Tecnológico de Monterrey

Proyecto Final

Cómputo Paralelo y Distribuido

Análisis Comparativo de Técnicas de Paralelización de la Multiplicación de Matrices Usando CUDA

Camilo Andrés Rivera Lozano

4 de julio de 2015

Índice

1. Introducción

La computación de altas prestaciones es una herramienta que se está volviendo casi indispensable para el desarrollo tecnológico y científico de hoy en día. Esto se debe a que ha permitido alcanzar metas antes no pensadas, resolver problemas cuya resolución se creía imposible o inalcanzable e incluso ha traído nuevas soluciones sobre la mesa de trabajo. Esto se debe a que ahora en vez de tener un sólo computador (o cerebro) con grandes especificaciones, alta potencia de cálculo y una gran cantidad de recursos, ahora contamos con cientos de procesadores unidos y trabajando en equipo por una misma causa. Cada uno de estos procesadores puede incluso trabajar con su propio conjunto de recursos de memoria y comunicación.

Entre los modelos de trabajo paralelo de alta prestación, aparece uno recientemente que es el trabajo con Unidades de Procesamiento Gráfico (GPU por sus siglas en inglés), las cuales fueron inicialmente diseñadas para funcionar como co-procesadores para el trabajo sobre gráficos y alta definición de imágenes y videos. Pero dada su arquitectura y su alto poder de cómputo se prestaron para el procesamiento paralelo de alto poder de procesamiento.

En el presente documento se presenta un ejemplo de la forma en que una GPU, particularmente un chip de NVIDIA bajo la programación de CUDA, puede ser utilizado para el procesamiento de gran cantidad de datos en un mucho menor tiempo. El ejemplo discutido es la multiplicación de matrices de hasta 1000x1000 elementos, y su comparativa con el tiempo que tardaría un procesador escalar, o secuencial, en llevar a cabo dicha tarea.

2. Fundamento Teórico

2.1. Arquitectura GPU

Antes que nada es importante comprender cómo se organizan las GPU's en términos de hardware y recursos. Contrario a las CPU's, o Unidades de Procesamiento Central, tradicionales, en donde se tienen pocas ALU's (Unidades Aritmético-lógicas) con una gran cantidad de memoria y una caché particularmente rápida, las GPU como se puede ver en la Figura 1, poseen una gigantesca cantidad de procesadores escalares especialmente diseñados para trabajar en conjunto con operaciones matemáticas y de álgebra lineal.

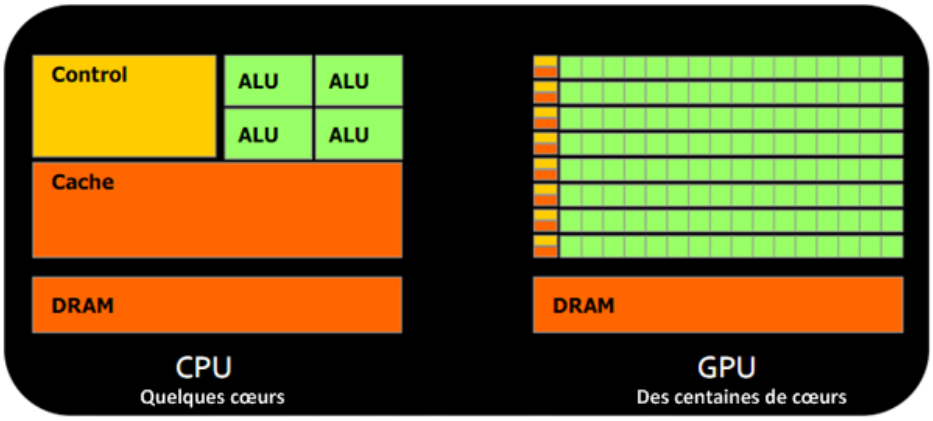

Figura 1: Diferencia de arquitecturas entre CPU y GPU[1]

Al programar en el lenguaje nativo para las GPU's de NVIDIA, CUDA, se utilizan los cores de la GPU llamando threads dentro de la ejecución del programa. Cada thread es ejecutado en uno de los cores en un arreglo llamado bloque, el cual se ejecuta en un multiprocesador. Este concepto es importante debido a que por más que por más hilos o bloques que se llamen, sólo se ejecutarán a la vez los disponibles por hardware en la GPU, la cuál viene con una cantidad de multiprocesadores fija, cada uno con sus propios recursos, como se muestra en la Figura 2. Cada procesador escalar cuenta con su propio set de registros y todos tiene acceso a las memorias globales, bien sea la de la GPU, la constante o la de texturas. Pero de todas maneras cada uno de los multiprocesadores (o arreglo en bloque de procesadores escalares) cuenta con su propia memoria compartida. De manera que la manera en que se resuelve un problema no depende únicamente del problema en sí, sino que también de los recursos que se tengan disponibles para su programación.

Es importante tener en cuenta que el GPU no toma el control de las operaciones en general, sino que sigue bajo el mando del CPU. De manera que este último es el encargado de lanzar las funciones conocidas como kernels, las cuales se ejecutan en la GPU, especificando el arreglo de procesadores (hilos) que se van a utilizar y la geometría que van a ser, ya sea unidimensional o multidimensional, y de la misma manera la cantidad de bloques que se van a ejecutar en la GPU o grid.

3. Justificación

El ordenamiento de datos en forma de matrices es ampliamente usado en casi todos los ámbitos de la ciencia y tecnología, debido a que éste permite una mejor organización de los datos tanto como para

[1]Imagen tomada de `http://imagefriend.com/cpu-vs-gpu.shtm`

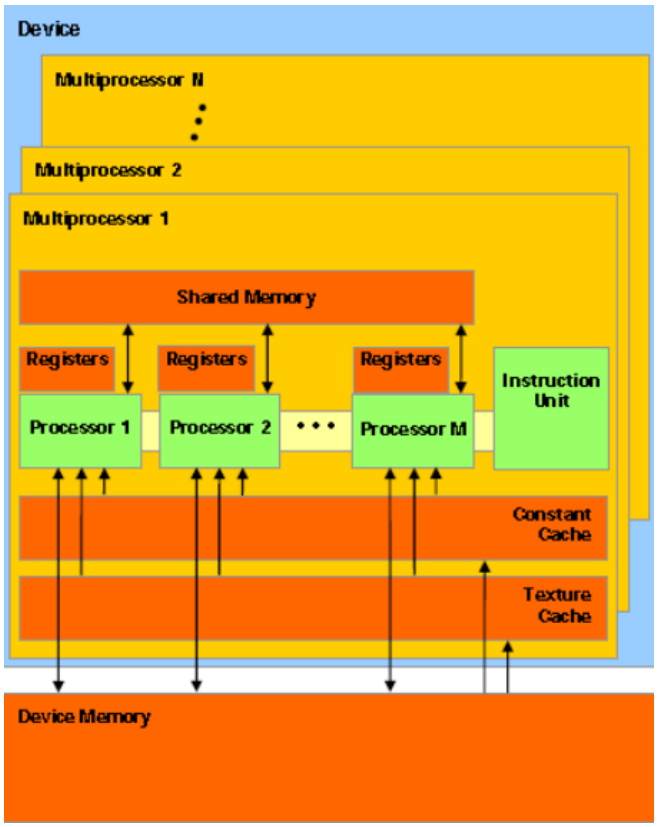

Figura 2: Diferencia de arquitecturas entre CPU y GPU [Datla and Gidijala, 2009]

fines prácticos como técnicos. Cualquier problema que se plantee en una situación multimensional o con una gran cantidad de datos o bien sea que utilicen álgebra lineal; bien sea problemas de optimización, física, química, entre otros, permitirán el uso de matrices. Si se requiere operar sobre dichas matrices o despejar algún grupo de términos lo más seguros es que inevitablemente se desemboque en una multiplicación de matrices. Dependiendo de la cantidad de datos o grados de libertad, estos problemas pueden crecer en gran medida y se vuelve necesario optimizar dicha multiplicación. Es por esto que este proyecto pretende introducir tanto el manejo de programación paralela de GPU mediante CUDA, así como diferentes aproximaciones que se pueden tener al manejar matrices, y mostrar las diferencias en tiempo de cada uno.

4. Objetivos

- Encontrar diferentes mejoras ante diferentes algoritmos paralelos de multiplicación de matrices respecto al método secuencial
- Utilizar la memoria compartida y observar mejoras en los tiempos
- Generar una serie de códigos que se ejecuten en una GPU y sean funcionales para el propósito establecido

5. Desarrollo

El desarrollo fue realizado en una Laptop Samsung que posee un procesador Intel de dos cores de segunda generación, y además una tarjeta gráfica NVIDIA GeForce 540M, que permitía un total de 1024 hilos por bloque, con dos multiprocesadores y 48kb de memoria compartida por cada uno. Fue muy importante tener estos aspectos en cuenta para el desarrollo del proyecto debido a que no sólo permitieron una correcta optimización y ejecución del programa, sino que también facilitaron encontrar errores y fallas.

5.1. Multiplicación de Matrices: Secuencial

Para realizar la multiplicación de matrices se debe explotar al máximo la capacidad de distribución de tareas de manera matricial dentro de la GPU, pero también se debe tener en cuenta que al transferir los datos del *host* al *device* se cuenta espacio limitado. Es importante también tomar en cuenta que la forma en que organicemos nuestros datos definirá lo rápido que se accede a memoria influyendo en el tiempo final de ejecución.

Para el caso secuencial se siguió el código mostrado a continuación. Si bien dada la notación utilizada en el método escrito hace que sea mucho más sencillo crear arreglos bidimensionales, estos entorpecen el proceso y vuelven más lento el acceso a memoria, transformando las matrices a vectores, bien sea por columnas o por filas, es la mejor manera de trabajar con ellas.

```
void mult_mat(int *a, int *b, int *c, int N)
{
  int i,j,k,suma;
  for (i=0; i < N; i++)
  {
    for (j=0; j < N; j++)
    {
      suma=0;
      for (k=0; k < N; k++)
        suma += a[k+i*N] * b[j+k*N];
      c[j+i*N] =suma;
    }
  }
}
```

Aquí se realiza una doble iteración para recorrer la matriz resultado para la cual se realiza una última iteración para recorrer las filas de la primera matriz y las columnas de la segunda.

5.2. Multiplicación de Matrices: Paralelo

Para la multiplicación en paralelo se implementaron tres métodos más una implementación utilizando la memoria compartida. Los dos primeros métodos utilizan la misma metodología del algoritmo secuencial, salvo que sólo se ejecuta la última iteración debido a que organizando los *threads* de manera matricial podemos ahorrarnos las dos primeras iteraciones. Para el primer algoritmo se realiza primero un cálculo bajo el cual se decide la cantidad de bloques e hilos (manteniendo al mínimo la cantidad de bloques ya que los multiprocesadores son recursos más limitados), teniendo en cuenta que el número de hilos por bloque no supere el límite establecido. Después de ello ejecuta el kernel (mostrado a continuación) y recupera los resultados.

```
__global__ void matrixMultGPU(int *a, int *b, int *c, int N){

  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;
  }
}
```

Para el segundo esquema de ejecución, se utiliza la idea de ejecutar todas las tareas en un único bloque, por lo cual se limitan los posibles tamaños de las matrices, pero como se mostrará en la Sección 6, se muestra una mejoría considerable en el rendimiento.

Finalmente, para el tercer caso se optó por dividir las matrices entrantes en 4 partes iguales (creando la restricción de que sean matrices cuadradas de dimensión par) y multiplicando las partes correspondientes. Esto evita realizar recorridos largos, agregando más procesos por cada matriz, pero sobretodo puede evitar la necesidad de asignar grandes espacios de memoria en la GPU, debido a que se puede se puede ejecutar una sub-multiplicación a la vez. La idea general se muestra a continuación,

$$\begin{pmatrix} A_{UL} & A_{UR} \\ A_{LL} & A_{LR} \end{pmatrix} \cdot \begin{pmatrix} B_{UL} & B_{UR} \\ B_{LL} & B_{LR} \end{pmatrix} = \begin{pmatrix} C_{UL} & C_{UR} \\ C_{LL} & C_{LR} \end{pmatrix}$$

$$C = \begin{pmatrix} A_{UL}B_{UL} + A_{UR}B_{LL} & A_{UL}B_{UR} + A_{UR}B_{LR} \\ A_{LL}B_{UL} + A_{LR}B_{LL} & A_{LL}B_{UR} + A_{LR}B_{LR} \end{pmatrix}$$

Hay varias formas de realizar este algoritmo, entre ellas puede que la más efectiva sea lanzar 8 veces el kernel, realizando los preparativos correspondientes cada vez. Para este proyecto se optó por la que ofrecía una menor carga para la CPU, lo cual se hizo por el hecho de que es un ejemplo demostrativo, por lo cual no se van a realizar grandes multiplicaciones y habrá memoria suficiente en el GPU para albergar todas las submatrices a la vez. Como se ve en el código a continuación, una única ejecución del kernel realiza las 8 multiplicaciones. En caso contrario y la implementación demande altas prestaciones se debe extraer una a una las submatrices de la matriz principal e irlas enviando a la GPU para realizar la multiplicación.

```
__global__ void matrixMultGPU(int *a_ll,int *a_lr,int *a_ul, int *a_ur,int
    *b_ll,int *b_lr,int *b_ul, int *b_ur, int *c_ll,int *c_lr,int *c_ul, int
    *c_ur, int *t_ll,int *t_lr,int *t_ul, int *t_ur,int N){

  int k, sum_cur = 0,sum_cul = 0,sum_cll = 0,sum_clr = 0,sum_tur = 0,sum_tul
      = 0,sum_tll = 0,sum_tlr = 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_cul += a_ul[fil * N + k] * b_ul[k * N + col];
      sum_cur += a_ul[fil * N + k] * b_ur[k * N + col];
      sum_cll += a_ll[fil * N + k] * b_ul[k * N + col];
      sum_clr += a_ll[fil * N + k] * b_ur[k * N + col];

      sum_tul += a_ur[fil * N + k] * b_ll[k * N + col];
      sum_tur += a_ur[fil * N + k] * b_lr[k * N + col];
      sum_tll += a_lr[fil * N + k] * b_ll[k * N + col];
      sum_tlr += a_lr[fil * N + k] * b_lr[k * N + col];
    }
    c_ul[fil * N + col] = sum_cul;
    c_ur[fil * N + col] = sum_cur;
    c_ll[fil * N + col] = sum_cll;
    c_lr[fil * N + col] = sum_clr;

    t_ul[fil * N + col] = sum_tul;
    t_ll[fil * N + col] = sum_tll;
    t_lr[fil * N + col] = sum_tlr;
    t_ur[fil * N + col] = sum_tur;
    __syncthreads();

    c_ul[fil * N + col]+=t_ul[fil * N + col];
    c_ll[fil * N + col]+=t_ll[fil * N + col];
    c_lr[fil * N + col]+=t_lr[fil * N + col];
    c_ur[fil * N + col]+=t_ur[fil * N + col];
  }
}
```

Finalmente para la memoria compartida se fijó un tamaño de 31 para todas las ejecuciones (excepto para el tercer caso) para que el caso 2 pudiese operar en un único bloque sin desbordar el número de threads. Como se muestra a continuación, al entrar al kernel se copian las matrices a la memoria compartida y a partir de ellas se realiza el cálculo de la multiplicación.

```
#define N 31
__global__ void matrixMultGPU(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;
  __shared__ int s_a[N*N];
  __shared__ int s_b[N*N];
  if (col < N && fil < N)
  {
    s_a[fil * N + col]=a[fil * N + col];
    s_b[fil * N + col]=b[fil * N + col];
    for (k = 0; k < N; k++)
    {
      sum += s_a[fil * N + k] * s_b[k * N + col];
    }
    c[fil * N + col] = sum;
  }
}
```

6. Resultados

Para obtener los resultados hubo varios problemas en cuestión de transferencia y uso de memoria, puesto que la cantidad de datos crece de manera cuadrática con respecto a la dimensión de la matriz. Es por esto que se tuvieron que agregar condiciones en busca de errores tanto al asignar la memoria en la GPU como en la copia de memoria hacia ésta. Además hubo problemas que se saltaron las comprobaciones sin haber respuesta aparente del programa, arrojando tiempos casi coherentes, los cuales sólo pudieron ser detectados imprimiendo regiones conocidas de la matriz de resultado. Por dichos problemas de memoria se tomaron los tiempos para los tamaños de 100, 200 y 500 (sólo el último método permitió una correcta multiplicación para matrices de 1000 x 1000). En la Tabla 1 se muestran los tiempos obtenidos por multiplicación para cada una de las metodologías.

Tamaño	Secuencial	Par 1	Par 2	Par 3	Compartida
100	4.09 ms	0.201ms	0.218 ms	0.105 ms	–
200	38.28 ms	2.12 ms	0.182 ms	0.78ms	–
500	979.22 ms	32.28 ms	3.52 ms	10.6 ms	–
31	–	14.3 us	14.9 us	11.6 us[2]	6.38 us

Tabla 1: Tabla de tiempos

Es importante tener en cuenta que no todos los problemas (por sus algoritmos internos) utilizaron la misma distribución de bloques e hilos, la cual se puede observar en la Tabla 2, teniendo en cuenta que para la técnica 3 estos bloques e hilos son por cada submatriz ingresada en el kernel.

Tamaño	Par 1	Par 2	Par 3	Compartida
100	4x25	4x25	2x25	–
200	7x29	7x29	4x25	–
500	16x32	16x32	16x16	–
31	1x31	1x31	1x16	1x31

Tabla 2: Tamaños de los kernels (bloquesxhilos)

7. Conclusiones

Como se puede observar en la Tabla 1 hay una mejoría considerable a la hora de realizar una multiplicación de matrices, debido a que se reduce de manera significativa el tiempo de ejecución. Realizando el cociente de los tiempos, notamos que para un tamaño de 100x100 la mejoría oscila entre 18.7 veces (Par 2) hasta 40 veces (Par 3). Por otro lado, para el caso de 500x500 la mejoría es de 30.33 veces para Par 1, 92.37 veces para Par 3, pero para el algoritmo Par 2 el speed-up es de un factor de 278.18 veces. Finalmente observamos que al utilizar la memoria compartida, la velocidad de ejecución tiende a duplicarse respecto al algoritmo base, que en este caso sería el Par 1.

Para trabajos futuros, queda realizar una optimización del uso de memoria, así como intentar con funciones como la cudaMemcpy2D que permite la manipulación de matrices. Para el caso del algoritmo Par 3 se recomienda intentar lanzar el kernel 8 veces con cada multiplicación mostrada anteriormente. De esta manera sólo se reserva la memoria correspondiente a 3 submatrices en la GPU, mientras que en la GPU sólo almacena las dos que se van a multiplicar en cada paso, a pesar de que se tienen que almacenar las 8 de salida.

[2]Para el caso de la técnica de división de matrices se utilizó un tamaño de 32

Referencias

[Datla and Gidijala, 2009] Datla, S. and Gidijala, N. S. (2009). Parallelizing motion JPEG 2000 with CUDA. In *2009 Second International Conference on Computer and Electrical Engineering*. IEEE. 3

[NVIDIA Corporation, 2015] NVIDIA Corporation (2015). Cuda toolkit documentation - v7.0. [online] Disponible: `http://docs.nvidia.com/cuda`.