The Ministry of Education and Science of the Russian Federation

Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program

of the Lobachevsky State University of Nizhni Novgorod

among the world's research and education centers

Strategic initiative

“Achieving leading positions in the field of supercomputer technology

and high-performance computing”

Introduction to GPU programming

Practice 7. Matrix multiplication

Nizhni Novgorod

2014

Author: S.I. Bastrakov

OBJECTIVES

The objective of this practice is to implement and analyze naive matrix multiplication algorithm, and implement optimized block algorithm using shared memory.

ABSTRACT

This practice considers naive and block matrix multiplication algorithms. We analyze both algorithms and explain how shared memory leads to significant speedup of block algorithm.

BRIEF OVERVIEW

We consider matrix multiplication of two rectangular matrices, and suppose number of rows and columns of all matrices is a multiple of 16.

Implementation on CPU is very straightforward (not it is very far from optimal on CPU):

```
void mmult(int m, int n, int k, const float * a, const float * b,
 float * c)
{
    for (int i = 0; i < m; ++i)
        for (int j = 0; j < k; ++j)
        {
            c[i * k + j] = 0;
            for (int l = 0; l < n; ++l)
                c[i * k + j] += a[i * n + l] * b[l * k + j];
        }
}
```

This implementation is easily transformed into a naïve kernel and the code to launch it:

```
__global__ void mmult_kernel(int m, int n, int k, const float * a, const
float * b, float * c)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    float sum = 0;
    for (int l = 0; l < n; ++l)
        sum += a[i * n + l] * b[l * k + j]; // sum += a(i, l) * b(l, j)
    c[i * k + j] = sum; // c(i, j) = sum
}
void mmult_gpu(int m, int n, int k, const float * a, const float * b, float *
c)
{
    dim3 blocks(m / BLOCK_SIZE, k / BLOCK_SIZE);
    dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
    mmult_kernel<<<blocks, threads>>>(m, n, k, a, b, c);
}
```

Let us analyze memory access pattern of the naïve implementation. Each thread reads a row of matrix A and a column of matrix B, overall 2n numbers. Thread block reads overall 2n * BLOCK_SIZE2 numbers. To compute a block of matrix C sized BLOCK_SIZE x

BLOCK_SIZE it is only required to read BLOCK_SIZE rows of A, and BLOCK_SIZE columns B, that is only 2n * BLOCK_SIZE numbers. This observation is a key to block algorithm.

In block algorithm each thread block computes a block of resulting matrix using shared memory. It is implemented in CUDA as follows:

```
__global__ void mmult_kernel_opt(int m, int n, int k, const float * a, const
float * b, float * c)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    __shared__ float a_block[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float b_block[BLOCK_SIZE][BLOCK_SIZE];
    float sum = 0;
    for (int p = 0; p < n / BLOCK_SIZE; ++p)  {
        a_block[threadIdx.x][threadIdx.y] = a[i * n + p * BLOCK_SIZE +
threadIdx.y];
        b_block[threadIdx.x][threadIdx.y] = b[(p * BLOCK_SIZE + threadIdx.x)
* k + j];
        __syncthreads();
        for (int l = 0; l < BLOCK_SIZE; ++l)
            sum += a_block[threadIdx.x][l] * b_block[l][threadIdx.y];
        __syncthreads();
    }
    c[i * k + j] = sum; // c(i, j) = sum
}
void mmult_gpu_opt(int m, int n, int k, const float * a, const float * b,
float * c)
{
    dim3 blocks(m / BLOCK_SIZE, k / BLOCK_SIZE);
    dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
    mmult_kernel_opt<<<blocks, threads>>>(m, n, k, a, b, c);
}
```

Block algorithm is superior over naïve for matrices that are large enough to not fit global memory cache due to putting frequently used data to shared memory. This is one of the basic optimization approaches on GPU.

FOR STUDENTS

Empirical evaluation of various performance optimization techniques is presented in [1]. Advanced optimization topics are covered in [2, 3].

REFERENCES

1. Farber R. CUDA Application Design and Development. – Morgan Kaufmann, 2011. – 336 p.
2. GPU Computing Gems Emerald Edition, ed. Wen-mei W. Hwu. – Morgan Kaufmann, 2011. – 886 p.
3. NVIDIA CUDA C Best Practices Guide [http://docs.nvidia.com/cuda/cuda-c-best-practices-guide#axzz3JRcPurfI]

INDIVIDUAL WORK

1. Modify block algorithm so that each thread computes several elements instead of 1.
2. Empirically find optimal block size.
3. Implement Strassen matrix multiplication algorithm using the developed block algorithm implementation for small enough matrices. Compare performance of Strassen and block algorithms depending on matrix size.