|  |  |
| --- | --- |
|  |  |

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:

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 , , and : . Signals are periodic with period : . Cyclic convolution is defined as:

Computing by definition has complexity . It can be done faster using DFT. Let and be Fourier images of and . Then image of convolution is: . Result is found from by inverse DFT. In case FFT is used, complexity is .

Implementation of the naive algorithm on CPU is as follows:

void convolve(const cufftComplex \* a, const cufftComplex \* b, int n, cufftComplex \* 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|| < ε.
2. Implement convolution algorithm via FFT on CPU using libraries fftw or Intel MKL. Compare performance with GPU implementation.