1.Introducción y objetivos

El siguiente trabajo es parte de la asignatura Tarjetas Gráficas y Aceleradores de la Facultad de Informática de Barcelona. Se trata de un trabajo de temática abierta pero directamente relacionado con la aplicación del lenguaje CUDA.

Para la realización del proyecto me he decantado por analizar diferentes estrategias de paralelización mediante CUDA de un algoritmo genético (AG de ahora en adelante), el cual se explicará con algo más de detalle en la posterior sección.

El objetivo principal de este trabajo es ver qué implementación puede aportar el mayor Speedup a la aplicación, la cual simplemente mediante el AG minimiza la función Sphere (más detalles en el Anexo I).

El SpeedUp se mediara mediante 2 parametros concretos:

- POBLACION: Es el numero total de valores que el programa procesará
- LONG_COD: Es la longitud del codigo genetico en bits. Contra mayor es esta, mejor precisión obtenemos en el calculo.

Para facilitar las cosas y no tenernos que preocupar de el hecho de que estos valores no sean fácilmente divisibles entre los bloques y los threads, siempre serán potencias de 2.

Para obtener este Speedup llevare a cabo dos estrategias diferenciadas:

- Reescribir las funciones más utilizadas en forma de Kernel.
- Reestructurar todo el código para que cada ronda del AG se realice dentro de un solo Kernel en la medida de lo posible y siempre y cuando la función sea suficientemente significativa como para introducirla en un kernel.

2.Algoritmos Genéticos

En esta sección podría explicar en qué consisten los algoritmos genéticos en un nivel general, pero al no ser este el objetivo que nos atañe, explicaré concretamente las funciones importantes para el algoritmo con el que trabajaremos , aunque como pincelada general podemos ver el esquema de cualquier algoritmo genético en la imagen que tenemos a continuación.

Analizando el código podemos apreciar que la función que lleva el peso del computo es AG(). Entrando a analizar esta función vemos que consta de un gran bucle que calcula nuevas poblaciones hasta que la elite de esta población cumpla la condición de salida, para nuestro caso encontrar los mínimos de la función sphere.

```
do
{
    seleccion = seleccionTorneos(poblacion);
    cruzarSeleccion(seleccion);
    seleccion[POBLACION-1] = elite(poblacion);
    free(poblacion);
    poblacion = seleccion;
    generacion++;
} while (elite(poblacion).aptitud > pow(10,-2));
```

Por tanto con este pequeño análisi, es fácil apreciar que las funciones en las cuales hay que centrarse a la hora de convertir en kernel no son más que 3:

1. selecciónTorneos
2. CruzarSelección
3. elite

2.1. SeleccionTorneos

Esta función simplemente rellena un vector con los mejores especímenes de la población actual para que pasen a una fase posterior. Esta selección se lleva a cabo mediante un comparación entre 2 de ellos y seleccionando el que posee un mejor aptitud, para el caso que nos atañe el que más se acerca al mínimo de la función sphere.

```
for (i=0; i<POBLACION-1; i++)
{
    candidato_a = poblacion[(int) (((double) POBLACION)*rand()/(RAND_MAX+1.0))];
    candidato_b = poblacion[(int) (((double) POBLACION)*rand()/(RAND_MAX+1.0))];

    if (candidato_a.aptitud < candidato_b.aptitud)
        seleccion[i] = candidato_a;
    else
        seleccion[i] = candidato_b;
}
```

Como podemos apreciar el cuerpo de esta función ofrece un potencial de paralelización muy alto, ya que no tiene ningún tipo de dependencia de datos, y los candidatos a competir se seleccionan de manera pseudoaleatoria. A continuación vemos como finalmente quedaría esta función después de traducirla a kernel Cuda.

```
__device__ float my_rand(unsigned int *seed){

unsigned long a = 16807;
unsigned long m = 2147483647;// 2^31-1;
unsigned long x = (unsigned long) *seed;

x = (a*x) % m;

*seed = (unsigned int) x;

return((float) x)/m;

}

__global__ void seleccionTorneos_CUDA(Individuo * poblacion_act,Individuo * next_generation){

        unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
        unsigned int seedone = tid +1;
        unsigned int seedtwo = tid+2;

        Individuo a = poblacion_act[(int) (((double) POBLACION)*my_rand(&seedone)/(RAND_MAX+1.0))];
        Individuo b = poblacion_act[(int) (((double) POBLACION)*my_rand(&seedtwo)/(RAND_MAX+1.0))];

        if(a.aptitud < b.aptitud)
                next_generation[tid] = a;
        else
                next_generation[tid] = b;

}
```

Como podemos ver se ha eliminado el bucle ya que ahora cada thread calcula a un miembro de la siguiente generación. Destacar la necesidad de crear una función que ejerciera de rand() ya que desde cuda no se puede llamar a esta función.

2.2. CruzarSeleccion

En esta funcion se recorre el vector que contiene la población intercanviando parte del genoma de dos individuos almacenados consecutivamente, en función de si un cierto valor aleatorio es inferior a una probabilidad de cruce definida. Finalmente una vez cruzado el genoma se añade cierta mutación aleatoria a este para prepararlo para la siguiente ronda.

```
void cruzarSeleccion (Individuo * seleccion)
{
    int i, j, punto, aux;
    double x, y;

    for(i=0; i<POBLACION-1; i+=2)
    {
        if((double) rand()/(RAND_MAX+1.0) < PROB_CRUCE)
        {
            punto = (int) (((double) LONG_COD)*rand()/(RAND_MAX+1.0));

            for(j=punto; j<LONG_COD; j++)
            {
                aux=seleccion[i].genotipo[j];
                seleccion[i].genotipo[j]=seleccion[i+1].genotipo[j];
                seleccion[i+1].genotipo[j]=aux;
            }

            mutacionHijos(&seleccion[i]);

            decoder(&x, &y, seleccion[i].genotipo);
            seleccion[i].aptitud = fitness(x,y);

            decoder(&x, &y, seleccion[i+1].genotipo);
            seleccion[i+1].aptitud = fitness(x,y);
        }
    }
}
```

Por tanto podemos ver que la función tiene a priori un gran paralelismo, aunque si la analizamos detenidamente hay que ser cuidadoso en este ya que encontramos condiciones de carrera y el hecho de que se esten tratando dos elementos en una sola iteración provoca que a la hora de realizar las llamadas a las funciones se tenga que tener en cuenta de cara a reducir o bien el numer de threads en cada block a la mitad o bien el numero de blocks. En referencia a las condiciones de carrera, en este caso en particular al contar el algoritmo con una componente aleatoria muy fuerte no nos condicionan en exceso estas, ya que lo unico que provoca es una aleatoriedad añadida en cada ejecución que incluso podria resultar veneficiosa. En cuanto a las llamadas a funciones en el interior de esta, a pesar de ser estas paralelizables se ha decidido

convertirlas en funciones de device y asi minimizar el impacto a la hora de mover datos del host al device y viceversa.

```
__global__
void cruzarSeleccion_CUDA(Individuo * seleccion){

    unsigned int tid = (blockIdx.x * blockDim.x + threadIdx.x) * 2;

    if((double) my_rand(tid)/(RAND_MAX+1.0) < PROB_CRUCE)
        {
            punto = (int) (((double) LONG_COD)*my_rand(tid)/(RAND_MAX+1.0));

            for(j=punto; j<LONG_COD; j++)
            {
                aux=seleccion[tid].genotipo[j];
                seleccion[tid].genotipo[j]=seleccion[tid+1].genotipo[j];
                seleccion[tid+1].genotipo[tid]=aux;
            }

            mutacionHijos_CUDA(&seleccion[tid]);

            decoder_CUDA(&x, &y, seleccion[tid].genotipo);
            seleccion[tid].aptitud = pow(x,2) + pow(y,2);

            decoder_CUDA(&x, &y, seleccion[tid+1].genotipo);
            seleccion[tid+1].aptitud = pow(x,2) + pow(y,2);
        }
    }
}
```

2.3 - Elite

Como podemos ver en esta función se recorre toda la población buscando cual es el individuo con mejor aptitud, o dicho de otra manera cual es el individuo que más se aproxima a la mejor solución final.

```
Individuo elite (Individuo * poblacion)
{
    int i;
    Individuo best = poblacion[0];

    for(i=0; i<POBLACION; i++)
        if(best.aptitud > poblacion[i].aptitud)
            best = poblacion[i];

    return best;
}
```

Como podemos apreciar esta función es dificil de paralelizar ya que siempre requiere conocer todos los valores de la aptitud de la población. Una posible solución seria ordenar el vector de población por aptitudes, por tanto esta función solo tendria que visitar la posición 0 para obtener el individuo con mejor aptitud. De todas formas esta mejora sería difícil de implementar en cuda y no aportaría ningún speed up para el tamaño de datos que estamos tratando. Por tanto considerando que esta optimización queda algo alejada de los objetivos de este trabajo me he decantado por no hacer ningún tipo de mejora sobre esta función y dejar que sea ejecutada sobre el host.

2.4 - Unificando Funciones

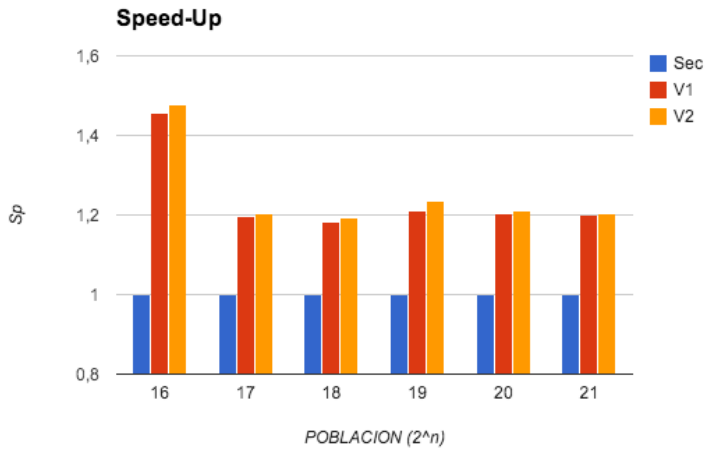
En el siguiente apartado unificaremos las funciones expuestas anteriormente en una sola, de tal manera que podamos reducir la comunicación con el host para no copiar datos más de una vez. También hay que tener en cuenta que un warp comparte el PC para los threads que lo componen, por tanto ahorrandonos algunas llamadas a las funciones deberíamos obtener una mejora de tiempo evitando la serialización de los threads. También se han añadido mejoras adicionales con respecto a la eliminación de ciertos saltos que tambien provocaban serialización y además se ha añadido una mejor gestión de la memoria del host evitando accesos a la memoria compartida haciendo una copia local de los individuos a tratar

```
__global__ void seleccionTorneos_and_cruzar_CUDA(Individuo * poblacion_act,Individuo * next_generation){

    unsigned int tid = (blockIdx.x * blockDim.x + threadIdx.x) * 2;
    unsigned int seed = tid;

    Individuo a = poblacion_act[(int) (((double) DPOBLACION)*my_rand(&seed)/(RAND_MAX+1.0))];
    Individuo b = poblacion_act[(int) (((double) DPOBLACION)*my_rand(&seed)/(RAND_MAX+1.0))];
    Individuo c = poblacion_act[(int) (((double) DPOBLACION)*my_rand(&seed)/(RAND_MAX+1.0))];
    Individuo d = poblacion_act[(int) (((double) DPOBLACION)*my_rand(&seed)/(RAND_MAX+1.0))];

    if(a.aptitud < b.aptitud)
        next_generation[tid] = a ;
    else
        next_generation[tid] = b;

    if(c.aptitud < d.aptitud)
        next_generation[tid+1] = c ;
    else
        next_generation[tid+1] = d;

    int j, punto, aux,i;
    double x, y;
    x=0.0;
    y=0.0;

    a=next_generation[tid];
    b=next_generation[tid+1];
```

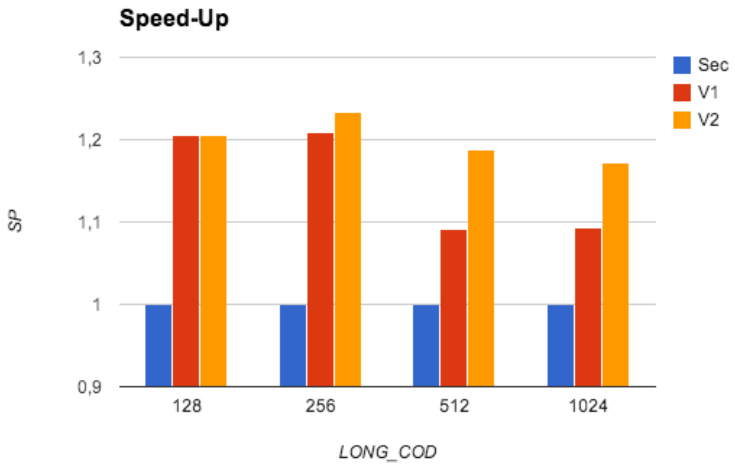
```
if((double) my_rand(&seed)/(RAND_MAX+1.0) < PROB_CRUCE)
    {
        punto = (int) (((double) LONG_COD)*my_rand(&seed)/(RAND_MAX+1.0));

        for(j=punto; j<LONG_COD; j++)
        {
            aux=a.genotipo[j];
            a.genotipo[j]=b.genotipo[j];
            b.genotipo[j]=aux;
        }

            for(j=0; j<LONG_COD; j++){
                a.genotipo[j]= ((double) my_rand(&seed)/(RAND_MAX+1.0) < PROB_MUTACION);
                b.genotipo[j]= ((double) my_rand(&seed)/(RAND_MAX+1.0) < PROB_MUTACION);
            }


        //decoder_CUDA(&x, &y, seleccion[tid].genotipo);
        // calculo del primer decimal
        for(i=0; i<LONG_COD/2; i++)
            x += a.genotipo[i] *  (1<<((unsigned int)((LONG_COD/2)-(i+1))));
        x = (x) * INTERVALO + LIMITE;

        //calculo del segundo decimal
        for(;i<LONG_COD;i++)
            y += a.genotipo[i] * (1<<((unsigned int)((LONG_COD/2)-(i+1))));
        y = (y) * INTERVALO + LIMITE;
        a.aptitud = (x*x) + (y*y);

        //decoder_CUDA(&x, &y, seleccion[tid+1].genotipo);

        for(i=0; i<LONG_COD/2; i++)
            x += b.genotipo[i] *  (1<<((unsigned int)((LONG_COD/2)-(i+1))));
        x = (x) * INTERVALO + LIMITE;

        //calculo del segundo decimal
        for(;i<LONG_COD;i++)
            y += b.genotipo[i] * (1<<((unsigned int)((LONG_COD/2)-(i+1))));
        y = (y) * INTERVALO + LIMITE;
        b.aptitud = (x*x) + (y*y);

        next_generation[tid] = a;
        next_generation[tid+1] = b;
```

3 - Resultados

En el siguiente apartado mostraré dos gráficos diferentes, en uno se modifica el tamaño de la POBLACION dejando constante LONG_COD y en el otro realizaremos lo contrario. Los datos son los siguientes:



Como podemos observar obtenemos un speed-Up medio de 1.3, aunque como vemos no escala de la manera esperada, ya que aunque variemos el tamaño de la entrada este se mantiene constante, hecho realmente extraño. Este se puede deber a la cantidad de bucles que contienen las funciones y que provocan la serialización de un warp, y por eso aunque podamos aumentar el número de threads el escalado no es el esperado. También remarcar que las mejoras aplicadas entre la versión 1 y la version 2 no surten el efecto esperado, hecho que la verdad no se como explicar, ya que en principio operar sobre registros en vez de memoria debería darnos un plus de rendimiento que no se percibe.



Como podemos apreciar en este caso la versión 2 del código destaca algo más sobre la versión 1, aunque el speed-up sigue sin escalar. En este caso el hecho de haber guardado en registros los valores de acceso a memoria si vemos como nos ofrece una mejora temporal, y esto es debido a que LONG_COD se utiliza como variable de salida de un bucle y en es bucle se accede a memoria en cada iteración, por tanto cada vez que aumentamos el tamaño de LONG_COD nos estamos ahorrando accesos a memoria.

4 - Conclusiones

Finalmente y como cierre a este trabajo me gustaría destacar lo verdaderamente difícil que es conseguir un Speed-Up relativamente importante en aplicaciones que a priori parecen dar muchas posibilidades de obtener buenos resultados.

A pesar de no haber obtenido el Speed-Up deseado creo que este trabajo me ha servido para aprender un gran número de cosas referentes a CUDA que desconocía por completo.