2/12/2018

Kadambari, SaiiDevesh

University of Massachusetts Lowell

Assignment-2

EECE 7110: High-Performance Comp. on GPUs

FLOW CHART: CPU GPU

Clear data allocated

J=J/2 until J>0

Threads (0-249)[thread -> column]

BLOCK\_1   
(Each block would cover 80 consecutive Rows of the Matrix A.)

Index<end(index++)

Load result to C

Add j-1 data + j/2 offset data.

Add j-1 data to j-2  
dec j=j-1.

t.x<j

J is odd

Load A\*B in shared memory and wait for threads

Shared memory\_250  
ROW=Block#  
tid=b\*no\_th+t.x  
step=#rows/#blocks  
index=block#\*step  
end= (block# +1)\*step

Perform the operations on GPU with 128 Blocks x 250 threads. (kernel call)

Copy the values from CPU to GPU memory

Allocate space in the heap(device) by the sizes given.

Allocate space in the heap(device) by the sizes given.

Input Random values to matrices A,B

Allocate space in the heap(host) by the sizes given.

no

yes

no

Yes

no

Load c\_device to host.wait until all threads are terminated .Print time

Yes

// allocating memory on heap in the RAM which is in CPU

int \*h\_a, \*h\_b, \*h\_c;// \*h\_cc;

cudaMallocHost((void \*\*) &h\_a, sizeof(int)\*m\*n); //dynamic allocation A=10240\*250

cudaMallocHost((void \*\*) &h\_b, sizeof(int)\*n\*k); // dynamic allocation B=250\*1

cudaMallocHost((void \*\*) &h\_c, sizeof(int)\*m\*k); // dynamic allocation C=10240\*1

/\* Insert random values with the help of rand() function from stdlib library into every element\*/

for (int i = 0; i < m; ++i) {

for (int j = 0; j < n; ++j) {

h\_a[i \* n + j] = rand() % 1024; //h\_a[row\_variable\*Max\_column + column\_variable]

}

}

/\* Insert random values with the help of rand() function from stdlib library into every element in B\*/

for (int i = 0; i < n; ++i) {

for (int j = 0; j < k; ++j) {

h\_b[i \* k + j] = rand() % 1024;

}

}

/\*start timer from time.h library\*/

clock\_t t;

t = clock();

/\*dynamically allocate data on heap in GPU DRAM\*/

int \*d\_a, \*d\_b, \*d\_c;

cudaMalloc((void \*\*) &d\_a, sizeof(int)\*m\*n); //dynamic allocation A=10240\*250 using pass by reference

cudaMalloc((void \*\*) &d\_b, sizeof(int)\*n\*k); // dynamic allocation B=250\*1

cudaMalloc((void \*\*) &d\_c, sizeof(int)\*m\*k); // dynamic allocation C=10240\*1

/\*Copy the dataelements from CPU to GPU through cudamemcpy \*/

cudaMemcpy(d\_a, h\_a, sizeof(int)\*m\*n, cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, h\_b, sizeof(int)\*n\*k, cudaMemcpyHostToDevice);

/\*Define your grid size(number of blocks) and block size(number of threads in a block) \*/

dim3 dimGrid(128);

dim3 dimBlock(256);

/\*Perform the operations on the GPU by the kernel call gpu\_matrix\_mult with grid size and block size along with arguments of matrix A and B \*/

gpu\_matrix\_mult<<<dimGrid, dimBlock>>>(d\_a, d\_b, d\_c, m, n, k);

/\*\_\_global\_\_ tells the cpu to execute the function on GPU void is the return type and the rest are arguments \*/

\_\_global\_\_ void gpu\_matrix\_mult(int \*a,int \*b, int \*c, int m, int n, int k)

/\*\_\_shared\_\_ tells the gpu that allotted memory per block(128) will be shared by all the threads\*/

\_\_shared\_\_ int smem[250];

/\*Each block takes care of its first 80 rows therefore blockid will be row id and each thread takes care of each column element . Step is defined as the ratio of data elements to number of blocks\*/

int row = blockIdx.x ;

int tid = blockIdx.x \* blockDim.x + threadIdx.x;

int step = m/gridDim.x; //step=80

int index\_begin = row \* step; // beginning of the row in each block

int index\_end= (row+ 1) \* step; // End of the row in each block

/\*This is initialized for parallel reduction purpose to store decimal value which is used to separate odd and even numbers\*/

float f=(blockDim.x)/2;int k\_b;

/\*wait until all the threads reach this point \*/

\_\_syncthreads();

/\*The for loop incremented for each row until the step is covered.Each iteration of 1st for loop represents each uniquie row. The multiplication of the elements of A and B is carried in the 1st loop. In the 2nd loop the parallel reduction of the elements is taken place. Each iteration would reduce the offset of the addition by half.When it cant be evenly reduced then the extra element(usually last of the offset) would be added to the previous element(the one before the offset). If it is evenly done then normal parallell reduction would take place\*/

for(int i=(index\_begin);i<index\_end;i++) //Row=0->80

{ smem[tid] =a[i \* (blockDim.x)+tid] \*b[tid]; //save multiplication value into the smem buffer

\_\_syncthreads(); //wait until all the threads reach this point

for(int j=((blockDim.x)/2);j>0;j=ceilf(f)) //i=250/2 is 125->62.5(63)->(63-1)->(21)->20->(10)->(5)->(3)->(2)->(1)

{ k\_b=2\*f;

if(((k\_b)%2!=0) && (threadIdx.x == (j-1)))

{

smem[threadIdx.x -1]+=smem[threadIdx.x]; // add extra element to previous element

j=j-1;

f=j;

}

if(threadIdx.x < j) //thread offset should be lessthan the offset

{ int temp =smem[threadIdx.x]+smem[threadIdx.x + j];

smem[threadIdx.x]=temp;

}

\_\_syncthreads();

f=f/2;

}

c[i]=smem[zero]; //store the value in the resultant matrix

}

}

/\*Copy resultant matrix from GPU to CPU\*/

cudaMemcpy(h\_c, d\_c, sizeof(int)\*m\*k, cudaMemcpyDeviceToHost);

/\*wait until all the threads have terminated successfully\*/

cudaThreadSynchronize();

/\*Stop the timer and print the timer value\*/  
t = clock()-t;

double time\_taken = ((double)t)/CLOCKS\_PER\_SEC;

printf("Time elapsed on matrix multiplication of %dx%d . %dx%d on CPU: %lf ms.\n\n", m, n, n, k, (time\_taken\*1000));

/\*Free allocated memory on device(GPU) and host(CPU)\*/

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

cudaFreeHost(h\_a);

cudaFreeHost(h\_b);

cudaFreeHost(h\_c);

/\*Exit the main thread with return 0 \*/

return 0;

}

OUTPUT:

