Universidad Nacional de San Antonio Abad del Cusco

Departamento Académico de Informática

ALGORITMOS PARALELOS Y DISTRIBUIDOS

Práctica Nº 8

TIPOS DE MEMORIA EN CUDA

1. OBJETIVO.

- Conocer el uso de la memoria constante y la memoria compartida.
- Conocer algoritmos de reducción.

2. INTRODUCCION.

2.1. Tipos de Memoria.

Cada dispositivo CUDA tiene varias memorias que pueden ser usadas para aumentar la eficiencia en la ejecución. Se define el CGMA como la razón entre operaciones realizadas y accesos a la memoria Global que se han realizado. El objetivo es hacerlo lo mayor posible; es decir: realizar el mayor número de operaciones aritméticas con el menor número de accesos a la memoria Global. En la figura 1 se representa el modelo de distribución de memorias particular para el modelo GeForce 8800 GTX. En la parte inferior tenemos la memoria Global y la Constante, ambas son accesibles por parte del Host en modo lectura-escritura. La memoria constante sólo es accesible en modo lectura por el dispositivo, sin embargo, ofrece accesos más rápidos y más paralelos que la global. Por encima de los bloques correspondientes a los hilos encontramos los registros y las memorias compartidas. Los datos almacenados en ambos tipos son accesibles de manera muy rápida. Los registros son asignados a hilos de manera individual; cada hilo sólo puede acceder a sus propios registros. Una función kernel usará los registros para almacenar datos que son usados frecuentemente, pero que son particulares para cada hilo. Las memorias compartidas (shared) son asignadas a bloques de hilos; todos los hilos de un bloque pueden acceder a datos en la memoria compartida. El uso de la memoria compartida es esencial para que los hilos cooperen compartiendo datos.

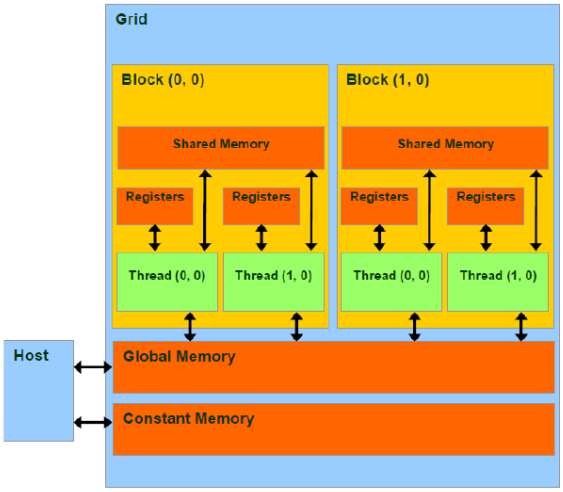
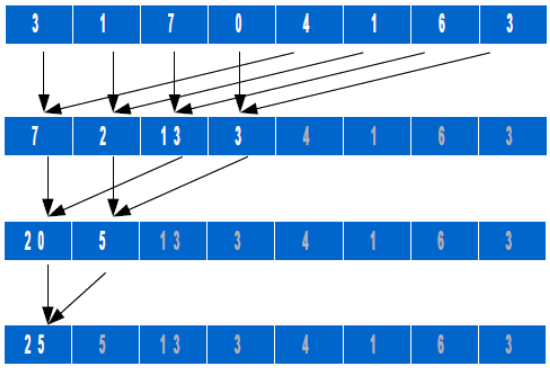


Fig. 1. Tipos de memoria

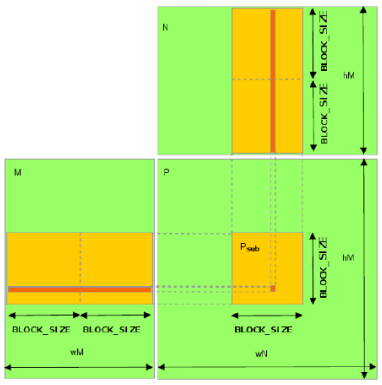
2.2. Reducción.

Una reducción es una combinación de todos los elementos de un vector en un valor único, utilizando para ello algún tipo de operador asociativo. Las implementaciones paralelas aprovechan esta asociatividad para calcular operaciones en paralelo, calculando el resultado en O(logN) pasos sin incrementar el número de operaciones realizadas. Un ejemplo de este tipo de operación se muestra en la siguiente figura:



2.3. Multiplicación de matrices.

- Producto de MxN.
- Cada hilo computa un elemento de P.



2.4. Cálculo de Pi.

De forma aproximada se puede realizar el cálculo del valor del número π utilizando la siguiente relación entre el número π y la serie formada por los inversos de los cuadrados de los números naturales:

$$\frac{\pi^2}{6} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots + \frac{1}{n^2} + \cdots$$

3. DESARROLLO DE LA PRÁCTICA

3.1. Ejercicio 1.

En el siguiente programa se implementa un algoritmo de reducción que utiliza memoria compartida

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>

// defines
#define N 16
// kernel
__global__ void reduccion(float* vector, float* suma)
{
	// Reserva de espacio en la zona de memoria compartida
	__shared__ float temporal[N];
	// Indice local de cada hilo -> kernel con un solo bloque
	int myID = threadIdx.x;
	// Copiamos en 'temporal' el vector y sincronizamos
	temporal[myID] = vector[myID];
	__syncthreads();

	//Reduccion paralela
	int salto = N / 2;
	// Realizamos log2(N) iteraciones
	while (salto)
	{
		// Solo trabajan la mitad de los hilos
		if (myID < salto)
		{
			temporal[myID] = temporal[myID] + temporal[myID + salto];
		}
		__syncthreads();
		salto = salto / 2;
	}
	// El hilo no.'0' escribe el resultado final en la memoria global
	if (myID == 0)
	{
		suma[0] = temporal[myID];
	}
}

int main()
{
	// declaraciones
	float* vector1;
	float* dev_vector1;

	float* suma;
	float* dev_suma;
	// reserva en el host
```

```
        vector1 = (float*)malloc(N * sizeof(float));
        suma = (float*)malloc(sizeof(float));
        // reserva en el device
        cudaMalloc((void**)&dev_vector1, N * sizeof(float));
        cudaMalloc((void**)&dev_suma, sizeof(float));
        // inicializacion de vectores
        for (int i = 0; i < N; i++)
        {
            vector1[i] = (float)i + 1;

        }

        // copia de datos hacia el device
        cudaMemcpy(dev_vector1, vector1, N * sizeof(float),
                                            cudaMemcpyHostToDevice);

        // lanzamiento del kernel
        printf("Vector de %d elementos\n", N);
        reduccion << < 1, 16 >> > (dev_vector1, dev_suma);

        // recogida de datos desde el device
        cudaMemcpy(suma, dev_suma, sizeof(float), cudaMemcpyDeviceToHost);

        // impresion de resultados
        printf("Resultado: %.2f\n", suma[0]);

        // liberamos memoria en el device
        cudaFree(dev_vector1);
        cudaFree(suma);
        // salida
        printf("\npulsa INTRO para finalizar...");
        fflush(stdin);
        char tecla = getchar();
        return 0;
    }
```

3.2. Ejercicio 2.

En el siguiente ejemplo se utiliza la memoria constante

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>
#define N 8
// CUDA constants
__constant__ float dev_A[N][N];
// declaracion de funciones
// GLOBAL: funcion llamada desde el host y ejecutada en el device (kernel)

__global__ void traspuesta(float* dev_B)
{ // kernel lanzado con un solo bloque y NxN hilos
    int columna = threadIdx.x;
    int fila = threadIdx.y;
    int pos = columna + N * fila;

    // cada hilo coloca un elemento de la matriz final
    dev_B[pos] = dev_A[columna][fila];
}

// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv) {
    // declaraciones
    float* hst_A, * hst_B;
    float* dev_B;
    // reserva en el host
```

```
hst_A = (float*)malloc(N * N * sizeof(float));
hst_B = (float*)malloc(N * N * sizeof(float));
// reserva en el device
cudaMalloc((void**)&dev_B, N * N * sizeof(float));
// inicializacion
for (int i = 0; i < N * N; i++)
{
    hst_A[i] = (float)i;
}
// copia de datos
cudaMemcpyToSymbol(dev_A, hst_A, N * N * sizeof(float));
// dimensiones del kernel
dim3 Nbloques(1);
dim3 hilosB(N, N);
// llamada al kernel bidimensional de NxN hilos
traspuesta <<< Nbloques, hilosB >>> (dev_B);
// recogida de datos
cudaMemcpy(hst_B, dev_B, N * N * sizeof(float),
    cudaMemcpyDeviceToHost);

// impresion de resultados  printf("Resultado:\n");  printf("ORIGINAL:\n");
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        printf("%2.0f ", hst_A[j + i * N]);
    }
    printf("\n");
}

printf("TRASPUESTA:\n");
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        printf("%2.0f ", hst_B[j + i * N]);
    }
    printf("\n");

}
```

3.3. Ejercicio 3.

En siguiente programa multiplica dos matrices utilizando memoria compartida.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#define N 4

// ***** Sección de código del device *****
__global__ void MatMultGPU(int* a, int* b, int* c)
{
	int k, sum = 0;
	int col = threadIdx.x + blockDim.x * blockIdx.x;
	int fil = threadIdx.y + blockDim.y * blockIdx.y;

	if (col < N && fil < N)
	{
		for (k = 0; k < N; k++)
		{
			sum += a[fil * N + k] * b[k * N + col];
		}
		c[fil * N + col] = sum;
	}
}

// ***** Sección de código del host *****
```

```
// ===== Código para mostrar los elementos de la matriz
void MatMostrar(char* Titulo, int a[N][N])
{
	printf("\n %s \n", Titulo);
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
			printf("%d ", a[i][j]);
		printf("\n");
	}
}

int main()
{
	// -- Declarar variables
	int h_a[N][N], h_b[N][N], h_c[N][N];
	int* d_a, * d_b, * d_c;
	int cont;

	// -- Inicializar variables con datos
	for (int i = 0; i < N; i++)
	{
		cont = 0;
		for (int j = 0; j < N; j++)
		{
			h_a[i][j] = cont;
			h_b[i][j] = cont;
			cont++;
		}
	}

	int size = N * N * sizeof(int);

	cudaMalloc((void**)&d_a, size);
	cudaMalloc((void**)&d_b, size);
	cudaMalloc((void**)&d_c, size);

	cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
	cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

	dim3 dimGrid(1, 1);
	dim3 dimBlock(N, N);

	MatMultGPU << < dimGrid, dimBlock >> > (d_a, d_b, d_c);

	cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

	cudaFree(d_a);
	cudaFree(d_b);
	cudaFree(d_c);

	// imprimiendo
	MatMostrar("Matriz A", h_a);
	MatMostrar("Matriz B", h_b);
	MatMostrar("Matriz C", h_c);

	// -- Pausa antes de salir
	printf("\npulsa INTRO para finalizar...");
	fflush(stdin);
	char tecla = getchar();

	return 0;

}
```

3.4. Ejercicio 4.

Calculo de Pi.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// . . .
// defines
#define N 512
// kernel
__global__ void pi(float* suma)
{
	// Reserva de espacio en la zona de memoria compartida
	__shared__ float temporal[N];
	// Indice local de cada hilo -> kernel con un solo bloque
	int myID = threadIdx.x;
	// Generamos en 'temporal' el valor de la serie
	int D = myID + 1;
	temporal[myID] = 1.0 / (D * D);
	__syncthreads();
	//Reduccion paralela
	int salto = N / 2;
	// Realizamos log2(N) iteraciones
	while (salto)
	{
		// Solo trabajan la mitad de los hilos
		if (myID < salto)
		{
			temporal[myID] = temporal[myID] + temporal[myID + salto];
		}
		__syncthreads();
		salto = salto / 2;
	}
	// El hilo no.'0' escribe el resultado final en la memoria global
	if (myID == 0)
	{
		suma[0] = temporal[myID];
	}
}


int main()
{
	// declaraciones
	float* suma;
	float* dev_suma;
	// reserva en el host
	suma = (float*)malloc(sizeof(float));
	// reserva en el device
	cudaMalloc((void**)&dev_suma, sizeof(float));

	// lanzamiento del kernel
	printf("Calcular el valor de Pi en base %d términos\n", N);
	pi << < 1, N >> > (dev_suma);

	// recogida de datos desde el device
	cudaMemcpy(suma, dev_suma, sizeof(float), cudaMemcpyDeviceToHost);

	// calcular el  valor de pi
	float ValorPi = sqrt(suma[0] * 6);
	// impresion de resultados
```

```
        printf("Resultado: %.2f\n", ValorPi);

        // liberamos memoria en el device
        cudaFree(suma);
        // salida
        printf("\npulsa INTRO para finalizar...");
        fflush(stdin);
        char tecla = getchar();
        return 0;
}
```

1. TAREA

1.1. Considerar un array $a = (a_0, a_1, \ldots, a_{n-1})$ de números enteros. Construir un kernel *invierte_array* que reciba a y devuelva $b = (a_{n-1}; a_{n-2}, \ldots, a_1, a_0)$.

¿Cuál es el máximo tamaño que puede tener el vector? ¿Por qué?

1.2. Escribir un programa paralelo para calcular el valor de la siguiente integral.

$$\int_0^1 e^{-x^2} dx$$

2. BIBLIOGRAFIA.

- César Represa Pérez. (2016) Introducción a la Programación en CUDA. Universidad de Burgos.
- Rob Farber. (2011). CUDA Application Design and Development. Ed. Elsevier
- Jason Sanders. (2011) CUDA By Example. Addison Wesley.
- John Cheng, Max Grossman, Ty McKercher (2014) Professional CUDA Programming. Ed. Wrox.