PROYECTO DE ACELERACIÓN DE ALGORITMOS CON GPGPU

Grupo: Los Quantums

Sergio Quevedo
Ana Bernabeu
Sergio Cortés
Erik Ikaev
Esteban Pi

ÍNDICE

INTRODUCCIÓN

En este proyecto de aceleración de algoritmos con GPGPU, se busca evaluar cómo CUDA puede mejorar el rendimiento de algoritmos computacionalmente intensivos. Se analizará el proceso de selección, implementación y evaluación de rendimiento de cada algoritmo (en nuestro caso, es el de simulación del movimiento de una superficie sobre un torno de mecanizado) destacando el papel de la función principal proporcionada. El objetivo es comprender el impacto de la aceleración GPU en diversas aplicaciones y fomentar un conocimiento profundo de su potencial transformador en el desarrollo de software de alto rendimiento.

¿Qué es una GPU?

Una Unidad de Procesamiento Gráfico (GPU) es un tipo de procesador especializado diseñado para procesar y renderizar gráficos en computadoras y dispositivos móviles. A diferencia de la CPU, que está diseñada para manejar una variedad de tareas generales, la GPU se centra en operaciones matemáticas intensivas, como cálculos vectoriales y de matriz, que son fundamentales para renderizar imágenes y vídeos de alta calidad. Además, con el surgimiento de las GPGPUs, las GPUs también se utilizan para acelerar aplicaciones no gráficas mediante la paralelización de tareas, lo que las convierte en una herramienta poderosa para la computación de alto rendimiento.

¿Son lo mismo las GPGPU?

Las GPGPU (Unidad de Procesamiento Gráfico de Propósito General) se refiere a la utilización de GPUs para propósitos generales más allá de los gráficos. Esto implica aprovechar la capacidad de cálculo masivamente paralela de las GPU para acelerar tareas computacionales intensivas en campos como la simulación científica, el aprendizaje automático, la criptografía y más.

Por tanto, mientras que una GPU se centra principalmente en el procesamiento gráfico, una GPGPU se utiliza para una amplia gama de aplicaciones computacionales.

¿Qué es CUDA?

CUDA (Compute Unified Device Architecture) es una arquitectura de cálculo paralelo desarrollada por NVIDIA, diseñada para aprovechar la potencia de las GPUs (Unidades de Procesamiento Gráfico) con el fin de mejorar significativamente el rendimiento del sistema.

Esta plataforma proporciona extensiones de C y C++ que permiten implementar paralelismo en el procesamiento de tareas y datos, así como APIs que facilitan el desarrollo tanto de código host (en la CPU) como de código kernel (que se ejecuta en la GPU).

Un programa en CUDA consta de dos partes: el código host, que interactúa con la GPU, y el código kernel, que se ejecuta en la GPU. Existen APIs Runtime y Driver para el desarrollo en GPU, diferenciadas por su nivel de abstracción y complejidad.

En cuanto a su estructura interna, un kernel CUDA se ejecuta sobre un grid de hilos, donde cada hilo ejecuta el mismo código con sus propios índices para direccionar la memoria y realizar la lógica. La organización de los bloques e hilos en el kernel permite una programación flexible y eficiente, facilitando el desarrollo de aplicaciones con alto nivel de paralelismo.

IMPLEMENTACIÓN DE LA FUNCIÓN SimulacionTornoGPU

A continuación se muestran capturas del código realizado en el archivo simutorno.cu, con una breve explicación de cada una de ellas, con el que hemos realizado el algoritmo Simulación por Torno en la GPU.

Función kernel(TPoint3D**, double*, int, int, double)

```
TPoint3D** bufferGPU;
double* bufferSolGPU;
// ------------------------------------------------------------
// ------------------------------------------------------------
// FUNCION A IMPLEMENTAR POR EL GRUPO (paralelización de la anterior)
// ------------------------------------------------------------
// ------------------------------------------------------------
__global__ void kernel(TPoint3D** bufferGPU, double* bufferSolGPU,int pasossim, int vtotal,double ph) {
    int v = blockIdx.x;
    int u = blockIdx.y;

    double incA = 360.0 / (double)ph;
    double AvanceMin = 1e10;
    double angle = 0.0;
    for (int i = 0; i < pasossim; i++)  /* Giro del torno en el eje X */
    {
        // Se rota el punto actual (solo interesa la coordenada y)
        double py = bufferGPU[v][u].y * cos(angle * M_PI_180) - bufferGPU[v][u].z * sin(angle * M_PI_180);
        // Calcula la distancia al origen del punto transformado
        // Si y es la menor se almacena
        if (py < AvanceMin)
        {
            AvanceMin = py;
        }
        angle += incA;
    }
    int p = vtotal * u + v;
    bufferSolGPU[p] = AvanceMin;

}
```

La función kernel paraleliza el cálculo de la simulación de un torno en una GPU usando CUDA. Cada hilo de ejecución procesa un punto de la malla tridimensional bufferGPU, rotando dicho punto en el eje X según el ángulo calculado en cada iteración. La rotación se realiza para un número determinado de pasos pasossim, actualizando las coordenadas de los puntos en bufferSolGPU con las coordenadas y mínimas obtenidas tras cada rotación. El índice “v” corresponde a la coordenada vertical y “u” a la horizontal de la malla, asignando un bloque de CUDA a cada punto de la malla.

Función SimulacionTornoGPU(int, int, int)

Primera Parte de la Función

```
int SimulacionTornoGPU(int pasossim, int vtotal, int utotal) {
    cudaError_t cudaStatus;

    // Verificar si la matriz buffer está inicializada
    if (S.Buffer == NULL) return ERROR;

    // Reserva de memoria en la CPU para la matriz buffer
    TPoint3D* bufferCPU = (TPoint3D*)malloc(S.UPoints * S.VPoints * sizeof(TPoint3D));
    if (bufferCPU == NULL) {
        fprintf(stderr, "Error: malloc failed\n");
        return ERROR;
    }

    // Copia de la matriz buffer de la CPU a la GPU
    TPoint3D* bufferGPU;
    cudaStatus = cudaMalloc((void**)&bufferGPU, S.UPoints * S.VPoints * sizeof(TPoint3D));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "Error: cudaMalloc failed\n");
        free(bufferCPU);
        return ERROR;
    }
    cudaStatus = cudaMemcpy(bufferGPU, S.Buffer, S.UPoints * S.VPoints * sizeof(TPoint3D), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "Error: cudaMemcpy failed\n");
        cudaFree(bufferGPU);
        free(bufferCPU);
        return ERROR;
    }
```

La función SimulacionTornoGPU realiza la simulación de un torno utilizando la GPU para acelerar el cálculo. Primero, verifica si la matriz S.Buffer está inicializada. Luego, reserva memoria en la CPU para la matriz bufferCPU y en la GPU para bufferGPU. A continuación, copia la matriz S.Buffer desde la memoria de la CPU a la memoria de la GPU. En caso de error en la reserva o copia de memoria, se libera la memoria reservada y se devuelve un código de error. Este proceso prepara los datos necesarios en la GPU para la simulación posterior.

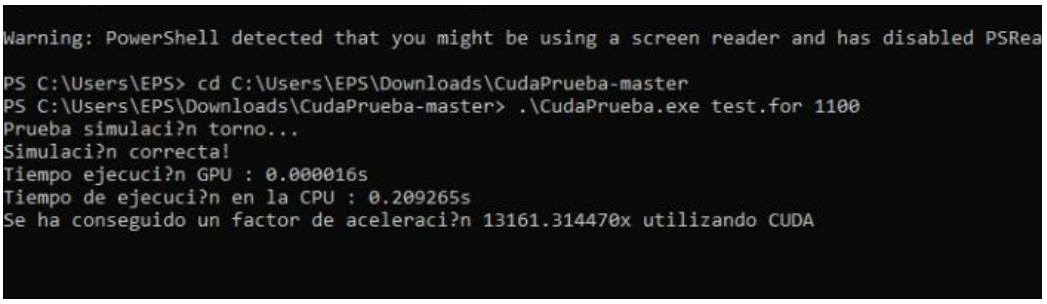
Segunda Parte de la Función

```
    // Reserva de memoria en la GPU para el vector de solución
    double* bufferSolGPU;
    cudaStatus = cudaMalloc((void**)&bufferSolGPU, S.UPoints * S.VPoints * sizeof(double));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "Error: cudaMalloc failed\n");
        cudaFree(bufferGPU);
        free(bufferCPU);
        return ERROR;
    }

    // Llamada al kernel de CUDA
    dim3 gridsize(S.VPoints, S.UPoints, 1);
    kernel<<<gridsize, 1>>>(bufferGPU, bufferSolGPU, pasossim, S.VPoints, PuntosVueltaHelicoide);

    // Copiar solución GPU a la memoria principal de la CPU
    double* GPUBufferMenorY = (double*)malloc(S.UPoints * S.VPoints * sizeof(double));
    if (GPUBufferMenorY == NULL) {
        fprintf(stderr, "Error: malloc failed\n");
        cudaFree(bufferGPU);
        cudaFree(bufferSolGPU);
        free(bufferCPU);
        return ERROR;
    }
    cudaStatus = cudaMemcpy(GPUBufferMenorY, bufferSolGPU, S.UPoints * S.VPoints * sizeof(double), cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "Error: cudaMemcpy failed\n");
        cudaFree(bufferGPU);
        cudaFree(bufferSolGPU);
        free(bufferCPU);
        free(GPUBufferMenorY);
        return ERROR;
    }

    // Liberación de memoria
    cudaFree(bufferGPU);
    cudaFree(bufferSolGPU);
    free(bufferCPU);

    return OKSIM;
}
```

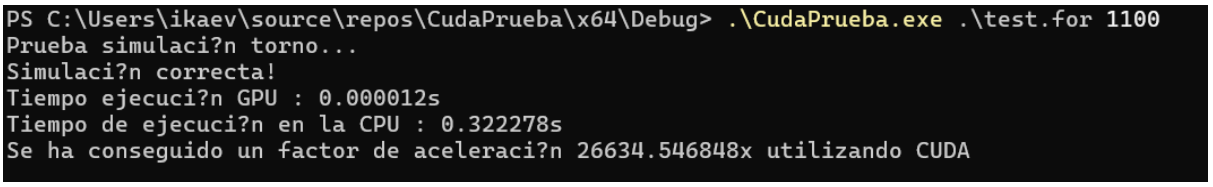
Más tarde se reserva memoria en la GPU para almacenar el vector de solución bufferSolGPU. En caso de fallo en esta reserva, se libera cualquier memoria previamente asignada y se devuelve un código de error. Luego, se configura el tamaño de la cuadrícula para lanzar el kernel CUDA, y se llama al kernel para realizar la simulación en paralelo en la GPU. Posteriormente, se reserva memoria en la CPU para almacenar los resultados obtenidos de la GPU (GPUBufferMenorY), y se copia el contenido de bufferSolGPU desde la memoria de la GPU a la memoria de la CPU. Si la copia falla, se libera toda la memoria previamente asignada y se devuelve un código de error. Después de esto, se libera la memoria asignada en la GPU (bufferGPU y bufferSolGPU), así como en la CPU (bufferCPU). Finalmente, si todos los pasos se completan sin errores, la función retorna OKSIM, indicando que la simulación se ha realizado correctamente.

RESULTADOS Y EVALUACIÓN

A continuación, para ejecutar el programa se requerirá el ejecutable .exe y el test .for, que contendrá todos los datos de la geometría (puesto en el enunciado). Junto con ello, se necesitará el número de las iteraciones:



(Ordenar de EPS, intel i5 12ªGen + 1650 gtx)



(Portátil, intel i7 8ªGen + 1050 gtx)

La diferencia entre dos máquinas, es que la primera es más rápida que la segunda por 4 segundos de GPU, ya que el primero es más actualizado y en caso de CPU parecido. Además, cabe recalcar que el grupo usó para ejecución del CUDA con NVIDIA Toolkit v12 y no la versión 8, como los ordenadores de la EPS, que la solución es crear un nuevo proyecto y extraer los archivos .cu y .h. También, desafortunadamente, la mayoría de los miembros de este grupo no tienen la gráfica de NVIDIA para poder ejecutar y comparar, y por ese motivo no hay tantas pruebas.

REFERENCIAS

- Materiales y PDFs proporcionados por la asignatura.
- https://developer.nvidia.com/