**UNIVERSIDAD MAYOR DE SAN ANDRÉS**

**FACULTAD DE CIENCIAS PURAS Y NATURALES**

**CARRERA DE INFORMÁTICA**



**PROYECTO INF-317**

**PROGRAMACIÓN PARALELA Y DISTRIBUIDA**

**ASIGNACIÓN DE MEMORIA EN GPU Y SU IMPACTO EN LA PROGRAMACIÓN PARALELA**

**INTEGRANTES:**

* UNIV. Castillo M. Roxana
* UNIV. Quispe Ch. Miguel Matheus
* UNIV. MITA Ramírez victor Manuel

LA PAZ – BOLIVIA

2024

## Introducción

Las GPUs (Unidades de Procesamiento Gráfico) han evolucionado desde ser dispositivos exclusivamente para renderizado gráfico hasta convertirse en plataformas de cálculo de alto rendimiento. Uno de los elementos clave que permite este rendimiento es su arquitectura de memoria, diseñada para soportar un altísimo grado de paralelismo. Este informe explora las diferentes jerarquías de memoria en las GPUs, su impacto en la programación paralela, una comparativa con las memorias de CPU y PC, y los mecanismos de asignación de memoria en GPUs.

## Tipos de Memoria en GPUs

### 1. Memoria Global

* **Descripción:**
  + Es la memoria principal de la GPU, compartida por todos los bloques y hilos.
  + Tiene alta capacidad pero también una latencia considerable.
* **Ventajas:**
  + Gran capacidad de almacenamiento.
  + Accesible por todos los hilos de un kernel.
* **Limitaciones:**
  + Acceso lento si no está coalescente.
  + Requiere optimización para evitar “bottlenecks”.

### 2. Memoria Compartida

* **Descripción:**
  + Una región de memoria localizada en el SM (Multiprocesador Escalar) y compartida entre todos los hilos de un bloque.
* **Ventajas:**
  + Latencia baja y rendimiento similar al de registros.
  + Muy eficiente para comunicación entre hilos del mismo bloque.
* **Limitaciones:**
  + Espacio limitado (normalmente unos pocos kB).
  + Solo accesible dentro del bloque.

### 3. Memoria de Registros

* **Descripción:**
  + Memoria más rápida y localizada dentro de cada hilo.
  + Se utiliza para almacenar variables locales.
* **Ventajas:**
  + Acceso extremadamente rápido.
* **Limitaciones:**
  + Espacio muy limitado.
  + El “spillover” a la memoria local puede reducir el rendimiento.

### 4. Memoria Constante

* **Descripción:**
  + Memoria de solo lectura optimizada para accesos repetidos a valores constantes.
* **Ventajas:**
  + Alta eficiencia para datos que no cambian durante la ejecución.
* **Limitaciones:**
  + Espacio limitado (normalmente 64 kB).

### 5. Memoria Local

* **Descripción:**
  + Una extensión de la memoria global, utilizada para datos que no caben en los registros.
* **Ventajas:**
  + Puede almacenar datos temporales.
* **Limitaciones:**
  + Acceso lento, ya que se encuentra en la memoria global.

### 6. Memoria de Texture y Surface

* **Descripción:**
  + Diseñada para acceder a datos de manera irregular.
  + Beneficiada por cachés optimizados para patrones de acceso dispersos.
* **Ventajas:**
  + Ideal para datos gráficos o complejas transformaciones.

## Impacto en la Programación Paralela

El diseño de la memoria afecta significativamente el rendimiento de las aplicaciones paralelas:

1. **Coalescencia de Memoria Global:**
   * La eficiencia de la memoria global depende de que los accesos sean “coalescentes”, es decir, que los hilos accedan a direcciones contiguas.
2. **Uso de Memoria Compartida:**
   * Reduce el uso de memoria global al permitir que los hilos compartan datos localmente.
   * Es crucial para la optimización de algoritmos como multiplicación de matrices.
3. **Minimización de Accesos:**
   * Los accesos frecuentes a memoria global pueden convertirse en cuellos de botella, por lo que se prioriza el uso de registros y memoria compartida.
4. **Sincronización:**
   * La comunicación entre hilos de diferentes bloques es costosa, ya que implica accesos a memoria global.

## Comparativa con Memorias de CPU y GPU

|  |  |  |
| --- | --- | --- |
| **Característica** | **Memoria en GPUs** | **Memoria en CPUs** |
| **Latencia** | Alta (memoria global), baja (registros y compartida) | Generalmente baja (RAM + Cachés). |
| **Ancho de banda** | Muy alto para operaciones paralelas | Alto, pero diseñado para tareas secuenciales. |
| **Jerarquía** | Global, compartida, registros, constante, local. | RAM, Caché L1/L2/L3, registros. |
| **Capacidad** | Alta en global, limitada en compartida y registros. | RAM mucho mayor, pero menos registros. |
| **Optimizaciones** | Necesita coalescencia y sincronización. | Menos dependiente de patrones de acceso. |

## Algoritmos de Asignación de Memoria en GPU

Los algoritmos de asignación de memoria en GPUs son fundamentales para optimizar el rendimiento de las aplicaciones que aprovechan el paralelismo masivo de estas unidades de procesamiento. Estos algoritmos determinan cómo se distribuyen los datos en la memoria de la GPU y cómo se acceden a ellos por parte de los núcleos de la GPU.

**La elección del algoritmo adecuado depende de diversos factores, como:**

* **Tamaño de los datos:** La cantidad de datos a procesar influye en la forma en que se distribuyen en la memoria.
* **Patrones de acceso a los datos:** La forma en que los hilos de la GPU acceden a los datos determina la eficiencia de la caché y la memoria.
* **Arquitectura de la GPU:** Las diferentes arquitecturas de GPU tienen características y limitaciones específicas que afectan la elección del algoritmo.
* **Tipo de aplicación:** Diferentes aplicaciones tienen requisitos de memoria distintos.

A continuación, se presentan algunos de los algoritmos y técnicas más comunes:

### Algoritmos Básicos

* **Asignación Contigua:** La memoria se asigna en bloques contiguos, lo que simplifica la gestión pero puede no ser óptimo para todos los patrones de acceso.
* **Asignación por Paginación:** La memoria se divide en páginas de tamaño fijo, lo que permite una gestión más flexible pero introduce sobrecarga por gestión de tablas de páginas.

### Técnicas de Optimización

* **Coalescencia de Memoria:** Los hilos de un warp (grupo de hilos) deben acceder a elementos contiguos de la memoria global para maximizar el ancho de banda.
* **Caché:** Las GPUs tienen una jerarquía de caché que puede ser explotada para reducir la latencia de acceso a la memoria.
* **Memoria Compartida:** La memoria compartida se utiliza para compartir datos entre los hilos de un bloque, lo que puede mejorar significativamente el rendimiento en ciertas operaciones.
* **Texturas:** La memoria de textura se utiliza para almacenar datos que se acceden de forma regular y ordenada, como imágenes.

### Algoritmos Avanzados

* **Asignación Basada en Colocación:** Los datos se colocan en la memoria de manera que minimicen la fragmentación y maximicen la localidad.
* **Asignación Dinámica:** La memoria se asigna y libera dinámicamente durante la ejecución del programa, lo que permite una mayor flexibilidad pero puede introducir sobrecarga.
* **Asignación Jerárquica:** Se utilizan múltiples niveles de jerarquía de memoria para optimizar el acceso a los datos.

### Consideraciones Adicionales

* **Fragmentación de Memoria:** La fragmentación ocurre cuando la memoria se asigna y libera de forma aleatoria, lo que puede llevar a una disminución del rendimiento.
* **Over-subscription de Memoria:** Cuando se asigna más memoria de la que está disponible físicamente, la GPU utiliza una técnica llamada swapping para mover los datos entre la memoria principal y la memoria de la GPU.
* **Gestión de Memoria Virtual:** Algunas GPUs soportan memoria virtual, lo que permite asignar más memoria de la que está físicamente disponible.

### Implementaciones en Frameworks

* **CUDA:** Proporciona funciones como cudaMalloc, cudaFree, y cudaMemcpy para la gestión de memoria.
* **OpenCL:** Ofrece una API similar a CUDA para la programación de GPUs.
* **HIP:** Permite escribir código portátil que se puede compilar tanto para GPUs de AMD como de NVIDIA.

## Asignación de Memoria en GPUs con CUDA

Link: https://colab.research.google.com/drive/1zBQMI8VSPGjiWAeDcaLnNtMpn\_v8Adt2?usp=sharing

Codigo:

%%writefile gpu.cu

#include <iostream>

#include <iomanip>

#include <cuda\_runtime.h>

#include <chrono>

\_\_global\_\_ void sumaVectorGPU(const float\* A, const float\* B, float\* C, int N) {

    int i = blockIdx.x \* blockDim.x + threadIdx.x;

    if (i < N) {

        C[i] = A[i] + B[i];

    }

}

void printVector(const float\* vec, int N, int limit = 10) {

    std::cout << "Vector (mostrando primeros y ultimos " << limit << " elementos):\n";

    for (int i = 0; i < limit && i < N; ++i) {

        std::cout << vec[i] << " ";

    }

    if (N > 2 \* limit) {

        std::cout << "... ";

        for (int i = N - limit; i < N; ++i) {

            std::cout << vec[i] << " ";

        }

    }

    std::cout << "\n";

}

int main() {

    const int N = 10000000;

    size\_t size = N \* sizeof(float);

    // Asignación de memoria en host

    float\* h\_A = new float[N];

    float\* h\_B = new float[N];

    float\* h\_C = new float[N];

    // Inicialización de vectores

    for (int i = 0; i < N; ++i) {

        h\_A[i] = 1.0f;

        h\_B[i] = 2.0f;

    }

    // Asignación de memoria en GPU

    float \*d\_A, \*d\_B, \*d\_C;

    cudaMalloc((void\*\*)&d\_A, size);

    cudaMalloc((void\*\*)&d\_B, size);

    cudaMalloc((void\*\*)&d\_C, size);

    // Copia de datos al dispositivo

    cudaMemcpy(d\_A, h\_A, size, cudaMemcpyHostToDevice);

    cudaMemcpy(d\_B, h\_B, size, cudaMemcpyHostToDevice);

    // Configuración del kernel

    int hilosPorBloque = 256;

    int bloquesPorGrilla = (N + hilosPorBloque - 1) / hilosPorBloque;

    // Ejecución del kernel

    auto inicio = std::chrono::high\_resolution\_clock::now();

    sumaVectorGPU<<<bloquesPorGrilla, hilosPorBloque>>>(d\_A, d\_B, d\_C, N);

    cudaDeviceSynchronize();

    auto fin = std::chrono::high\_resolution\_clock::now();

    // Copia del resultado al host

    cudaMemcpy(h\_C, d\_C, size, cudaMemcpyDeviceToHost);

    // Imprimir resultados

    printVector(h\_C, N);

    // Limpieza

    cudaFree(d\_A);

    cudaFree(d\_B);

    cudaFree(d\_C);

    delete[] h\_A;

    delete[] h\_B;

    delete[] h\_C;

    // Tiempo de ejecución

    std::chrono::duration<double> elapsed = fin - inicio;

    std::cout << std::fixed << std::setprecision(10); // Configuración para salida

    std::cout << "Tiempo en GPU: " << elapsed.count() << " segundos\n";

    return 0;

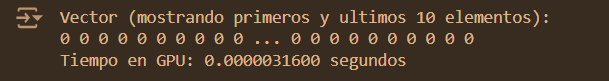
}



%%bash

nvcc -std=c++17 gpu.cu -o gpu

./gpu



### 1. Memoria Global

* Asignada mediante cudaMalloc() en el dispositivo.
* Liberada con cudaFree().
* Requiere transferencia explícita entre host y dispositivo mediante cudaMemcpy().

### 2. Memoria Compartida

* Declarada dentro del kernel con \_\_shared\_\_.
* Asignada estáticamente o dinámicamente mediante parámetros del kernel.
* Es accesible solo por los hilos del bloque correspondiente.

### 3. Memoria Constante

* Asignada mediante cudaMemcpyToSymbol() para copiar datos desde el host a un espacio constante en el dispositivo.
* Declarada con \_\_constant\_\_.

### 4. Memoria Unificada

* Introducida en CUDA 6, asignada con cudaMallocManaged().
* Automáticamente es gestionada entre host y dispositivo.
* Simplifica el desarrollo al eliminar transferencias explícitas.

### 5. Memoria de Páginas Bloqueadas

* Asignada con cudaHostAlloc().
* Permite transferencias más rápidas entre host y dispositivo.

## Conclusión

La asignación contigua es eficiente para asignar bloques grandes de memoria, pero puede causar fragmentación.

La jerarquía de memoria en GPUs está cuidadosamente diseñada para maximizar el rendimiento en aplicaciones paralelas. Aunque introduce complejidad en la programación, el entendimiento y la optimización de su uso pueden proporcionar enormes beneficios en términos de eficiencia y escalabilidad. Comparada con las memorias tradicionales de CPU y PC, la memoria en GPUs está optimizada para aprovechar el alto grado de paralelismo inherente a estas arquitecturas.