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 9. CUDA Libraries. Minimal residual method. Convolution

Nizhni Novgorod

2014

Author: S.I. Bastrakov

OBJECTIVES

The objective of this practice is to apply CUBLAS and CUFFT libraries to solve practically important problems: implement minimum residual method for iterative SLA solving and convolution of complex signals.

ABSTRACT

This practice considers two important algorithms: minimum residual method for iterative SLA solving and convolution of complex signals. We demonstrate how to use optimized CUDA libraries CUBLAS and CUFFT to solve these problems.

BRIEF OVERVIEW

Minimal residual method is an iterative method for solving linear systems Ax = b. If matrix A is positively defined, it converges from all initial points x0. Each iteration refines x as follows:

$$r = Ax - b$$

$$\tau = \frac{(Ar, r)}{(Ar, Ar)}$$

$$x = x - \tau r$$

Here is implementation of one iteration using CUBLAS:

```
void min_residual_gpu(int n, const float * A, const float * b,
                      float * x, float * r, float * Ar)
{
    cublasScopy(n, b, 1, r, 1); // r <- b
    cublasSgemv('N', n, n, 1.0f, A, n, x, 1, -1.0f, r, 1); // r <- Ax - r
    cublasSgemv('N', n, n, 1.0f, A, n, r, 1, 0, Ar, 1); // Ar
    float tau = cublasSdot(n, Ar, 1, r, 1) / cublasSdot(n, Ar, 1, Ar, 1); //
tau <- (Ar, r) / (Ar, Ar)
    // x <- x - tau * r
    cublasSaxpy(n, -tau, r, 1, x, 1);
}
```

In function main we allocate memory on GPU using CUBLAS routines and call iteration of the method in a loop:

```
    float * A_gpu, * b_gpu, * x_gpu;
    cublasAlloc(n * n, sizeof *A_gpu, (void **) &A_gpu);
    cublasAlloc(n, sizeof *b, (void **) &b_gpu);
    cublasAlloc(n, sizeof *x, (void **) &x gpu);
    cublasSetVector(n * n, sizeof *A_gpu, A, 1, A_gpu, 1);
    cublasSetVector(n, sizeof *b_gpu, b, 1, b_gpu, 1);
    cublasSetVector(n, sizeof *x_gpu, x, 1, x_gpu, 1);
    float * r_gpu, * Ar_gpu;
    cublasAlloc(n, sizeof *r_gpu, (void **) &r_gpu);
    cublasAlloc(n, sizeof *Ar_gpu, (void **) &Ar_gpu);
cublasSetVector(n, sizeof *Ar_gpu, x, 1, Ar_gpu, 1);
    for (int i = 0; i < iterations; ++i)
```

```
        min_residual_gpu(n, A_gpu, b_gpu, x_gpu, r_gpu, Ar_gpu);
    cublasGetVector(n, sizeof *x_gpu, x_gpu, 1, x, 1);
```

Convolution problem statement is as follows. There are 2 discrete complex signals of length n, a: $a(0), a(1), \dots, a(n-1)$, and b: $b(0), b(1), \dots, b(n-1)$. Signals are periodic with period n: $a(-m) = a(n-m), b(-m) = b(n-m)$. Cyclic convolution is defined as:

$$s = a * b$$

$$s(i) = \sum_{j=0}^{n-1} a(j)b(i-j)$$

Computing by definition has complexity $O(n^2)$. It can be done faster using DFT. Let A and B be Fourier images of a and b. Then image of convolution S is: $S(k) = A(k)B(k)$. Result s is found from S by inverse DFT. In case FFT is used, complexity is $O(n \log n)$.

Implementation of the naive algorithm on CPU is as follows:

```
void convolve(const cufftComplex * a, const cufftComplex * b, int n, cufft-
Complex * result) {
    for (int i = 0; i < n; ++i)     {
        result[i].x = 0;
        result[i].y = 0;
        for (int j = 0; j < n; ++j)         {
            int idx = ((j <= i) ? (i - j) : (n + i - j));
            result[i].x += a[j].x * b[idx].x - a[j].y * b[idx].y;
            result[i].y += a[j].x * b[idx].y + a[j].y * b[idx].x;
        }
    }
```

Implementation of FFT-based convolution using CUFFT:

```
__global__ void mult_scale(cufftComplex * a, const cufftComplex * b, int n,
float scale)
{
    const int num_threads = blockDim.x * gridDim.x;
    const int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    for (int k = thread_id; k < n; k += num_threads)
    {
        float ax = (a[k].x * b[k].x - a[k].y * b[k].y) * scale;
        float ay = (a[k].x * b[k].y + a[k].y * b[k].x) * scale;
        a[k].x = ax;
        a[k].y = ay;
    }
}

void convolve_gpu(cufftComplex * a, cufftComplex * b, int n, cufftComplex *
result)
{
    cufftHandle plan;
    cufftPlan1d(&plan, n, CUFFT_C2C, 1);
    cufftExecC2C(plan, a, a, CUFFT_FORWARD);
    cufftExecC2C(plan, b, b, CUFFT_FORWARD);
    mult_scale<<<4, 256>>>(a, b, n, 1.0f / n);
    cufftExecC2C(plan, a, result, CUFFT_INVERSE);
    cufftDestroy(plan);
}
```

FOR STUDENTS

Detailed information about CUBLAS is presented in [1], about CUFFT in [2].

REFERENCES

1. NVIDIA CUBLAS Documentation [http://docs.nvidia.com/cublas/index.html#axzz3JRcPurfI]
2. NVIDIA CUFFT Documentation [http://docs.nvidia.com/cufft/index.html#axzz3JRcPurfI]

INDIVIDUAL WORK

1. Modify implementation of minimal residual method so that stop condition is $||Ax - b|| < \varepsilon$.
2. Implement convolution algorithm via FFT on CPU using libraries fftw or Intel MKL. Compare performance with GPU implementation.