Листинг программы использующей глобальную память

#include <stdio.h>

#define BLOCK\_SIZE 16 // submatrix size

#define N 1536 // matrix size is N\*N

\_\_global\_\_ void matMult ( float \* a, float \* b, int n, float \* c )

{

int bx = blockIdx.x; // block index

int by = blockIdx.y;

int tx = threadIdx.x; // thread index

int ty = threadIdx.y;

float sum = 0.0f; // computed subelement

int ia = n \* BLOCK\_SIZE \* by + n \* ty; // a [i][0]

int ib = BLOCK\_SIZE \* bx + tx;

// Multiply the two matrices together;

for ( int k = 0; k < n; k++ )

sum += a [ia + k] \* b [ib + k\*n];

// Write the block sub-matrix to global memory;

// each thread writes one element

int ic = n \* BLOCK\_SIZE \* by + BLOCK\_SIZE \* bx;

c [ic + n \* ty + tx] = sum;

}

int main ( int argc, char \* argv [] )

{

int numBytes = N \* N \* sizeof ( float );

// allocate host memory

float \* a = new float [N\*N];

float \* b = new float [N\*N];

float \* c = new float [N\*N];

for ( int i = 0; i < N; i++ )

for ( int j = 0; j < N; j++ )

{

a [i] = 0.0f;

b [i] = 1.0f;

}

// allocate device memory

float \* adev = NULL;

float \* bdev = NULL;

float \* cdev = NULL;

cudaMalloc ( (void\*\*)&adev, numBytes );

cudaMalloc ( (void\*\*)&bdev, numBytes );

cudaMalloc ( (void\*\*)&cdev, numBytes );

// set kernel launch configuration

dim3 threads ( BLOCK\_SIZE, BLOCK\_SIZE );

dim3 blocks ( N / threads.x, N / threads.y);

// create cuda event handles

cudaEvent\_t start, stop;

float gpuTime = 0.0f;

cudaEventCreate ( &start );

cudaEventCreate ( &stop );

// asynchronously issue work to the GPU (all to stream 0)

cudaEventRecord ( start, 0 );

cudaMemcpy ( adev, a, numBytes, cudaMemcpyHostToDevice );

cudaMemcpy ( bdev, b, numBytes, cudaMemcpyHostToDevice );

matMult<<<blocks, threads>>> ( adev, bdev, N, cdev );

cudaMemcpy ( c, cdev, numBytes, cudaMemcpyDeviceToHost );

cudaEventRecord ( stop, 0 );

cudaEventSynchronize ( stop );

cudaEventElapsedTime ( &gpuTime, start, stop );

// print the cpu and gpu times

printf("time spent executing by the GPU: %.2f millseconds\n", gpuTime );

// release resources

cudaEventDestroy ( start );

cudaEventDestroy ( stop );

cudaFree ( adev );

cudaFree ( bdev );

cudaFree ( cdev );

delete a;

delete b;

delete c;

return 0;

}

Листинг программы использующей разделяемую память

#ifndef \_MATRIXMUL\_H\_

#define \_MATRIXMUL\_H\_

// Thread block size

#define BLOCK\_SIZE 16

#define MY\_SIZE 32

#define CPU\_THREAD\_COUNT 2

// Matrix dimensions

// (chosen as multiples of the thread block size for simplicity)

#define WA (MY\_SIZE \* BLOCK\_SIZE) // Matrix A width

#define HA (MY\_SIZE \* BLOCK\_SIZE) // Matrix A height

#define WB (MY\_SIZE \* BLOCK\_SIZE) // Matrix B width

#define HB WA // Matrix B height

#define WC WB // Matrix C width

#define HC HA // Matrix C height

#endif // \_MATRIXMUL\_H\_

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <math.h>

// includes, project

#include <cutil\_inline.h>

// includes, kernels

#include <matrixMul\_kernel.cu>

////////////////////////////////////////////////////////////////////////////////

// declaration, forward

void runTest(int argc, char\*\* argv);

void randomInit(float\*, int);

void printDiff(float\*, float\*, int, int);

extern "C"

void computeGold(unsigned int, float\*, const float\*, const float\*, unsigned int, unsigned int, unsigned int);

////////////////////////////////////////////////////////////////////////////////

// Program main

////////////////////////////////////////////////////////////////////////////////

int

main(int argc, char\*\* argv)

{

runTest(argc, argv);

cutilExit(argc, argv);

}

////////////////////////////////////////////////////////////////////////////////

//! Run a simple test for CUDA

////////////////////////////////////////////////////////////////////////////////

void

runTest(int argc, char\*\* argv)

{

if( cutCheckCmdLineFlag(argc, (const char\*\*)argv, "device") )

cutilDeviceInit(argc, argv);

else

cudaSetDevice( cutGetMaxGflopsDeviceId() );

// set seed for rand()

srand(2006);

// allocate host memory for matrices A and B

unsigned int size\_A = WA \* HA;

unsigned int mem\_size\_A = sizeof(float) \* size\_A;

float\* h\_A = (float\*) malloc(mem\_size\_A);

unsigned int size\_B = WB \* HB;

unsigned int mem\_size\_B = sizeof(float) \* size\_B;

float\* h\_B = (float\*) malloc(mem\_size\_B);

// initialize host memory

randomInit(h\_A, size\_A);

randomInit(h\_B, size\_B);

// allocate device memory

float\* d\_A;

cutilSafeCall(cudaMalloc((void\*\*) &d\_A, mem\_size\_A));

float\* d\_B;

cutilSafeCall(cudaMalloc((void\*\*) &d\_B, mem\_size\_B));

// copy host memory to device

cutilSafeCall(cudaMemcpy(d\_A, h\_A, mem\_size\_A,

cudaMemcpyHostToDevice) );

cutilSafeCall(cudaMemcpy(d\_B, h\_B, mem\_size\_B,

cudaMemcpyHostToDevice) );

// allocate device memory for result

unsigned int size\_C = WC \* HC;

unsigned int mem\_size\_C = sizeof(float) \* size\_C;

float\* d\_C;

cutilSafeCall(cudaMalloc((void\*\*) &d\_C, mem\_size\_C));

// allocate host memory for the result

float\* h\_C = (float\*) malloc(mem\_size\_C);

// create and start timer

unsigned int timer = 0;

cutilCheckError(cutCreateTimer(&timer));

cutilCheckError(cutStartTimer(timer));

// setup execution parameters

dim3 threads(BLOCK\_SIZE, BLOCK\_SIZE);

dim3 grid(WC / threads.x, HC / threads.y);

// execute the kernel

matrixMul<<< grid, threads >>>(d\_C, d\_A, d\_B, WA, WB);

// check if kernel execution generated and error

cutilCheckMsg("Kernel execution failed");

// copy result from device to host

cutilSafeCall(cudaMemcpy(h\_C, d\_C, mem\_size\_C,

cudaMemcpyDeviceToHost) );

// stop and destroy timer

cutilCheckError(cutStopTimer(timer));

printf("----------- %i ", WA);

printf("----------- \n");

printf("GPU processing time: %f (ms) \n", cutGetTimerValue(timer));

cutilCheckError(cutDeleteTimer(timer));

// malloc for reference solution

//float\* reference = (float\*) malloc(mem\_size\_C);

// // create and start timer

// timer = 0;

// cutilCheckError(cutCreateTimer(&timer));

// cutilCheckError(cutStartTimer(timer));

//// compute reference solution

//computeGold(0, reference, h\_A, h\_B, HA, WA, WB);

//computeGold(1, reference, h\_A, h\_B, HA, WA, WB);

//

//// stop and destroy timer

// cutilCheckError(cutStopTimer(timer));

// printf("1CPU processing time: %f (ms) \n", cutGetTimerValue(timer));

// cutilCheckError(cutDeleteTimer(timer));

// // check result

// CUTBoolean res = cutCompareL2fe(reference, h\_C, size\_C, 1e-6f);

// printf("Test %s \n", (1 == res) ? "PASSED" : "FAILED");

// if (res!=1) printDiff(reference, h\_C, WC, HC);

// clean up memory

free(h\_A);

free(h\_B);

free(h\_C);

// free(reference);

cutilSafeCall(cudaFree(d\_A));

cutilSafeCall(cudaFree(d\_B));

cutilSafeCall(cudaFree(d\_C));

cudaThreadExit();

}

// Allocates a matrix with random float entries.

void randomInit(float\* data, int size)

{

for (int i = 0; i < size; ++i)

data[i] = rand() / (float)RAND\_MAX;

}

void printDiff(float \*data1, float \*data2, int width, int height)

{

int i,j,k;

int error\_count=0;

for (j=0; j<height; j++) {

for (i=0; i<width; i++) {

k = j\*width+i;

if (data1[k] != data2[k]) {

printf("diff(%d,%d) CPU=%4.4f, GPU=%4.4f n", i,j, data1[k], data2[k]);

error\_count++;

}

}

}

printf(" nTotal Errors = %d n", error\_count);

}

#ifndef \_MATRIXMUL\_KERNEL\_H\_

#define \_MATRIXMUL\_KERNEL\_H\_

#include <stdio.h>

#include "matrixMul.h"

#define CHECK\_BANK\_CONFLICTS 0

#if CHECK\_BANK\_CONFLICTS

#define AS(i, j) cutilBankChecker(((float\*)&As[0][0]), (BLOCK\_SIZE \* i + j))

#define BS(i, j) cutilBankChecker(((float\*)&Bs[0][0]), (BLOCK\_SIZE \* i + j))

#else

#define AS(i, j) As[i][j]

#define BS(i, j) Bs[i][j]

#endif

////////////////////////////////////////////////////////////////////////////////

//! Matrix multiplication on the device: C = A \* B

//! wA is A's width and wB is B's width

////////////////////////////////////////////////////////////////////////////////

\_\_global\_\_ void

matrixMul( float\* C, float\* A, float\* B, int wA, int wB)

{

// Block index

int bx = blockIdx.x;

int by = blockIdx.y;

// Thread index

int tx = threadIdx.x;

int ty = threadIdx.y;

// Index of the first sub-matrix of A processed by the block

int aBegin = wA \* BLOCK\_SIZE \* by;

// Index of the last sub-matrix of A processed by the block

int aEnd = aBegin + wA - 1;

// Step size used to iterate through the sub-matrices of A

int aStep = BLOCK\_SIZE;

// Index of the first sub-matrix of B processed by the block

int bBegin = BLOCK\_SIZE \* bx;

// Step size used to iterate through the sub-matrices of B

int bStep = BLOCK\_SIZE \* wB;

// Csub is used to store the element of the block sub-matrix

// that is computed by the thread

float Csub = 0;

// Loop over all the sub-matrices of A and B

// required to compute the block sub-matrix

for (int a = aBegin, b = bBegin;

a <= aEnd;

a += aStep, b += bStep) {

// Declaration of the shared memory array As used to

// store the sub-matrix of A

\_\_shared\_\_ float As[BLOCK\_SIZE][BLOCK\_SIZE];

// Declaration of the shared memory array Bs used to

// store the sub-matrix of B

\_\_shared\_\_ float Bs[BLOCK\_SIZE][BLOCK\_SIZE];

// Load the matrices from device memory

// to shared memory; each thread loads

// one element of each matrix

AS(ty, tx) = A[a + wA \* ty + tx];

BS(ty, tx) = B[b + wB \* ty + tx];

// Synchronize to make sure the matrices are loaded

\_\_syncthreads();

// Multiply the two matrices together;

// each thread computes one element

// of the block sub-matrix

for (int k = 0; k < BLOCK\_SIZE; ++k)

Csub += AS(ty, k) \* BS(k, tx);

// Synchronize to make sure that the preceding

// computation is done before loading two new

// sub-matrices of A and B in the next iteration

\_\_syncthreads();

}

// Write the block sub-matrix to device memory;

// each thread writes one element

int c = wB \* BLOCK\_SIZE \* by + BLOCK\_SIZE \* bx;

C[c + wB \* ty + tx] = Csub;

}

#endif // #ifndef \_MATRIXMUL\_KERNEL\_H\_