Universitat Autònoma de Barcelona

Cómputo de Altas Prestaciones

Informe CUDA

Oriol Martínez Acón - 1460561
Marc Mezquita Meseguer - 1455119
Profesora: Anna Bàrbara Sikora

Conocimientos previos

CUDA (Compute Unified Device Architecture)
Es una plataforma de computación en paralelo y un modelo de programación desarrollado por NVIDIA para computación general en unidades de procesamiento gráfico (GPU).

Distribución del SLURM del laboratorio
El nodo “aolin23” dispone de una Nvidia GeForceGTX1080 (2560 cores, 8GB).
El nodo “aolin24” dispone de una Nvidia GeForceGTX1080 Ti (3584 cores, 11GB).
El nodo “aomaster” dispone de una Nvidia GeForce GTX-750 Ti (640cores, 2GB).

Como podemos ver, el nodo “aolin23” y el nodo “aolin24” nos proporcionan un mayor nivel de cores, anchos de banda y tamaño de memoria. Por la cual cosa para poder ejecutar el código de la red neuronal usaremos cualquiera de los dos nodos mencionados.

Introducción

La idea de esta práctica es la aplicación de CUDA en el código de la red neuronal. De esta forma podremos evitar la secuencialidad que presenta el programa original y aplicar un nivel de paralelismo bastante alto.

Compilación y ejecución

Antes de poder compilar o ejecutar el código primero, nos deberemos de loguear en el laboratorio a través del siguiente comando:
ssh -p 54022 cap-2-5@aolin-login.uab.es

Una vez logeados especificaremos en el fichero “job.sub” el acceso al “slurm”, la compilación del código de la red neuronal y la ejecución de este programa.

```
#!/bin/bash -l
#

#SBATCH --job-name=GPU
#SBATCH -N 1 # number of nodes
#SBATCH --partition=cuda.q
#SBATCH --gres=gpu:1

module load cuda/9.0
module unload gcc/8.2.0

g++ -zmuldefs -c common.cpp -o common.o

nvcc -c nn-vo-db.cu -o nn-vo-db.o

g++ -zmuldefs -o nn-vo-db common.o nn-vo-db.o -L/usr/local/cuda-9.0/lib64 -lm -lcudart -lrt

./nn-vo-db
```

Imagen del fichero “job.sub”.

Tal y como podemos ver en la imagen en este fichero se encuentra todas las especificaciones para poder ejecutar la red neuronal en el SLURM:

#!/bin/bash -l: Especifica que el fichero es de tipo bash.
#SBATCH -N1: Solicita el número de nodos a utilizar. En nuestro caso hemos especificado solo 1.
#SBATCH --partition=cuda.q: Solicita la partición específica para la asignación de recursos.
Si no se especifica este parámetro, el comportamiento por defecto es permitir que el controlador del SLURM seleccione la partición predeterminada por el administrador del sistema.
#SBATCH --gres=gpu:1: Especifica una lista delimitada por comas de recursos consumibles genéricos.

Podemos ver que en cuanto a los módulos de los compiladores tenemos dos especificaciones:

module load cuda/9.0: Carga la última versión de cuda, la 9.0, para la ejecución de la red neuronal.
module unload gcc/8.2.0: Descarga la versión 8.2.0 del compilador gcc. Una vez ejecutado este comando la versión del compilador por defecto debería ser la 4.8.5.

Antes de poder usar las especificaciones de la compilación tenemos que cambiar la extensión de los siguientes ficheros a través de los siguientes comandos:

mv ./common.c ./common.cpp: Cambia la extensión del fichero “common.c” a “common.cpp”.
mv ./nn-vo-db.c ./nn-vo-db.cu: Cambia la extensión del fichero “nn-vo-db.c” a “nn-vo-db.cu”.

Una vez cambiadas las extensiones podemos seguir con las especificaciones de compilación:
g++ -zmuldefs -c common.cpp -o common.o: Compilación del fichero common.cpp a través de g++ (compilador de cpp). Este comando genera como ejecutable el fichero “common.o”.
nvcc -c nn-vo-db.cu -o nn-vo-db.o: Compilación del fichero de cuda a través de nvcc (compilador de cuda). Este comando genera como ejecutable el fichero “nn-vo-db.o”.
g++ -zmuldefs -o nn-vo-db common.o nn-vo-db.o -L/usr/local/cuda-9.0/lib64 -lm -lcudart -lrt: Enlazamiento de los dos ejecutables a través del compilador g++. Este comando genera como ejecutable el fichero “nn-vo-db”.

Una vez hecho esto lo único que queda es la especificación del ejecutable:
./nn-vo-db: Ejecución del ejecutable “nn-vo-db”.

Para poder ejecutar el ejecutable en el SLURM solo deberemos de usar el commando “**sbatch job.sub**“.

Enfoque de la paralelización

Como hemos podido ver en las anteriores prácticas la función que más nos ocupa tiempo del programa es “train”. Por la cual cosa nos centraremos en la paralelización solo de ésta.

Paralelizaciones

Primer for (Cálculo de “Hidden”)

La primera paralelización a implementar fué el cálculo de “Hidden”, este cálculo lo dividimos en dos Kernels para facilitar la implementación y poder especificar en cada caso el número de bloques y threads más adecuado para evitar el desuso de los threads restantes. Un Kernel hará el cálculo de “SumH” mientras que el segundo hará el cálculo de “Hidden”.

```
for( int j = 0 ; j < NUMHID ; j++ ) {    // compute hidden unit activations
    float SumH = 0.0;
    for( int i = 0 ; i < NUMIN ; i++ ) SumH += tSet[p][i] * WeightIH[j][i];
    Hidden[p][j] = 1.0/(1.0 + exp(-SumH)) ;
}
```

Imagen de la función original.

Antes de empezar con la implementación de los Kernels reservamos en la memoria de la gráfica las variables que posteriormente utilizaremos en nuestros Kernels. Una vez hecho también guardaremos dentro de dos de estas variables el valor inicial sobre el que empiezan a aplicarse los cálculos. Esto se consigue copiando el contenido de las variables iniciales del procesador en las variables de la gráfica.

```
//FOR 1
float *d_WeightIH;
float *d_Hidden;
char *d_tSet;
float *d_totalSumH;
float *d_SumH;

//Assignar espai en la memoria de la grafica
cudaMalloc((void **)&d_WeightIH, NUMHID*NUMIN*sizeof(float));
cudaMalloc((void **)&d_Hidden, NUMPAT*NUMHID*sizeof(float));
cudaMalloc((void **)&d_tSet, NUMPAT*NUMIN*sizeof(char));
cudaMalloc((void **)&d_totalSumH, 1*sizeof(float));
cudaMalloc((void **)&d_SumH, NUMHID*sizeof(float));
//Copiar Host to Device
cudaMemcpy(d_WeightIH, WeightIH, NUMHID*NUMIN*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_tSet, tSet, NUMPAT*NUMIN*sizeof(char), cudaMemcpyHostToDevice);
```

Imagen de la declaración de memoria en la GPU.

Cálculo de “SumH”

```
Kernel_for_1<<<NUMHID, NUMIN>>>(d_WeightIH, d_tSet, p, d_SumH);
```

Imagen de la llamada a la funcion de Kernel.

Esta función de Kernel la definiremos con tantos bloques como filas tiene la matriz y tantos threads como columnas.
En la llamada a esta función de Kernel le pasamos las variables que van a utilizarse ya sea para leer o escribir algún valor en ellas. Para empezar declaramos un array llamado “totalSumH” de tamaño “NUMIN” de tipo float el cual al ser shared será compartido a nivel de bloque. En cada una de las posiciones de este array cada thread guarda el cálculo de una posición de “WeightIH” * “tSet”. Para que cada thread utilice el índice correcto para acceder a las posiciones de los array hemos tenido en cuenta la ID del thread, la ID del bloque y el tamaño del array a recorrer.

```
__global__ void Kernel_for_1( float *d_WeightIH, char *d_tSet, int p, float *d_SumH)
{
  __shared__ float totalSumH[NUMIN]; //threads per block

  unsigned int indexi = threadIdx.x; //index per cada thread
  unsigned int indexj = blockIdx.x; //ID del block

  totalSumH[indexi] = d_WeightIH[indexj*NUMIN+indexi] * d_tSet[p*NUMIN+indexi];

  for(int stride = NUMIN/2; stride > 0; stride /=2 )
  {
    __syncthreads();
    if(indexi < stride ) totalSumH[indexi] += totalSumH[indexi + stride];
  }

  if(indexi == 0)
    d_SumH[indexj] = totalSumH[0];
}
```

Imagen de la función Kernel.

De esta forma tenemos en “totalSumH” los valores calculados de cada posición de los arrays pero cada bloque tiene los valores de la fila a la que corresponde su ID de bloque. De modo que nos queda hacer la suma de todos los valores del array por cada bloque y por último al estar la acumulación en la posición de “totalSumH[0]”, solo un thread de cada bloque guardará en la variable global “d_SumH[IdBloque]” el valor acumulado de su bloque.

Cálculo de “Hidden”

El cálculo de “Hidden” lo realizará otra función de Kernel, esta función sólo tendrá un bloque pero tendrá tantos threads como el tamaño de Hidden. Cada thread simplemente debe

acceder a una posición de la variable global “d_SumH” calculada anteriormente, realizar una serie de cálculos y guardar el valor en la posición correspondiente de “Hidden”.

```
__global__ void Kernel_SumH(float *d_SumH, float *d_Hidden, int p)
{
  unsigned int indexi = threadIdx.x;
  unsigned int indexj = blockIdx.x;

  d_Hidden[p*NUMHID+indexi] = 1.0/(1.0 + expf(-d_SumH[indexi]));
}
```

Imagen de la función Kernel.

Segundo for (Cálculo de “BError” y “DeltaO”)

```
for( int k = 0 ; k < NUMOUT ; k++ ) {    // compute output unit activations and errors
    float SumO = 0.0;
    for( int j = 0 ; j < NUMHID ; j++ ) SumO += Hidden[p][j] * WeightHO[k][j] ;
    Output[p][k] = 1.0/(1.0 + exp(-SumO)) ;   // Sigmoidal Outputs
    BError += 0.5 * (Target[p][k] - Output[p][k]) * (Target[p][k] - Output[p][k]) ;
    DeltaO[k] = (Target[p][k] - Output[p][k]) * Output[p][k] * (1.0 - Output[p][k]) ;
}
```

Imagen de la función original.

El siguiente for con el que nos encontramos es el cálculo de “BError” y “DeltaO”. Este bucle lo dividiremos en dos funciones Kernel, la primera de ellas hará el cálculo de “BError” (individualmente), “Output” y “DeltaO”. La segunda hará la acumulación y actualización de “BError”. Antes de empezar la implementación debemos reservar espacio en la gráfica y realizar la copia para la inicialización de algunas variables.

Cálculo de “Output”, “BError” y “DeltaO”

```
Kernel_for_2<<<NUMOUT, NUMHID>>>(d_WeightHO, d_Hidden, d_Target, d_Output, d_DeltaO, p, d_BError);
```

Imagen de la llamada a la funcion de Kernel.

En este caso hemos declarado de la misma forma que en el primer for un array llamado “totalSumO” compartido entre bloques para realizar el primer cálculo en el que cada thread accede a una posición de los array y realiza la multiplicación para guardarlo posteriormente en esta variable compartida. La acumulación de este valor en este caso es un poco más complicada que la anterior ya que no es divisible entre 2 para poder aplicar reduction. Nuestra estrategia busca la siguiente potencia de 2 del tamaño del array y realiza la acumulación con este tamaño.
Por último solo un thread de cada bloque realiza el cálculo de “d_Output”, “d_BError” y “d_DeltaO”.

```
__global__ void Kernel_for_2(float *d_WeightHO,float *d_Hidden, float *d_Target,float *d_Output, float *d_DeltaO,int p, float *d_BError)
{
  __shared__ float totalSumO[NUMHID]; // no es potencia de 2 (117) suma + 11

  unsigned int indexi = threadIdx.x; //index per cada thread
  unsigned int indexj = blockIdx.x; //ID del block

  unsigned int pow = 1;
  while (pow < NUMHID) pow *= 2;

  totalSumO[indexi] = d_Hidden[p*NUMHID+indexi] * d_WeightHO[indexj*NUMHID+indexi];

  for(int stride = pow/2; stride > 0; stride /=2 )
  {
    __syncthreads();
    if(indexi < stride)
      if((indexi+stride)<NUMHID)
        totalSumO[indexi] += totalSumO[indexi + stride];
  }

  if (indexi == 0)
  {
    d_Output[p*NUMOUT+indexj] = 1.0/(1.0 + expf(-totalSumO[0]));
    d_BError[indexj] = 0.5 * (d_Target[p*NUMOUT+indexj] - d_Output[p*NUMOUT+indexj]) * (d_Target[p*NUMOUT+indexj] - d_Output[p*NUMOUT+indexj]);
    d_DeltaO[indexj] = (d_Target[p*NUMOUT+indexj] - d_Output[p*NUMOUT+indexj]) * d_Output[p*NUMOUT+indexj] * (1.0 - d_Output[p*NUMOUT+indexj]);
  }
}
```

Imagen de la función Kernel.

Acumulación de “BError”

```
Kernel_BError<<<1, NUMOUT>>>(d_BError, d_BErrorAcc);
```

Imagen de la llamada a la funcion de Kernel.

Este Kernel tiene como finalidad la acumulación del array calculado anteriormente “d_BError” y la actualización con el valor de la iteración del patrón anterior. Nos encontramos con el mismo problema que con la implementación del reduction en el Kernel anterior. La estrategia adoptada sigue siendo la misma buscando la siguiente potencia de 2. Por último solo queda la actualización con el valor del “BError” de la iteración de los patrones anteriores.

```
__global__ void Kernel_BError(float *d_BError, float *d_BErrorAcc)
{
  __shared__ float totalBError[NUMOUT]; //multiple de 2

  unsigned int indexi = threadIdx.x;
  unsigned int indexj = blockIdx.x;

  totalBError[indexi] = d_BError[indexi];

  int pow = 1;
  while (pow < NUMOUT) pow *= 2;

  for(int stride = pow/2; stride > 0; stride /=2 )
  {
    __syncthreads();
    if(indexi < stride )
      if((indexi+stride) < NUMOUT)
        totalBError[indexi] += totalBError[indexi + stride];
  }

  if(indexi == 0)
  {
    totalBError[0] += d_BErrorAcc[0];
    d_BError[0] = totalBError[0];
    d_BErrorAcc[0] = d_BError[0];
  }

}
```

Imagen de la función Kernel.

Tercer for (Cálculo de “DeltaH” y “DeltaWeightIH”)

```
for( int j = 0 ; j < NUMHID ; j++ ) {     // update delta weights DeltaWeightIH
    float SumDOW = 0.0 ;
    for( int k = 0 ; k < NUMOUT ; k++ ) SumDOW += WeightHO[k][j] * DeltaO[k] ;
    DeltaH[j] = SumDOW * Hidden[p][j] * (1.0 - Hidden[p][j]) ;
    for( int i = 0 ; i < NUMIN ; i++ )
        DeltaWeightIH[j][i] = eta * tSet[p][i] * DeltaH[j] + alpha * DeltaWeightIH[j][i];
}
```

Imagen de la función original.

El siguiente for a analizar es el cálculo de “DeltaH” y “DeltaWeightIH”. Este for se implementará en dos funciones de Kernel.Una se encargará de calculará el valor de cada posición de “DeltaH”. La segunda calculará el nuevo valor de “DeltaWeightIH” a través del valor calculado de “DeltaH”.

Cálculo de “DeltaH”

```
Kernel_for_3<<<NUMHID, NUMOUT>>>(d_Hidden, d_DeltaH, p, d_WeightHO, d_DeltaO);
```

Imagen de la llamada a la funcion de Kernel.

En este Kernel hemos declarado de la misma forma que en el primer for un array llamado “totalSumDOW” compartido entre bloques para realizar el primer cálculo en el que cada thread accede a una posición de los array y realiza la multiplicación para guardarlo posteriormente en esta variable compartida. La acumulación de este valor en este caso es un poco más complicada que la anterior ya que no es divisible entre 2 para poder aplicar “reduction”. Nuestra estrategia busca la siguiente potencia de 2 del tamaño del array y realiza la acumulación con este tamaño.
Por último solo un thread de cada bloque realiza el cálculo de DeltaH.

```
__global__ void Kernel_for_3(float *d_Hidden, float *d_DeltaH, int p, float *d_WeightHO, float *d_DeltaO)
{
  __shared__ float totalSumDOW[NUMOUT];

  unsigned int indexi = threadIdx.x;
  unsigned int indexj = blockIdx.x;

  int pow = 1;
  while (pow < NUMOUT) pow *= 2;

  totalSumDOW[indexi] = d_WeightHO[indexi*NUMHID+indexj] * d_DeltaO[indexi];

  for(int stride = pow/2; stride > 0; stride /=2 )
  {
    __syncthreads();
    if(indexi < stride )
      if((indexi+stride) < NUMOUT)
        totalSumDOW[indexi] += totalSumDOW[indexi + stride];
  }

  if(indexi == 0)
    d_DeltaH[indexj] = totalSumDOW[0] * d_Hidden[p*NUMHID+indexj] * (1.0 - d_Hidden[p*NUMHID+indexj]);
}
```

Imagen de la función Kernel.

Cálculo de “DeltaWeightIH”

```
Kernel_for_3_2<<<NUMHID, NUMIN>>>(d_DeltaH, p, d_DeltaWeightIH, eta, alpha, d_tSet);
```

Imagen de la llamada a la funcion de Kernel.

En este Kernel calcularemos el nuevo valor de “DeltaWeightIH” a partir de la multiplicación y sumas de constantes y de “DeltaH” (calculado en el anterior Kernel).

```
__global__ void Kernel_for_3_2(float *d_DeltaH, int p,float *d_DeltaWeightIH, float eta, float alpha, char *d_tSet)
{
  unsigned int indexi = threadIdx.x;
  unsigned int indexj = blockIdx.x;
  unsigned int blockdim = blockDim.x;

  d_DeltaWeightIH[indexj*blockdim + indexi] =  eta * d_tSet[p*blockdim + indexi] * d_DeltaH[indexj] + alpha * d_DeltaWeightIH[indexj*blockdim + indexi];
}
```

Imagen de la función Kernel.

Cuarto for (Cálculo de “DeltaWeightHO”)

```
for( int k = 0 ; k < NUMOUT ; k ++ )    // update delta weights DeltaWeightHO
    for( int j = 0 ; j < NUMHID ; j++ )
        DeltaWeightHO[k][j] = eta * Hidden[p][j] * DeltaO[k] + alpha * DeltaWeightHO[k][j];
```

Imagen de la función original.

El siguiente for a analizar es el cálculo de “DeltaWeightHO”. Este for se implementará en una función de Kernel. Ésta se encargará de calcular el nuevo valor de cada posición de la matriz de “DeltaWeightIH”.

Cálculo de “DeltaWeightHO”

```
Kernel_for_4<<<NUMOUT, NUMHID>>>(d_Hidden, d_DeltaO, p, d_DeltaWeightHO, eta, alpha);
```

Imagen de la llamada a la funcion de Kernel.

En este Kernel hemos calculado directamente el nuevo valor de “DeltaWeightHO” a partir de la multiplicación y sumas de constantes y de “DeltaO” (calculado en el segundo for).

```
__global__ void Kernel_for_4(float *d_Hidden, float *d_DeltaO, int p, float *d_DeltaWeightHO, float eta, float alpha)
{
  int indexi = threadIdx.x;
  int indexj = blockIdx.x;
  int blockdim = blockDim.x;

  d_DeltaWeightHO[indexj*blockdim + indexi] =  eta * d_Hidden[p*blockdim + indexi] * d_DeltaO[indexj] + alpha * d_DeltaWeightHO[indexj*blockdim + indexi];
}
```

Imagen de la función Kernel.

Cálculo de “WeightIH”

```
for( int j = 0 ; j < NUMHID ; j++ )      // update weights WeightI
        for( int i = 0 ; i < NUMIN ; i++ )
            WeightIH[j][i] += DeltaWeightIH[j][i] ;
```

Imagen de la función original.

En este Kernel hemos calculado directamente el nuevo valor de “WeightIH” a partir de la suma de “DeltaWeightIH” (calculado en el tercer for).

Cálculo de “WeightIH”

```
Kernel_for_WeightIH<<< NUMHID, NUMIN>>>(d_WeightIH, d_DeltaWeightIH);
```

Imagen de la llamada a la funcion de Kernel.

```
__global__ void Kernel_for_WeightIH(float *d_WeightIH, float *d_DeltaWeightIH)
{
  unsigned int indexi = threadIdx.x;
  unsigned int indexj = blockIdx.x;

  d_WeightIH[indexj*NUMIN+indexi] += d_DeltaWeightIH[indexj*NUMIN+indexi];
}
```

Imagen de la función Kernel.

Problemas surgidos

Debido al nivel de dificultad de CUDA nos han surgido una serie de problemas.

En primer lugar hemos visto que la función de cuda “cudaMemcpy” nos hacía cuello de botella en la ejecución debido. Esto es debido a que teníamos declarada esta función dentro de los patrones y por lo tanto se ejecutaba muchas veces. Para solucionar esto hemos trasladado las funciones memcpy fuera de cada batch (50 patrones).

En segundo lugar hemos tenido problemas para saber si las funciones de Kernel nos calculaba bien los valores de las matrices y vectores. Para solucionar esto hemos copiado el resultado de la función del Kernel en la CPU, hemos sumado todos los valores dentro del mismo array y, finalmente, hemos comparado el resultado del programa en paralelo con el resultado del programa secuencial (original).

Para acabar, hemos tenido problema con las matrices, “tSet”, “WeightIH” y “WeightHO”. Éstas al estar declaradas como dinámicas, no podían verificar que la ejecución de los Kernels fuera siempre correcta. Para solucionar este problema hemos cambiado estas matrices de dinámicas a estáticas.

Posibles futuras mejoras

Debido a muchas adversidades de tiempo, no hemos podido implementar todas las optimizaciones que planeamos, por eso ahora mencionaremos todas las mejoras planeadas.

1. Aplicar openmp para la ejecución correspondiente a la CPU, es decir, donde se encuentran las inicializaciones de matrices. De esta forma evitariamos la secuencialidad en prácticamente todo el programa.
2. Uso de grids de dos dimensiones para calcular las funciones de Kernel. De esta forma podríamos adaptar nuestras funciones Kernel con matrices más grandes de 1024.
3. Uso de variables locales de cada thread para el “reduction”. De esta forma evitariamos los aproximadamente 5 ciclos de acceso a la memoria compartida que tenemos actualmente.

Resultados

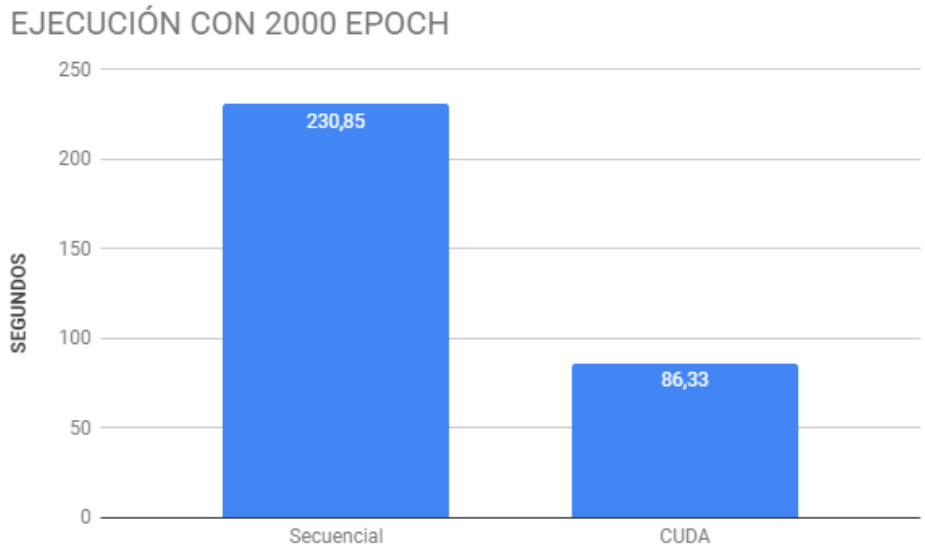

Imagen que muestra el tiempo de la ejecución de forma secuencial (original) y en paralelo (CUDA).

Tal y como podemos ver en la gráfica hemos conseguido un SpeedUpx2,67, aunque si aumentaramos la red neuronal podriamos una mayor mejora gracias al paralelismo de CUDA.