UNIVERSIDAD CENTRAL DE VENEZUELA

FACULTAD DE CIENCIAS – ESCUELA DE COMPUTACIÓN

CENTRO DE COMPUTACIÓN PARALELA Y DISTRIBUIDA

LABORATORIO DE COMPUTACIÓN HETEROGÉNEA DE ALTO RENDIMIENTO

ASIGNATURA: FUNDAMENTOS DE PROGRAMACIÓN PARALELA (FPP) - Semestre: 2-2023

Profesor: Carlos Acosta

MICROPROYECTO NO. 1:
PIPELINE DE TAREAS PARA EL PROCESAMIENTO PARALELO DE UNA MATRIZ CUADRADAS NxN USANDO OPENCL

ENUNCIADO DEL PROBLEMA:

El Laboratorio de Computación Heterogénea de Alto Rendimiento de la Universidad Central de Venezuela requiere que los estudiantes de la asignatura “Fundamentos de Programación Paralela” desarrollen un programa paralelo. Debe estar programado en lenguaje C++ usando el API OpenCL (Open Computing Language). Se debe correr en un ambiente heterogéneo simulando los dispositivos CPU, GPU y FPGA (cada uno se simula con una tarea paralela en el Host-CPU.

Esta aplicación debe cumplir con los siguientes requerimientos:

1) El programa paralelo debe implementar el patrón de computación “pipeline” con **P=3** etapas $\mathbf{P_1=T_1}$, $\mathbf{P_2=T_2}$ y $\mathbf{P_3=T_3}$, donde se encadenen dichas operaciones o tareas distintas para procesar la matriz **M**.

2) La matriz de entrada **M** es de dimensiones **NxN,** donde N es solo un parámetro, por ejemplo puede ser N=6, entonces seria una matriz 6x6). Cada fila de la matriz M contiene números **enteros positivos** $\mathbf{e_1, e_2, e_3, ..., e_i, ..., e_{n-1}, e_N}$. Además, esta matriz **M** debe procesarse por filas o lineas, donde a cada dato $\mathbf{e_i}$ de la fila se le aplicará una operación aritmética en cada etapa del “pipeline”.

3) Cada etapa del “pipeline” tiene asignada una tarea que se aplica a cada elemento de la fila de datos de la matriz **M.** De esta manera un operador aritmético distinto se realiza en cada etapa, asi la tarea $\mathbf{T_1}$**=operador 1,** $\mathbf{T_2}$**=operador 2, ...** $\mathbf{T_p}$**=operador p**. Estas tareas u operaciones serán subprogramas (subrutinas, métodos, funciones o procedimientos). En conclusión, la primera etapa del “pipeline” debe realizar la Tarea $\mathbf{T_1}$, la segunda etapa la Tarea $\mathbf{T_2}$ y asi sucesivamente.

4) Por ejemplo:

4.1) La Tarea $\mathbf{T_1}$ (primera etapa del “pipeline”) debe incrementar en una unidad ($\mathbf{e'_i = e_i}+1$) cada dato que entra al “pipeline” (de la fila de la matriz **M** que va ingresando a la primera etapa del “pipeline”).

4.2) La Tarea $\mathbf{T_2}$ (segunda etapa del “pipeline”) debe elevar al cuadrado ($\mathbf{e''_i = e'_i * e'_i}$) el dato resultante de la etapa previa del “pipeline” (de la fila de la matriz **M** que va atravesando el “pipeline”).

4.3) La Tarea $\mathbf{T_3}$ (tercera y ultima etapa del “pipeline”) debe restar una unidad ($\mathbf{e'''_i = e''_i - 1}$) al dato que recibe de la etapa anterior del “pipeline” (cada entrada de la fila de la matriz **M** que va atravesando el “pipeline”).

4.4) El resultado del procesamiento de la fila se va obteniendo, dato a dato, a la salida de la última etapa del “pipeline” y es una matriz **M’** (eme prima). Aqui cada entrada fue procesada y el resultado refleja este cambio de valor por las diferentes operaciones realizadas al atravesar el “pipeline”.

5) Los datos de la matriz **M** serán leidos desde un archivo de entrada **inFile** y los resultados de salida almacenados en un archivo de salida **outFile**.

6) El archivo de entrada **inFile** debe indicar en su primera linea la dimensión **N** de la Matriz cuadrada de entrada **M**. Luego, cada linea siguiente del archivo corresponde a una fila de la matriz **M**. A partir de la segunda linea y hasta la penúltima linea del archivo se tendrán las filas de datos de la matriz **M**, con **c**ada entrada de la fila separada por comas. La ultima fila, el primer carácter debe ser un ‘-1’ para indicar el final del archivo.

7) En el archivo de salida **outFile,** la primera linea debe indicar la dimensión **N** de la Matriz **M’** (eme prima) cuadrada de salida. Luego, en las lineas siguientes se deben colocar los datos resultante de cada linea, también separados por comas. Finalmente, el primer carácter de la última fila debe ser un ‘-1’ para indicar el final del archivo.

8) Se debe implementar el “pipeline” simulando el CPU, el GPU y el FPGA con tareas paralelas de software en openCL (Ver **Figura 1**). Ademas, el “pipeline” se debe implementar como una plantilla en C++ que reciba como parámetros la matriz M (archivo de entrada), su tamaño o dimensión, el número de etapas del “pipeline”, las tarea de cada etapa del” pipeline” y el archivo de salida.

PARÁMETROS DE LA PLANTILLA C++:

a) **N** es la dimensión de la matrices cuadradas **M** y **M'**,

b) **P** es el tamaño del pipeline, en este proyecto P=3,

c)**TareasEtapas** es el arreglo con apuntadores a las tareas, operaciones o métodos (subprogramas) que se asignan a cada etapa del "pipeline", respectivamente.

e) **"CPU", "GPU", "FPGA"** son etiquetas para identificar e indexar a cada dispositivo de procesamiento (simulado).

f) **inFile** es el archivo de entrada de la matriz **M**, con entradas e_1, e_2, e_3, …, e_i, …, e_{n-1}, e_N

g) **outFile** es el archivo de salida con la matriz resultante o procesada, con salidas r_1, r_2, r_3, …, r_i, …, r_{n-1}, r_N. Donde cada $r_i = P3(P2 (P1 (e_i)))$

$= P3(P2 (P1 (e_i+1)=e'_i)=e'_i * e'_i)=e''_i - 1= e'''_i$

FORMATO DE DATOS DEL ARCHIVO DE ENTRADA:

N,

e_{11}, e_{12}, e_{13}, …, e_{1i}, …, e_{1n-1}, e_{1N}

…

e_{N1}, e_{N2}, e_{N3}, …, e_{Ni}, …, e_{Nn-1}, e_{NN}

-1

FORMATO DE RESULTADOS DEL ARCHIVO DE SALIDA:

N,

r_{11}, r_{12}, r_{13}, …, r_{1i}, …, r_{1n-1}, r_{1N}

…

r_{N1}, r_{N2}, r_{N3}, …, r_{Ni}, …, r_{Nn-1}, r_{NN}

-1

HERRAMIENTAS A UTILIZAR:

Instale un compilador de C++, los drivers de los dispositivos compatibles con openCL y un editor de código como VS Code. Use una distro Linux compatible con Ubuntu 20.04 para realizar el microproyecto.

OBSERVACIÓN:

En la **Figura 2** se muestra un pseudocódigo (no funcional aún, debe reescribirse, completarse e instanciarse luego en un programa principal "*main ()"* en OpenCL/C) el cual muestra lo que se busca en el microproyecto. El programa fuente desarrollado se debe compilar y poderse ejecutar en el CPU del Host openCL simulando el CPU, GPU y FPGA con tareas paralelas.

Cualquier duda o pregunta deben resolverla de forma oportuna y con la debida anticipación a la entrega del microproyecto. La fecha de entrega del microproyecto es el lunes 19/02/2024.

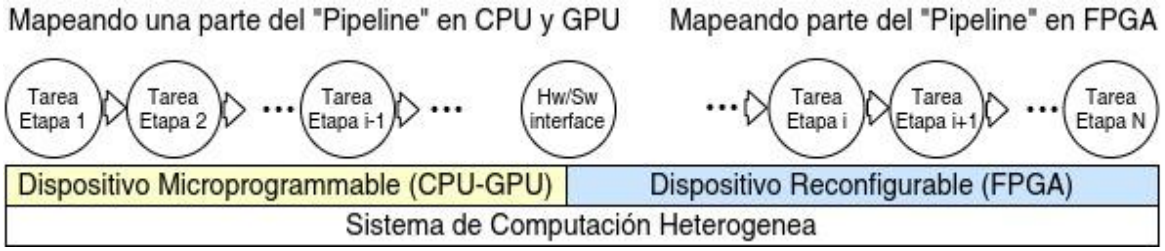

Figura 1. Diagrama demostrativo del patrón de cómputo "Pipeline" particionado entre CPU, GPU y FPGA (simulado) en un sistema de computación heterogénea.

```
void Plantilla_Pipeline(*device_cpu, *device_gpu, *device_fpga, *tasks_List[],
                *List_imagesIn, *List_imagesOut, segment_size) {
cl_uint num_devices_returned;
cl_device_id devices[3];
channel float cpu_gpu_channel, gpu_fpga_channel, fpga_cpu_channel;
func_type *cpu_Tasks, *gpu_Tasks, *fpga_Tasks;
cpu_Task = &tasks_List[cpu_pipesize];
gpu_Task = &tasks_List[gpu_pipesize];
fpga_Task = &tasks_List[fpga_pipesize];

/* Get the device(s) */
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &devices[0], &num_devices_returned);
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &devices[1], &num_devices_returned);
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_FPGA, 1, &devices[2], &num_devices_returned);

/* Create contexts for devices */
cl_context context;
context = clCreateContext(0, 3, devices, NULL, NULL, NULL, &err);

/* Create commands queue for devices */
cl_command_queue queue_gpu, queue_cpu, queue_fpga;
device_gpu = clCreateCommandQueue(context, devices[0], 0, &err);
device_cpu = clCreateCommandQueue(context, devices[1], 0, &err);
device_fpga = clCreateCommandQueue(context, devices[2], 0, &err);

/* Basic pipeline */
__kernel void cpu(__global int* in) {
  for (int i = 0; i < workload_size; ++i) {
    int *image_segment = read_Image();         // Get images by segments
    value[segment_size] = cpu_Tasks(&image_segment, 0, 255); // do some work
    write_channel(cpu_gpu_channel, &value);    // send data to the next partition
  }
}

__kernel void gpu() {
  for (int i = 0; i < workload_size; ++i) {
    int value = read_channel(cpu_gpu_channel);  // take data from cpu
    value[segment_size] = gpu_Tasks(&image_segment, 0, 255);  // do some work
    write_channel(gpu_fpga_channel, &value);    // send data to the next partition
  }
}

__kernel void fpga(__global int* out) {
  for (int i = 0; i < workload_size; ++i) {n
    int value = read_channel(gpu_fpga_channel);         // take data from gpu
    value[segment_size] = fpga_Tasks(&image_segment, 0, 255); // do some work
    write_channel(fpga_cpu_channel, &value);            // write result in the end
  }
}

/* Start concurrently tasks (kernels) for partitions in CPU-GPU-FPGA */
clEnqueueTask(device_cpu, *TasksList[cpu_size]);
clEnqueueTask(device_gpu, *TasksList[gpu_size]);
clEnqueueTask(device_fpga,*TasksList[fpga_size]);

clFinish(device_fpga); // last kernels in our pipeline

}
```

Figura 2. Pseudocódigo de plantilla para el patrón de cómputo “Pipeline” particionado entre CPU, GPU y FPGA (Este pseudocódigo no es funcional, es sólo para efectos demostrativos)