Práctica 3

Algoritmo Factorización de Cholesky en CUDA

Irene Huertas González

09-06-21

Explicación del algoritmo

Tenemos que A es una matriz cuadrada, simétrica y definida positiva. La factorización de Cholesky afirma que toda matriz con las características de A se puede descomponer en el producto de dos matrices: una triangular inferior L por una triangular superior U;

$$A = L * U$$

Además, U = L traspuesta.

¿Cómo se calculan los elementos de la matriz L a partir de A? Siguiendo este esquema:

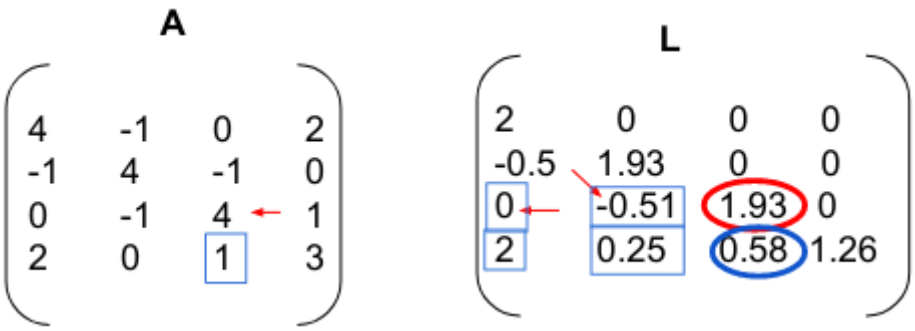
Si estamos calculando un elemento de la diagonal principal de L

$$l_{k,k} = \sqrt{a_{k,k} - \sum_{j=1}^{k-1} l_{k,j}^2} \quad (*1*)$$

Si estamos calculando un elemento que no es de la diagonal

$$l_{k,i} = \frac{a_{k,i} - (\sum_{j=1}^{i-1} l_{i,j} \cdot l_{k,j})}{l_{i,i}} \quad (*1*)$$

Ejemplo:



Para ver de una forma más visual el cálculo de un elemento de la diagonal y de otro no diagonal he puesto estos cálculos

$$1.93 = \sqrt{4 - (0^2 \cdot (-0.51)^2)}$$ usa los elementos marcados/señalados en rojo

$$0.58 = \frac{1 - (0 \cdot 2 + (-0.51) \cdot 0.25)}{1.93}$$ usa los elementos marcados en azul

Paralelización CUDA

En primer lugar reservamos espacio en la GPU y rellenamos las matrices con los valores de las matrices creadas en CPU

```
    // Reservamos espacio en la GPU para ambas matrices
    float* d_A, * d_L;
    cudaMalloc(&d_A, bytes);
    cudaMalloc(&d_L, bytes);


    // Rellenamos las matrices creadas en GPU con valores
    cudaMemcpy(d_A, a.data(), bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_L, l.data(), bytes, cudaMemcpyHostToDevice);

```

El trabajo de descomponer la matriz lo realizan 2 Kernel, uno hace el cálculo de elementos de la diagonal y el otro calcula elementos de la columna que están debajo del elemento de la diagonal.

```
//Kernel encargado de los elementos de la diagonal
__global__ void Diagonal(float* matriz_a, float* matriz_l, int n, int numCol) {

    int fila = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Solo sacamos elementos de la diagonal principal
    if (fila < n && col < n) {
        if (fila == col && col == numCol) {
            float diagSum = 0.0f;
            for (int k = 0; k < col; k++) {
                diagSum += (matriz_l[col * n + k]) * (matriz_l[col * n + k]);
            }
            matriz_l[fila * n + col] = sqrt((float)(matriz_a[fila * n + col] - diagSum));
        }
    }
}
```

```
//kernel encargado de los elementos de la columna
__global__ void Columna(float* matriz_a, float* matriz_l, int n, int num) {

    int fila = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Elementos de la columna por debajo de la diagonal principal
    if (col == num && fila > col && fila < n) {
        float sum = 0.0f;
        for (int k = 0; k < col; k++) {
            sum += (matriz_l[fila * n + k] * matriz_l[col * n + k]);
        }
        matriz_l[fila * n + col] = (matriz_a[fila * n + col] - sum) / matriz_l[col * n + col];
    }
}
```

Finalmente se copia los valores en CPU y ya se libera memoria de la GPU

Experimento 1

Fijamos el número de bloques a 1

Tiempo según num_hebras	Problema pequeño n=500	Problema grande n=6000	Problema supergrande n=30000
Threads = 32	0.00110631s	0.0186637s	0.0694047s
Threads = 64	8.9893e-05s	0.0010447s	0.00531407s
Threads = 256	9.0048e-05s	0.00105363s	0.00529861s

Tarda más cuánto mayor es el tamaño del problema y menor el número de hebras empleadas por bloque. Al variar el número de hebras por bloque, afectamos de forma directa a la distribución del trabajo: Al aumentar el número de hebras, reducimos la granularidad. Cuando una hebra tiene que gestionar muchos datos es cuando empiezan las colisiones y con ello el aumento del tiempo de ejecución.

Experimento 2

Fijamos el número de hebras a 64

Tiempo según num_bloques	Problema pequeño n=500	Problema grande n=6000	Problema supergrande n=30000
Bloques = 1	8.9893e-05s	0.0010447s	0.00531407s
Bloques = 10	9.0514e-05s	0.00105511s	0.00538732s
Bloques = 100	9.0142e-05s	0.00105069s	0.00532244s
Bloques = 240	9.1365e-05s	0.00106557s	0.0052667s

Aquí no varía el tiempo de ejecución, solo cuando el tamaño del problema es mayor se incrementa lo cual es normal. Vemos que variar el número de bloques sin variar el de las hebras no tiene efecto de mejora en la eficiencia. El cantidad de trabajo de los kernel es casi la misma siempre (no varía la estructura) por lo que el tiempo no se ve significativamente afectado. La mayoría de las hebras que trabajan en este kernel no hacen nada.

Comparativa tiempos distintas implementaciones

Tiempo	Problema pequeño n=500	Problema grande n=6000
Secuencial (Local)	0,0244	43,5414
MPI	-	-
CUDA	9.0142e-05s	0.00105069s

Se ve como CUDA es la mejor en eficiencia ya que el secuencial es muy lento y en MPI por la implementación que hice ni siquiera era capaz de ejecutar matrices tan grandes ya que necesitaba tantos procesos como dimensión tuviese la matriz.

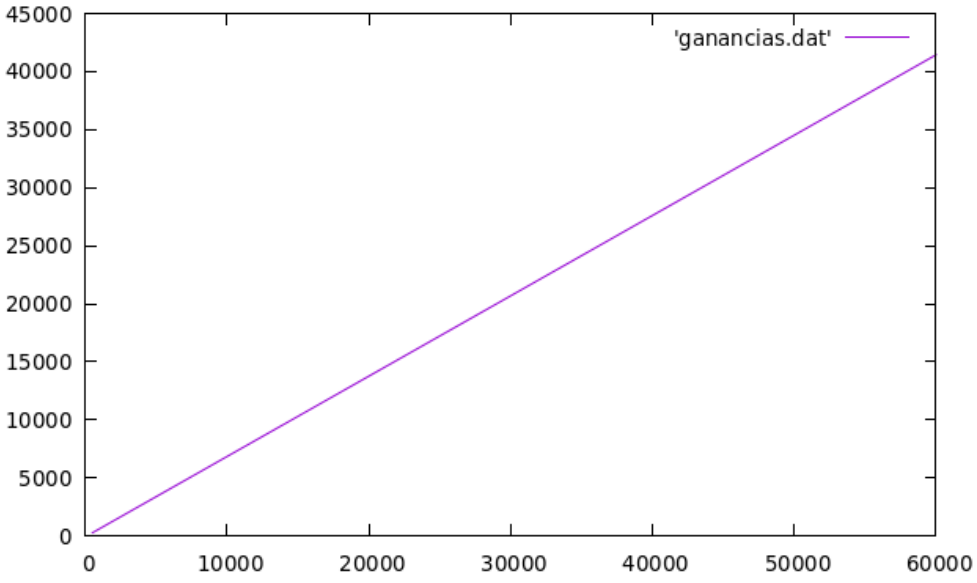
Lo máximo que llegué a calcularle fue matriz 27x27 con 3 procesos que tardaba 0,27114303s

He intentado mejorar la implementación de MPI pero no he sido capaz de sacar ninguna otra que me calculara bien la factorización.

La única ganancia que puedo calcular es la de CUDA con respecto a la implementación secuencial

	Problema pequeño n=500	Problema grande n=6000
Ganancia	270.68	41440.77

Las ganancias son exageradas. No se si mi programa mide bien los tiempos porque me sorprende mucho tan poco tiempo empleado.



No puedo calcular ganancias de las 3 implementaciones ya que si me rebajo a n=4 la matriz es tan pequeña que no son relevantes los tiempos obtenidos. El problema de todo esto es MPI.

n=4	Secuencial	MPI	CUDA
Tiempos	2,0532e-06s	0.000147	2.284e-06s

Comentario y valoración del trabajo

La peor implementación sin lugar a dudas es la de MPI. Nos limita a tamaños bajísimos a la hora de paralelizar ya que para matrices de dimensiones superiores a 4 ya requerimos de un hostfile y se vuelve un ejercicio de concurrencia con tiempos catastróficos. Por eso la considero peor incluso que secuencial a pesar de su lentitud.

La conclusión a la que he llegado (de hecho ya lo comenté en la práctica 2) es que he elegido un algoritmo no paralelizable por culpa de las dependencias entre los cálculos. Creo que debería haber escogido otro mejor para poder haber calculado ganancias y haber podido llegar a conclusiones más elaboradas en vez de remarcar errores como he estado haciendo en esta práctica.

Aún así podemos ver lo potente que es una herramienta como CUDA ya que nos soluciona problemas que en local nos lleva cerca de 1 minuto en pocos segundos.

Creo que debería haberme organizado mejor en general a la hora de realizar las prácticas en general pero he aprendido muchísimo realizando, más que en el resto de asignaturas que tengo este cuatrimestre.

Para concluir, opino que la paralelización es algo muy complejo pero si lo entiendes y lo empleas es una herramienta cuanto menos útil.