ECE 786

Programming assignment 3

Report

Thread Coarsening optimization:

This optimization aims at obtaining performance improvement through register reusing. In this technique the computations of 2 successive elements are carried out in the same thread, such that there is reuse of a register in the second computation. Hence the number of threads along a particular dimension reduces to half the initial number. The following changes were made in the code for thread coarsening:

```
m=m*2;                                  //reducing the number of threads along y-dimension
if (m<c_rows && n<c_cols)
{
  for(int p=0;p<=(j-1);p++)
 {
        for(int q=0;q<=(k-1);q++)
        {
            if(!((m-p)<0 || (n-q)<0 || (m-p)>=z || (n-q)>=i))          //For first element
            {
                c[(m*c_cols)+n]+=h[(p*k)+q]*a[((m-p)*i)+(n-q)];
                __syncthreads();
            }
            if(!(((m+1)-p)<0 || (n-q)<0 || ((m+1)-p)>=z || (n-q)>=i))    //For second element
            {
                c[((m+1)*c_cols)+n]+=h[(p*k)+q]*a[(((m+1)-p)*i)+(n-q)];
                __syncthreads();
            }
        }
  }
}
```

Performance:

Time taken for executing 500*500 A matrix without thread coarsening: 16us

Time taken for executing 500*500 A matrix with thread coarsening: 28us

Analysing global memory accesses:

Considering an input matrices A of dimension 5*5 and H of dimension 3*3 we get output matrix of dimension 7*7.

The thread block configuration is 32*32 with each element being of double type. Hence in each cache line 16 elements would be stored each of double type.

For warp 0 of threadblock 0:

Idx: 0~31

Idy:0

For matrix h:

Each element of h is accessed in the following manner for each computation: h[(p*k)+q]

Since the access of h is independent of idx or idy, therefore we obtain **coalesced** memory access for h matrix.

For matrix a:

Each element of a is accessed in the following manner for each computation: a[(idy-p)*a_cols+(idx-q)]

Consider p=0 & q=0. Since a_cols=5, idy=0 & idx=0~31,

Element of a=a[idx]= a[0~31]

Since each element is of double type, we get **uncoalesced** memory access for matrix a.

For matrix c:

Each element of a is accessed in the following manner for each computation: c[(idy*c_cols)+idx)]

Consider p=0 & q=0. Since c_cols=7, idy=0 & idx=0~31,

Element of c=c[idx]= c[0~31]

Since each element is of double type, we get **uncoalesced** memory access for matrix c.

Tiling Optimization

This optimization aims at obtaining performance improvement through cache reusing. In this technique submatrix of matrix a and h of dimensions Tile_size*Tile_size is loaded into the cache memory. The following changes were made in the code for tiling:

```
int n = blockIdx.x * Tile_size + threadIdx.x; //idx
int m = blockIdx.y * Tile_size + threadIdx.y; //idy
{
  __shared__ double H[Tile_size][Tile_size];
  __shared__ double A[Tile_size][Tile_size];
  if(m<j && n<k)
      H[m][n]=h[(k*m)+n];
  __syncthreads();
  if(m<i && n<z)
      A[m][n]=a[(i*m)+n];
  __syncthreads();
  if (m<c_rows && n<c_cols)
  {
      for(int p=0;p<=(Tile_size-1);p++)
      {
          for(int q=0;q<=(Tile_size-1);q++)
          {
              if(!((m-p)<0 || (n-q)<0 || (m-p)>=z || (n-q)>=i))
              {
                  c[(m*c_cols)+n]+=H[p][q]*A[(m-p)][(n-q)];
                  __syncthreads();
              }
          }
      }
  }
```

Performance:

Time taken for executing 16*16 A matrix without tiling: 16us

Time taken for executing 16*16 A matrix with tiling: 23us

Analysing global memory accesses:

Considering an input matrices A of dimension 5*5 and H of dimension 3*3 we get output matrix of dimension 7*7.

The thread block configuration is Tile_size*Tile_size, where the Tile_size is 16, with each element being of double type. Hence in each cache line 16 elements would be stored each of double type.

For warp 0 of threadblock 0:

Idx: 0~15

Idy:0~1

For matrix h:

Each element of h is accessed in the following manner for each computation: h[(h_cols*idy)+idx].

Since h_cols=3,

Element of h=h[3*idy+idx]

For idy =0: Elements of h=h[0~15]

For idy =1: Elements of h=h[3+0~15]

Therefore there would be 2 cache line accesses for accessing elements of matrix h. Hence we get **uncoalesced** memory access for matrix h.

For matrix a:

Each element of a is accessed in the following manner for each computation: a[(idy)*a_cols+(idx)]

Since a_cols=5,

Element of a=a[5*idy+idx]

For idy =0: Elements of a=a[0~15]

For idy =1: Elements of a=a[5+0~15]

Therefore there would be 2 cache line accesses for accessing elements of matrix a. Hence we get **uncoalesced** memory access for matrix a.

For matrix c:

Each element of a is accessed in the following manner for each computation: c[(idy*c_cols)+idx)]

Since c_cols=7,

Element of c=c[7*idy+idx]

For idy =0: Elements of c=c[0~15]

For idy =1: Elements of c=c[7+0~15]

Therefore there would be 2 cache line accesses for accessing elements of matrix c. Hence we get **uncoalesced** memory access for matrix c.

Analysing shared memory accesses:

Considering an input matrices A of dimension 5*5 and H of dimension 3*3 we get output matrix of dimension 7*7.

For warp 0 of threadblock 0:

Idx: 0~15

Idy:0~1

For matrix H:

Each element of H is accessed in the following manner for each computation: H[p][q].

Since the access of H is independent of idx or idy, therefore we obtain **no bank conflicts** for accessing elements of matrix H.

For matrix A:

Each element of A is accessed in the following manner for each computation: A[(idy-p)][(idx-q)].

Consider p=0 & q=0,

Elements of A= A[idy][idx]

For idy=0: Elements of A= A[0][0~15]

For idy=1: Elements of A= A[1][0~15]

Each shared memory bank has the capacity to store 4 bytes, hence 8 byte(type double) element of A would be stored in 2 consecutive banks. Hence we get **bank conflicts** for accessing elements of matrix A.