vivas y otro para celdas vacías.