Universidad Nacional de Córdoba

FaMAF | GPGPU Computing Group

Paralelizarás tu código por sobre todas las cosas

Dionisio E Alonso

El algoritmo de la regla de Simpson

- Se usa para el cómputo de integrales.
- Realiza sumas de intervalos.

$$\int_a^b f(x)\,dx \approx \frac{a-b}{6}\left[f(a)+4\,f\left(\frac{a+b}{2}\right)+f(b)\right]$$

En código

```
double simpsons_rule(double (*f)(double), double a, double b, int intervals) {
   int i=0;
   double h=0, simpson=0;

   h = (b-a)/intervals;

   for (i=0; i<=intervals; i++) {
      if (i==0 || i==intervals)
         simpson = simpson + f(a+i*h);
      else if (i%2 == 1)
         simpson = simpson + f(a+i*h)*4;
      else
         simpson = simpson + f(a+i*h)*2;
   }  /*
       * It is used (a+i*h) because (a) is the start of the function
       * interval and (i*h) is the amount computed so far.
       */

   return h/3*simpson;
}
```

Versión en MPI (1)

- Se realiza la configuración e inicialización del entorno MPI

```
#include "/opt/mpich/gnu/include/mpi.h"

int main(int argc, char **argv) {
   int i=0;
   double x=0, integ=0;
   int rank=0;

   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Versión MPI (2)

```
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &size);

MPI_Bcast(&intervals, 1, MPI_INT, MASTER, comm);

h = (b-a)/intervals;
interv_per_node = intervals/size;
extra_interv_per_node = intervals%size;

for (i=0; i<interv_per_node; i++) {
    p = interv_per_node*rank+i;
    if (p==intervals-1) simpson+=f(a+p*h);
    if (p==0 || p==intervals)
        simpson = simpson + f(a+p*h);
    else if (i%2 == 1)
        simpson = simpson + f(a+p*h)*4;
    else
        simpson = simpson + f(a+p*h)*2;
}   /*
     * It is used (a+i*h) because (a) is the start of the function
     * interval and (i*h) is the amount computed so far.
     */

MPI_Reduce(&simpson, &simpson_tot, 1, MPI_DOUBLE, MPI_SUM, MASTER, comm);

return h/3*simpson_tot;
```

Versión serial

```
double simpsons_rule(double (*f)(double), double a, double b, int intervals) {
   int i=0;
   double h=0, simpson=0;

   h = (b-a)/intervals;

   for (i=0; i<=intervals; i++) {
      if (i==0 || i==intervals)
         simpson = simpson + f(a+i*h);
      else if (i%2 == 1)
         simpson = simpson + f(a+i*h)*4;
      else
         simpson = simpson + f(a+i*h)*2;
   }  /*
       * It is used (a+i*h) because (a) is the start of the function
       * interval and (i*h) is the amount computed so far.
       */

   return h/3*simpson;
}
```

Versión MPI (3)

- Cierre y finalización

```
MPI_Finalize();
```

Versión OpenMP

```
#include <omp.h>

double simpsons_rule(double (*f)(double), double a, double b, int intervals) {
   int i=0;
   double h=0, simpson=0;

   h = (b-a)/intervals;

   #pragma omp parallel for shared(h) private(i) reduction(+:simpson)
   for (i=0; i<=intervals; i++) {
      if (i==0 || i==intervals)
         simpson = simpson + f(a+i*h);
      else if (i%2 == 1)
         simpson = simpson + f(a+i*h)*4;
      else
         simpson = simpson + f(a+i*h)*2;
   }  /*
       * It is used (a+i*h) because (a) is the start of the function
       * interval and (i*h) is the amount computed so far.
       */

   return h/3*simpson;
}
```

Versión serial

```
double simpsons_rule(double (*f)(double), double a, double b, int intervals) {
   int i=0;
   double h=0, simpson=0;

   h = (b-a)/intervals;

   for (i=0; i<=intervals; i++) {
      if (i==0 || i==intervals)
         simpson = simpson + f(a+i*h);
      else if (i%2 == 1)
         simpson = simpson + f(a+i*h)*4;
      else
         simpson = simpson + f(a+i*h)*2;
   }  /*
       * It is used (a+i*h) because (a) is the start of the function
       * interval and (i*h) is the amount computed so far.
       */

   return h/3*simpson;
}
```

¿Qué es CUDA?

- CUDA son las siglas para: ***C***ompute ***U***nified ***D***evice ***A***rchitecture
- CUDA está desarrollado por Nvidia, para el cómputo en paralelo sobre placas de video
- Es la arquitectura elegida a partir de la familia G8X de GPUs
- Viene en varios sabores (C, Fortran, OpenCL, etc.)

Breve y corta introducción a la arquitectura

- Las placas con capacidades CUDA tienen una división interna en grillas
- Cada grilla está subdividida en bloques
- Cada bloque está subdividido a su vez en hilos de ejecución (máximo 512)

¿Qué estructura necesita mi programa para correr en CUDA?

- Inicialización de la placa de video
- Configuración para el llamado a *«kernel»*
- Llamado del *«kernel»*
- Cierre y limpieza

Inicialización de un programa CUDA

```
#include <cuda.h>

extern "C" float simpsons_rule_c(float a, float b, int intervals) {
   float h=0, simpson=0, *ss=NULL, *d_res=NULL;

   h = (b-a)/intervals;

   ss = (float *) calloc(intervals+1, sizeof(float));

   CUDA_SAFE_CALL(cudaMalloc((void **) &d_res, (intervals+1)*sizeof(float)));
   CUDA_SAFE_CALL(cudaMemset((void *) d_res, 0, (intervals+1)*sizeof(float)));
```

¿Qué es un *«kernel»*?

- La placa no puede ejecutar cualquier instrucción
- Las funciones pueden ser llamadas:
  - Desde el host para correr en el host
  - Desde el host para correr en la placa
  - Desde la placa para correr en la placa
- Un kernel es llamado desde el host para correr en la placa

Ejemplo de kernel en Simpson

```
__global__ void simpson_kernel(float a, float h, int intervals, float *simpson) {
   // Determine element to process from thread index
   const int i = blockIdx.x * blockDim.x + threadIdx.x;

   if (i<=intervals) {
      if (i==0 || i==intervals)
         simpson[i] = simpson[i] + f(a+i*h);
      else if (i%2 == 1)
         simpson[i] = simpson[i] + f(a+i*h)*4;
      else
         simpson[i] = simpson[i] + f(a+i*h)*2;
   } /*
      * It is used (a+i*h) because (a) is the start of the function
      * interval and (i*h) is the amount computed so far.
      */
}
```

Versión serial

```
double simpsons_rule(double (*f)(double), double a, double b, int intervals) {
   int i=0;
   double h=0, simpson=0;

   h = (b-a)/intervals;

   for (i=0; i<=intervals; i++) {
      if (i==0 || i==intervals)
         simpson = simpson + f(a+i*h);
      else if (i%2 == 1)
         simpson = simpson + f(a+i*h)*4;
      else
         simpson = simpson + f(a+i*h)*2;
   }  /*
       * It is used (a+i*h) because (a) is the start of the function
       * interval and (i*h) is the amount computed so far.
       */

   return h/3*simpson;
}
```

Bueno, ya tengo mi kernel ¿Cómo lo llamo?

- Un kernel requiere configuración
- Es necesario especificar el número de bloques por grilla e hilos por bloques en los que va a ejecutar
- Para eso se utiliza una nueva sintaxis <<<, >>> interponiendola entre el nombre del kernel y sus parámetros.

Llamado del kernel

```
// Invoke kernel
dim3 dim_block(BLOCK_SIZE);
dim3 dim_grid(DIV_CEIL(intervals, dim_block.x));

simpson_kernel<<<dim_grid, dim_block>>>(a, h, intervals, d_res);
cudaThreadSynchronize();

CUT_CHECK_ERROR("Kernel execution failed");
// End kernel invocation
```

¿Qué me queda para limpiar y cerrar?

```
CUDA_SAFE_CALL(cudaMemcpy(ss, d_res, (intervals+1)*sizeof(float),
                                     cudaMemcpyDeviceToHost));

// Free device memory
cudaFree(d_res);

simpson = thrust::reduce(ss, ss+intervals+1);

return h/3*simpson;
```

¿Qué necesito instalar para tener CUDA?

- Hacen falta sólo 2 cosas:
  - Driver de la placa de video **actualizado**
  - El toolkit de compilador, debugger, etc.
- Opcionalmente se puede instalar el SDK (Software Development Kit)
- El sitio de descarga es:
  http://developer.nvidia.com/object/cuda_2_3_downloads.html

Configurando mi entorno CUDA

- Una vez instalado el toolkit se debe configurar las variables de entorno para poder acceder a un nuevo conjunto de bibliotecas
- Hay que configurar LD_LIBRARY_PATH para que apunte a <CUDA_INSTALL_PATH>/lib

Una vez instalado y configurado todo

- La compilación de software se realiza con el compilador provisto en el toolkit (<CUDA_INSTALL_PATH>/bin/nvcc)
- Se puede compilar tanto código CUDA como código C común
- El código en C es compilado internamente con gcc