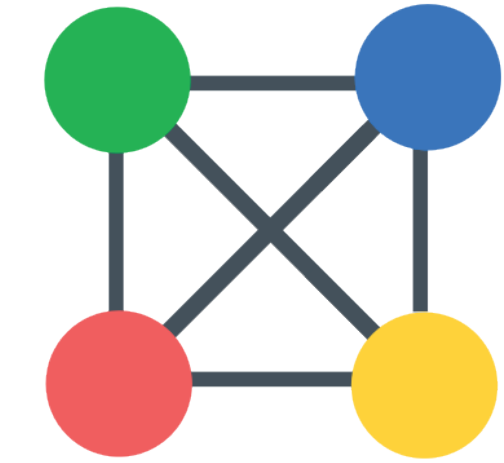
Herramientas para el Desarrollo de Programas Paralelos



MPI

NVIDIA®

CUDA®

Dra. Graciela Román Alonso
Dr. José Luis Quiróz Fabián
Dr. Miguel Alfonso Castro García
Dr. Manuel Aguilar Cornejo
Departamento de Ingeniería Eléctrica,
Universidad Autónoma Metropolitana-Iztapalapa

Resumen
Hoy por hoy existen diversas herramientas que permiten el desarrollo de programas paralelos. Cada una de estas se especializa en explotar al máximo las características de una arquitectura computacional particular. En este artículo se presentan tres de las herramientas más populares en el desarrollo de programas paralelos: OpenMP para programación en arquitecturas con memoria compartida, MPI para arquitecturas con memoria distribuida y CUDA para programación en tarjetas gráficas. Se describen las principales características de cada una de ellas usando un ejemplo de programación.

Palabras clave: OpenMP, MPI, CUDA, Cómputo Paralelo..

Abstract
Nowadays there are several tools that allow the development of parallel programs. Each of them specializes in fully exploiting the characteristics of a particular computational architecture. This article presents three of the most popular tools for developing parallel programs: OpenMP for programming on shared memory architectures, MPI for distributed memory architectures, and CUDA for programming on graphics cards. The main characteristics of each of them are described, showing a programming example.

Keywords: OpenMP, MPI, CUDA, Parallel Computing..

Introducción

En el día a día es común tener acceso a equipos multi procesador (multi-cores), redes de computadoras o tarjetas gráficas (GPU). Estas arquitecturas tienen un papel fundamental en la solución de problemas computacionales debido al alto poder de procesamiento que ofrecen al ejecutar programas paralelos gracias a lo cual diferentes partes de un programa pueden ser ejecutadas simultáneamente. Para programar estas arquitecturas se tienen en el mercado diversas herramientas que toman en cuenta las diferentes características de los equipos. En este artículo se describen 3 de las herramientas más populares para realizar programas paralelos: OpenMP (Open Multi -Processing) para sistemas con memoria compartida, MPI (Message Passing Interface) para arquitecturas con memoria distribuida y CUDA (Compute Unified Device Architecture) para la programación de tarjetas gráficas. Dichas herramientas se integran como extensiones de algún lenguaje de programación ya existente, en este documento usaremos las versiones asociadas al lenguaje C. Para cada herramienta se muestran sus principales primitivas de paralelismo usando el mismo ejemplo del cálculo del área bajo una curva.

Paralelización de un Programa

A diferencia de un programa secuencial, donde únicamente se requiere del conocimiento del problema a resolver y del lenguaje de programación con que se implementará, los programas paralelos requieren además del establecimiento de tres pasos. El primero se refiere al particionamiento de la solución identificando las tareas que pueden ejecutarse simultáneamente. Después debe especificarse si se requiere comunicación o sincronización entre las tareas. El último aspecto tiene que ver con la asignación de las tareas en los procesadores disponibles.

El conocimiento de las características de la arquitectura paralela con que se cuenta

puede guiar la manera de diseñar el particionamiento y la asignación de tareas, a fin de obtener un mejor rendimiento. Este aspecto por ahora no cae dentro del ámbito de este documento, sin embargo mostraremos cómo se pueden desarrollar los pasos de paralelización usando las tres herramientas OpenMP, MPI y CUDA.

Programación con OpenMP

OpenMP es una interfaz de programación de aplicaciones (API) para la programación de memoria compartida[1] en ambientes multi-núcleo [1]. Permite crear varios flujos de ejecución (hilos) en programas escritos en C, C ++ y Fortran mediante un conjunto de directivas que comienza con **#pragma omp**. Está disponible en muchas arquitecturas, incluidas las plataformas de Linux, Unix y de Microsoft Windows.

Particionamiento

En OpenMP cada tarea se define mediante un hilo[2]. El particionamiento es posible definirlo mediante un conjunto de hilos que ejecutan las mismas instrucciones (código) o instrucciones diferentes sobre un conjunto de datos.

Dos de las directivas más populares de OpenMP para que los hilos ejecuten las mismas instrucciones son **#pragma omp parallel** y **#pragma omp parallel for**. Mediante **#pragma omp parallel** se crea un grupo de hilos y las instrucciones que van ejecutarse se encierran entre llaves (ver Figura 1).

Mediante la directiva **#pragma omp parallel for** las iteraciones de un ciclo **for** se dividen entre un grupo de hilos.

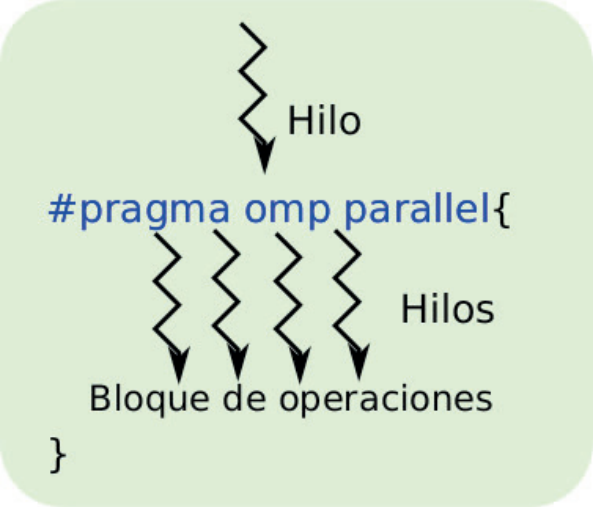


Fig 1: Definición de una región paralela en OpenMP.

Para que los hilos ejecuten instrucciones diferentes se definen secciones paralelas mediante **#pragma omp sections** y **#pragma omp section**. Cada directiva section define la sección de un hilo y se declara dentro de la directiva **sections**.

Comunicación y Sincronización

La comunicación en OpenMP es principalmente mediante variables compartidas o bien operaciones para obtener resultados globales. Para definir una variable compartida en las directivas de OpenMP se utilizan cláusulas[3]. La cláusula **shared** se utiliza para definir uno o más variables compartidas, por ejemplo **#pragma omp parallel shared(total,num)** define una región paralela con las variables compartidas **total** y **num**. Otra cláusula que se utiliza para comunicación es **reduction**, la cual permite aplicar una operación a un conjunto de variables (con el mismo nombre) que tienen los hilos.

[1] La memoria compartida es un bloque de memoria física que se comparte entre varios procesos.
[2] Flujo de ejecución dentro de un proceso.
[3] Modificadores de las directivas para establecer características propias de una región paralela.

OpenMP tiene diferentes directivas que permiten la sincronización de hilos. Algunas de estas son **#pragma omp critical** y **#pragma omp atomic**. La primera permite ejecutar un bloque de código en exclusión mutua, la segunda permite ejecutar una operación de forma atómica; es decir, no puede ser interrumpida por otro hilo.

Asignación de tareas
Por omisión, en una región paralela de OpenMP se generan tantos hilos como procesadores o núcleos se tienen en la máquina donde se ejecuta. Además, para la directiva **#pragma omp parallel for** es posible cambiar cómo se reparten las iteraciones los hilos por medio de la cláusula **schedule**.

Ejemplo del área bajo la curva en OpenMP
Aproximar el área bajo la curva de una función al sumar un número finito de rectángulos en la suma de Riemann puede obtener resultados muy exactos. Mientras más subintervalos se tomen, mejor es la aproximación. La Figura 2 muestra la representación del cálculo del área bajo la curva de la función *f(x)* donde se tienen N particiones o rectángulos de tamaño Δx.

La Figura 3 muestra la solución del cálculo del aŕea bajo la curva de la función $x^2 + 1$ en el intervalo de 1 a 3 usando OpenMP.

Particionamiento
Se utiliza la directiva **#pragma omp parallel for** para dividir el intervalo de 1 a 3 en 999,999,999 particiones las cuales se distribuyen entre los hilos en partes iguales.

Comunicación y sincronización
En este ejemplo, todos los hilos obtienen una suma parcial (la de su partición) y la almacenan en su propia variable suma (cada hilo tiene una variable suma). Al final se obtiene una suma global mediante la cláusula **reduction(+:suma)** sumando cada variable suma.

Asignación de tareas
En este ejemplo la asignación de tareas es la que tiene por omisión OpenMP, se crean tantos hilos como procesadores o núcleos.

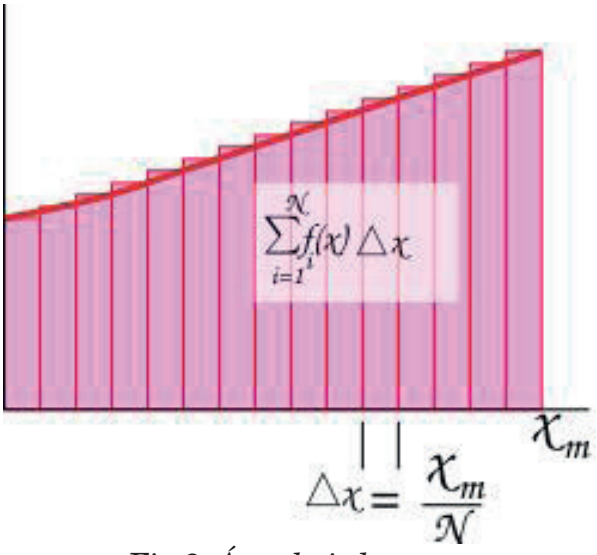


Fig 2: Área bajo la curva mediante sumas de Riemann

```
#include <math.h>
#include <stdio.h>
#define INICIO 1.0f
#define FIN 3.0f
#define PARTICIONES 999999999
#define DELTA ((FIN - INICIO) / PARTICIONES)
int main(){
      double suma = 0;
      int i;
      #pragma omp parallel for reduction(+:suma)
      for (i = 1; i <= PARTICIONES; i++){
            suma = suma + (DELTA * (pow((INICIO +
            (DELTA * i)),2) + 1));
      }
      printf("%f\n", suma);
      return 0;
}
```

Fig 3: Cálculo del área bajo la curva mediante OpenMP.

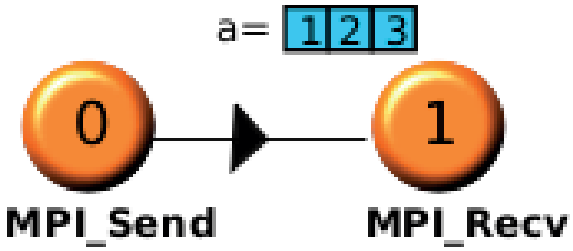


Fig 4: Comunicación punto a punto.

Programación con MPI

MPI (Message Passing Interface) [2] es una biblioteca de paso de mensajes diseñada para ser usada en arquitecturas de memoria distribuida. Los elementos principales que intervienen en el paso de mensajes son el proceso que envía, el que recibe y el mensaje. Existen múltiples implementaciones de la interfaz MPI que consisten en un conjunto de bibliotecas de rutinas que pueden ser utilizadas en programas escritos en los lenguajes de programación C, C++, y Fortran.

Particionamiento

En MPI cada tarea se define mediante un proceso. El particionamiento es posible definirlo mediante un conjunto de procesos que ejecutan las mismas instrucciones (código) o instrucciones diferentes sobre un conjunto de datos. Por omisión los procesos tienen el mismo código. Para que cada proceso tenga un código diferente se utiliza la primitiva **MPI_Comm_spawn**.

Comunicación y sincronización

MPI proporciona diferentes funciones primitivas para realizar la comunicación y sincronización de las tareas o procesos. La comunicación puede ser punto a punto (uno a uno) o bien de forma colectiva (por medio de un grupo de procesos).

Las operaciones más populares para realizar la comunicación punto a punto entre dos procesos son: **MPI_Send** y **MPI_Recv**. Estas primitivas permiten el envío de un arreglo de datos entre dos procesos. La Figura 4 muestra un ejemplo de la comunicación entre dos procesos (0 y 1), donde el proceso 0 le envía un arreglo de 3 enteros al proceso 1.

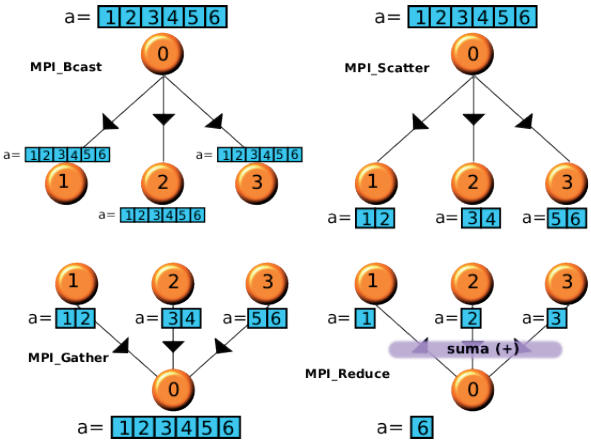


Fig 5: Operaciones colectivas en MPI.

Las operaciones colectivas de MPI permiten la comunicación entre un grupo de procesos. Algunas operaciones colectivas más populares en MPI son: MPI_Bcast, MPI_Scatter, MPI_Gather y MPI_Reduce, entre otras. **MPI_Bcast** permite difundir un mensaje entre un conjunto de procesos. **MPI_Scatter** permite dispersar un mensaje entre un conjunto de procesos. **MPI_Gather** permite recolectar mensajes entre un conjunto de procesos y **MPI_Reduce** permite la recolección de mensajes y aplicarles una operación para obtener un único dato. La Figura 5 muestra un ejemplo de estas operaciones colectivas donde intervienen 4 procesos y el mensaje son arreglos. El proceso 0 difunde un arreglo de 6 elementos, El proceso 0 dispersa un arreglo de 6 elementos, el proceso 0 recolecta arreglo de dos elementos de cada proceso y el proceso 0 recolecta 3 datos de cada proceso y les aplica la operación suma (+) para obtener un dato.

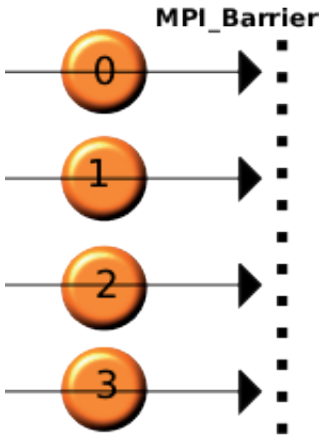


Fig 6: Sincronización por barreras.

Respecto a la sincronización, una de las operaciones más populares en MPI es **MPI_Barrier**. Esta primitiva permite bloquear a un grupo de procesos hasta que todos ejecuten la primitiva. La Figura 6 muestra un ejemplo de esta primitiva donde intervienen 4 procesos.

Asignación de tareas

Como se mencionó previamente, MPI es un estándar y existen diferentes implementaciones de la interfaz. Dependiendo de la implementación se tienen diferentes maneras de asignar las tareas o procesos a una computadora. Algunas de las políticas más populares son: un proceso por cada unidad de procesamiento (CPU). Si se tienen N máquinas cada una con K CPUs y se requiere asignar M procesos, la asignación es por rondas: las máquinas se ordenan y primero se llenan los K CPUS's de una máquina, después los K CPU's de la otra y así sucesivamente hasta contemplar los M procesos. O bien, se asigna un proceso por cada máquina y si faltan procesos por asignar, se realiza una repartición nuevamente entre todas las máquinas hasta contemplar los M procesos.

```
#include <stdio.h>
#include <omp.h>
#include "mpi.h"
#include <math.h>

#define INICIO 1.0f
#define FIN 3.0f
#define PARTICIONES 999999999
#define DELTA ((FIN-INICIO)/PARTICIONES)
int main(int argc, char *argv[]) {
 int rank, nprocs, leng,inicio, fin;
 int bloques, residuo, adicional,i;
 double suma=0,sumal=0;
 MPI_Init (&argc,&argv);
 MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
 MPI_Comm_rank (MPI_COMM_WORLD, &rank);
 bloques=PARTICIONES/nprocs;
 residuo=PARTICIONES%nprocs;
 inicio=rank*bloques+1;
 fin=inicio+bloques;
 for(i=inicio; i<fin; i++){
  sumal=sumal+(DELTA*(pow((INICIO+(DELTA*i)),2)+1));
 }
 MPI_Reduce(&sumal, &suma,1,MPI_DOUBLE, MPI_SUM, 0,
 MPI_COMM_WORLD);
 if(rank==0)
  printf("Suma %f\n",suma );
 MPI_Finalize();
}
```

Fig 7: Cálculo del área bajo la curva mediante MPI.

Ejemplo del área bajo la curva en MPI

La Figura 7 muestra la solución del área bajo curva en MPI.

Particionamiento

En este problema los procesos se dividen las particiones (sobre el rango donde se desea calcular el área) entre el total de procesos. Todos los procesos comparten el mismo código pero diferentes datos.

Comunicación y Sincronización

Cuando cada proceso termina de realizar las operaciones sobre datos (sub-rango que le correspondió) obtiene una suma parcial (área bajo la curva en su sub-rango), para obtener un solo resultado (área bajo la curva sobre todo el rango) el proceso 0 realiza una operación MPI_Reduce para sumar cada resultado de cada proceso en uno solo.

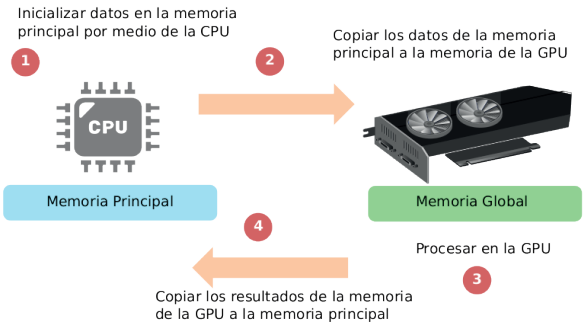


Fig 8: Programación en CUDA.

Asignación de tareas

La asignación de los procesos a los CPU's para este problema solo requiere que un proceso se asigne a uno y solo un CPU para evitar pérdida de rendimiento.

Programación en Tarjetas Gráficas mediante CUDA

CUDA (Compute Unified Device Architecture) [3] es una arquitectura de cálculo paralelo que permite a los programadores usar el lenguaje de programación C para codificar algoritmos en GPU's (unidades de procesamiento gráfico de la empresa Nvidia). Las aplicaciones idóneas para programar en CUDA son las que procesan grandes cantidades de datos independientes. CUDA cuenta con un compilador y con un conjunto de herramientas de desarrollo creadas por Nvidia. La GPU se conforma por varios multiprocesadores y cada multiprocesador de varios núcleos.

Programación

La programación en CUDA se realiza en 4 pasos, como lo muestra la Figura 8. En el primer paso, se inicializan los datos a procesar en la memoria RAM. En el segundo paso se copian los datos de la memoria RAM a la memoria de la GPU, la cual se conoce como memoria global. En el tercer paso los datos son procesados en paralelo por los núcleos de la GPU. Finalmente el contenido de la memoria de la GPU se copia a la memoria RAM.

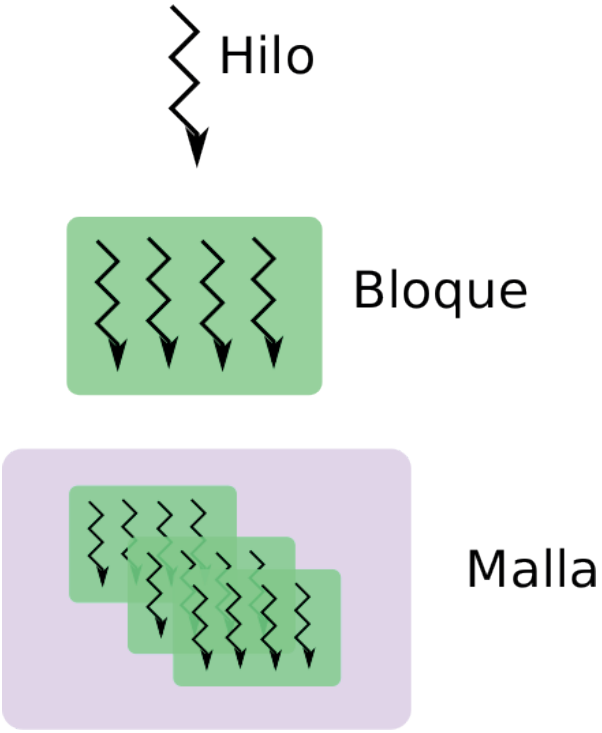


Fig 9: Organización de los hilos

Organización de los hilos

En CUDA se define una función conocida como **kernel**. Cuando se invoca esta función, en la GPU se crea una **malla** o **grid** de hilos que ejecutarán el kernel simultáneamente. La malla puede ser de una, dos o tres dimensiones. Los hilos dentro de la malla se organizan en **bloques**, que también pueden ser de una, dos o tres dimensiones (ver Figura 9).

Particionamiento

Por lo descrito anteriormente, el particionamiento en CUDA es por medio de bloques e hilos, donde los hilos deben preferentemente ejecutar el mismo código al mismo ritmo.

Comunicación y sincronización

La comunicación de los hilos en CUDA se da principalmente mediante el uso de la memoria de la GPU, la cual se conoce como memoria global. Todos los hilos pueden leer y escribir las variables en la memoria global. Respecto a la sincronización, en CUDA mediante **syncthreads()** solo los hilos de un mismo bloque se pueden sincronizar para detenerse en un punto particular del código.

Asignación de tareas

Los hilos de un mismo bloque en CUDA se ejecutan en uno y solo un microprocesador. Si hay más hilos en un bloque que núcleos en un multiprocesador, primero se ejecuta una parte de los hilos y después las otras.

Ejemplo: Área bajo la curva en CUDA

La Figura 10 muestra la programación en CUDA del problema del cálculo del área bajo la curva.

Particionamiento

Al igual que en OpenMP, se crea un conjunto de hilos que reparten las particiones del rango de 1 a 3. La cantidad de hilos que se van a crear se define en la función principal (main) usando los modificadores **dim3**. En este ejemplo se crean 112 bloques (28x4x1). Cada bloque tiene 32 hilos (32x1x1). En total se tienen 3584

Comunicación y Sincronización

En este ejemplo, la operación **cudaMalloc** permite reservar memoria en la GPU para guardar los resultados de cada hilo. Escribiendo **programa<<grid,block>>** se invoca al kernel (los modificadores **__gobal__ void** se usan para definir el kernel). Finalmente, el copiado de resultados de la GPU a la CPU se realiza mediante **cudaMemcpy**.

```
#include <stdio.h>
#define INICIO 1.0f
#define FIN 3.0f
#define PARTICIONES 1999999999
#define DELTA ((FIN - INICIO) / PARTICIONES)

__global__ void programa(double *datos, int total, double ini,
int particiones,double delta){
   int idHiloBloque= ((threadIdx.x * blockDim.y) + threadIdx.y) +
   (blockDim.x*blockDim.y)*threadIdx.z;
   int idBloqueGrid =  ((blockIdx.x * gridDim.y) + blockIdx.y) +
   (gridDim.x*gridDim.y)*blockIdx.z;
   int id = idBloqueGrid*(blockDim.x*blockDim.y*blockDim.z) +
   idHiloBloque;
   int i,bloques,residuo,inicio,fin,adicional;
   double sumal=0;

   bloques=particiones/total;
   residuo=particiones%total;
   inicio=id*bloques+1;
   fin=inicio+bloques;
   if(residuo>0 && id<residuo){
     adicional=particiones-id;
     sumal = sumal + (delta * (pow((ini + (delta * adicional)),2) + 1));
   }
   for (i = inicio; i < fin; i++){
     sumal = sumal + (delta * (pow((ini + (delta * i)),2) + 1));
   }
   datos[id]=sumal;
}

int main(){
  dim3 grid(28,4,1);
  dim3 block(32,1,1);
  double *sumas,*sumas_gpu,sumag=0.0;
  int i,tam=3584;
  sumas = (double*) malloc(sizeof(double)*tam);
  cudaMalloc((void**)&sumas_gpu, sizeof(double)*tam);
  programa<<<grid, block>>>(sumas_gpu,tam,INICIO,PARTICIONES,DELTA);
  cudaMemcpy(sumas, sumas_gpu, sizeof(double)*tam,cudaMemcpyDeviceToHost);
  for(i=0;i<tam;i++)
    sumag=sumag+sumas[i];
  printf("Area: %lf\n",sumag);
  return 0;
}
```

Fig 10: Ejemplo del área en CUDA.

Asignación de tareas

La asignación en este ejemplo es un hilo por cada núcleo que tiene la GPU.

Conclusiones

En este trabajo se presentaron tres herramientas de programación paralela para diferentes sistemas computacionales. OpenMP es recomendable en arquitecturas multi-núcleo, MPI en ambientes de memoria distribuida y CUDA en tarjetas gráficas de Nvidia. Además de las generalidades de cada una de estas herramientas se presentó un ejemplo del cálculo del área bajo la curva programado en las tres diferentes herramientas. En este artículo no se consideraron arquitecturas híbridas (por ejemplo redes de equipos multi-núcleo con tarjetas gráficas); no obstante, las herramientas presentadas pueden

combinarse como programación híbrida, MPI+OpenMP+CUDA, a fin de explotar dicha infraestructura. Tampoco se presenta una comparación de rendimiento debido a que el resultado depende de la arquitectura física (número de núcleos en el CPU para OpenMP, número de máquinas en MPI y el modelo de la tarjeta gráfica).

Referencias

[1] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. 2001. Parallel programming in OpenMP. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[2] William Gropp, Ewing Lusk, and Anthony Skjellum. 2014. Using MPI: Portable Parallel Programming with the Message-Passing Interface. The MIT Press.

[3] Shane Cook. 2012. CUDA Programming: A Developer's Guide to Parallel Computing with GPUs (1st. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.