Análisis numérico y computo científico

Factorización de Cholesky en GPU con arquitectura CUDA

Instructor: Erick Palacios
Walter Martínez Santana
José Carlos Castro

May 30, 2017

Introducción

El propósito del presente proyecto es implementar un algoritmo distribuido de la descomposición de Cholesky en una unidad de procesador gráfico. Mediante el uso de la arquitectura CUDA vamos a desarrollar el modelo de programación en paralelo. La idea es paralelizar el algoritmo de optimización secuencial y distribuir sobre las GPUs.

Fundamentos matemáticos

Descomposición de Cholesky

Si A es una matriz simétrica positiva definida de $n \times n$, entonces existen matrices triangulares tales que:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} = \begin{bmatrix} u_{11} & 0 & \dots & 0 \\ u_{21} & u_{22} & \dots & 0 \\ \vdots & & \ddots & \vdots \\ u_{n1} & u_{n2} & \dots & u_{nn} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & u_{nn} \end{bmatrix}$$

Por lo que cada entrada de A, a_{ij} puede estar representada por entradas de U operadas entre ellas, i.e

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} = \begin{bmatrix} u_{11}^2 & u_{11}u_{12} & \dots & u_{11}u_{1n} \\ u_{11}u_{12} & u_{12}^2 + u_{22}^2 & \dots & u_{12}u_{1n} + u_{22}u_{2n} \\ \vdots & & \ddots & \vdots \\ u_{11}u_{n1} & u_{12}u_{1n} + u_{22}u_{2n} & \dots & \Sigma_{i=1}^{n} u_{ii}^2 \end{bmatrix}$$

Por lo que para obtener las u's es necesario igualar y así obtenemos una solución recursiva por la derecha. Y obtenemos los coeficientes de u_{ij}, en términos de a_{ij} $u_{i-k,j-l}$ para $k, l = 1, \dots, n-1$. Por ejemplo si $n = 4$.

$$
\begin{aligned}
&\mathrm{u}_{11} = \sqrt{a_{11}},\ u_{12} = \frac{a_{12}}{u_{11}},\ u_{12} = \frac{a_{13}}{u_{11}}, u_{13} = \frac{a_{13}}{u_{11}}, u_{14} = \frac{a_{14}}{u_{11}},\\
&\mathrm{u}_{21} = 0,\ u_{22} = \sqrt{a_{22} - u_{12}^2},\ u_{23} = \frac{a_{23} - u_{12}u_{13}}{u_{22}}, u_{24} = \frac{a_{24} - u_{12}u_{14}}{u_{22}},\\
&\mathrm{u}_{31} = 0,\ u_{32} = 0\, u_{33} = \sqrt{a_{33} - u_{13}^2 - u_{23}^2},\ u_{34} = \frac{a_{34} - u_{13}u_{14} - u_{23}u_{24}}{u_{33}},\\
&\mathrm{u}_{41} = 0,\ u_{42} = 0\, u_{43} = 0\, u_{44} = \sqrt{a_{44} - u_{14}^2 - u_{24}^2 - u_{34}^2}
\end{aligned}
$$

Y de lo anterior podemos por hipótesis de inducción sobre n:

$$
\begin{aligned}
&\mathrm{u}_{ij} = \sqrt{a_{jj} - \Sigma_{k=1}^{j-1} u_{j,k}^2} \qquad si\ j = i,\\
&\mathrm{u}_{ij} = \frac{1}{u_{jj}}(a_{ij} - \Sigma_{k=1}^{j-1} u_{i,k}u_{j,k}), \qquad si\ j < i
\end{aligned}
$$

Antes de escribir un pseudocódigo para la forma general de las matrices triangulares, notemos que en general es fácil recorrer en u_{ij} sobre las j's, ya que lo que se necesita se conoce en el paso inmediato anterior. Y como partimos de a_{11} para conocer nuestra primera incógnita $u_1 1$. Entonces únicamente mediante elementos de A se pueden conocer los elementos de U, sin embargo si existen relaciones de dependencia con las entradas anteriores. Como podemos ver en nuestro pseudocódigo:

Pseudocódigo

Algorithm 1 Descomposición de cholesky

```
1:  procedure CHOL(n × n)                                  ▷ A positiva definida
2:      n = dim(A)
3:      int i, j, k
4:      for k in 1 : n − 1
5:      a_kk = √a_kk
6:          for j in k + 1 : n − 1
7:          a_k,j = a_k,j / a_k,k
8:          end
9:          for j in k + 1 : n − 1
10:             for j in i : n − 1
11:             a_i,j = a_i,j − a_k,i a_k,j
12:             end
13: end procedure
```

Fundamentos numéricos

El método de la factorización de Cholesky para ser realizado computacionalmente para una matriz A de nxn necesita unicamente el triangulo superior o inferior de A, ya que es simétrica por lo que comenzamos con $\frac{n \times n}{2} + n$ valores, a partir de los cuales tenemos dos tipos de operaciones. Entonces el costo total es el numero de multiplicaciones mas el número de sumas(ó restas).

$$(m+1)(m-k) - \Sigma_{j=k+1}^{m} j = \frac{(m-k)(m-k+1)}{2}\ multiplicaciones$$

$$\text{(m+1)(m-k) - } \Sigma_{j=k+1}^{m} j = \frac{(m-k)(m-k+1)}{2}\ restas$$

Además el ciclo for de afuera corre de 1 a m por lo que tendrá $m-1$ divisiones, $\frac{1}{3}m(m^2-1)$ multiplicaciones y $\frac{1}{3}m(m^2-1)$ restas. Por lo tanto el costo del algoritmo es de aproximadamente $\frac{2}{3}m^3 + f13m$ operaciones por lo que a gran escala podemos decir que es del orden de $\frac{2}{3}m^3$.

Por otra parte si el algoritmo descrito se utiliza para resolver un sistema de ecuaciones esta sujeto a tener un error de aproximación a x ya que únicamente encontramos $\hat{x}$ por lo que tenemos que analizar el error numérico inducido por el número de condición de la matriz i.e el error en los datos y también el error numérico de nuestra aproximación, i.e el error en el algorítmo.

Sea $A \in \mathbf{R}^{n \times n}$ matriz positiva definida, i.e $A = A*, x * Ax \leq 0$, entonces tomando el costo del algoritmo

$$\begin{aligned} A &= U^T U \\ \|U\| &= \sqrt{\|A\|} \\ A + \delta A &= \hat{U}^T \hat{U} \\ &= \frac{\|\delta A\|}{\|U\|\|U^T\|} \\ &= \frac{\|\delta A\|}{\|A\|} \\ &= \mathcal{O}(\epsilon) \end{aligned}$$

Por lo que el algoritmo será siempre numéricamente estable hacia atrás.

Paralelización en GPU utilizando arquitectura CUDA

Pseudocódigo

Algorithm 2 Descomposición de cholesky

```
1: procedure CHOL(n × n)                                   ▷ A positiva definida
2:     N = dim(A)
3:     while (k < N)
4:         if (rd1 == inicio)
5:         else
6:             while (rd1 < N2)
7:                 while (rd1 < N2)
8:                 M(rd1) = M(id1)/M(inicio)
9:                 id1+ = B_mx C_rix D_mx
10:            __synctrhead();
11:        rd = ids;
12:        inicio = (N − K);
13:        while (NN < N(N + 1)/2);
14:            id2 = id + NN;
15:            while(id2 < NN + (N − KK);
16:                m(id2) = m(id2) - m (id + kk) - m(kk);
17:                id2 += Grrd BLock;
18:            __synctrhead();
19:        NN += N-KK;
20:        KK++;
21:    K++
22:    N2 += N-K;
23: end procedure
```

El algoritmo que se muestra se centra en la idea de distribuir la factorización de cholesky, por lo que el propósito es escribir un archivo .cu que contenga tanto como el host como los kernels. Utilizando nuestro algoritmo base se sube la matriz al host y luego se copia a la memoria ram de nuestra tarjeta NVIDIA. Y aquí entra nuestra primera iteración que recorrerá las columnas. Y en cada iteración se inicializa algún Kernel y cada uno ejecuta una vez por cada elemento en la parte superior ó inferior de la matriz. Esta es la idea principal de como inicializar nuestro algorítmo, sin embargo hay más de una manera de decidir como se van a inicializar los Kernel.

1. En la versión Normal Single Kernel, el código del kernel para cada elemento es el mismo para el elemento diagonal principal y todos los elementos debajo de él, con una instrucción condicional que comprueba cuál de los dos es la ejecución actual. Una sentencia condicional como ésta puede ser muy perjudicial para el rendimiento, especialmente en plataformas como GPUs, que aprovechan las operaciones en serie en grandes conjuntos de datos.

2.- La versión In-Place Multiple Kernel utiliza varios kernels, es decir, Dos núcleos para cada columna, utiliza uno para la diagonal principal, y uno para el resto. Sin embargo, esta versión utiliza la misma matriz para su entrada y salida. Este pequeño cambio puede producir un gran aumento de rendimiento, ya que sólo almacena la mitad de la memoria. Y por lo tanto más datos útiles se pueden guardar en caché. El unico inconveniente es que requiere el uso de dos Kernels, ya que es la única manera que la sincronización global de los threads CUDA.

Conclusiones

El algoritmo presentado de factorización de Cholesky paralelizada hace uso óptimo de los caches de la computadora, para incrementar la eficiencia. Gracias a la arquitectura distribuída en CUDA podemos hacer uso del GPU para así tener un incremento considerable en la velocidad de ejecución en el caso de matrices de alta dimensionalidad. Por lo que pensamos que es posible concluír que este algorítmo conduce a una técnica efectiva para resolver problemas de algebra lineal, especialmente cuando trabajamos con matrices densas ya que para matrices ralas existen otro tipo de algorítmos que aprovechan la gran cantidad de ceros que tiene la matriz. Sin embargo presentamos algunos problemas para distribuir la malla de manera óptima, además de que contamos con varios ciclos iterativos que quizá se pueden reducir para mejorar el tiempo aún más. Además es un algorítmo que valdría la pena probar en diferentes equipos en una segunda iteración, así como diferentes versiones.