


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”

Lobachevsky State University of Nizhni Novgorod

Faculty of Computational mathematics and cybernetics

Introduction to GPU programming

09 Practice
CUDA Libraries. Minimal residual method. Convolution

Bastrakov S.I.
Software department

Contents

- Minimal residual method
- Convolution

Minimal residual method

Problem statement

- 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:

$$r \leftarrow Ax - b$$

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

$$x \leftarrow x - \tau r$$

Implementation 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);
}
```

Function main…

```
int main()
{
    const int n = 1024;
    const int iterations = 100;
    cublasInit();
    float * A, * b, * x;
    A = new float[n * n];
    b = new float[n];
    x = new float[n];
```

Function main…

```
for (int i = 0; i < n * n; ++i)
    A[i] = std::rand() / float(RAND_MAX);
for (int i = 0; i < n; ++i)
    A[i * n + i] += float(n);
for (int i = 0; i < n; ++i)
    b[i] = std::rand() / float(RAND_MAX);
for (int i = 0; i < n; ++i)
    x[i] = 0;
```

Function main…

```
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);
```

Function main…

```
 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);
float difference = 0;
for (int i = 0; i < n; ++i)
{
    float element = 0;
    for (int j = 0; j < n; ++j)
        element += A[j * n + i] * x[j];
    difference += (element - b[i]) * (element - b[i]);
}
```

Function main

```
    std::cout << std::fixed << std::setprecision(4);
    std::cout << "||Ax - b||^2 = " << difference << "\n";
    cublasFree(Ar_gpu);
    cublasFree(r_gpu);
    cublasFree(x_gpu);
    cublasFree(b_gpu);
    cublasFree(A_gpu);
    delete [] x;  delete [] b;  delete [] A;
    cublasShutdown();
    return 0;
}
```

Convolution

Problem statement

- 2 discrete complex signals of length n:

  a: $a(0), a(1), \ldots, a(n\text{-}1)$

  b: $b(0), b(1), \ldots, b(n\text{-}1)$

- Signals are periodic with period n:

  $a(-m) = a(n - m), b(-m) = b(n - m)$

- Cyclic convolution is defined as:

$$s = a * b$$

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

Convolution algorithm

- Computing by definition has complexity $O(n^2)$
- Computing using DFT:
  - Let A and B be Fourier images of a and b
  - Then image of convolution S is:

$$S(k) = A(k)B(k), k = \overline{0, n-1}$$

  - s is found from S by inverse DFT
  - In case FFT is used, complexity is $O(n*\log(n))$

Implementation of naive algorithm on CPU

```
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 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;
    }
}
```

Implementation using CUFFT

```
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);
}
```

Function main…

```
int main() {
    const int n = 1000;
    cufftComplex * a, * b, * a_gpu, * b_gpu, * result, *result_gpu,
        * result_verify;
    a = new cufftComplex[n];
    b = new cufftComplex[n];
    for (int i = 0; i < n; ++i)     {
        a[i].x = sin(6.28f * (float)i / n);
        a[i].y = cos(6.28f * (float)i / n);
        b[i].x = cos(6.0f * (float)i / n + 1);
        b[i].y = sin(6.0f * (float)i / n + 1);
    }
```

Function main…

```
    result = new cufftComplex[n];
    result_verify = new cufftComplex[n];
    cudaMalloc((void **) &a_gpu, n * sizeof *a_gpu);
    cudaMalloc((void **) &b_gpu, n * sizeof *b_gpu);
    cudaMalloc((void **) &result_gpu, n * sizeof *result_gpu);
    cudaMemcpy(a_gpu, a, n * sizeof *a_gpu,
cudaMemcpyHostToDevice);
    cudaMemcpy(b_gpu, b, n * sizeof *b_gpu,
cudaMemcpyHostToDevice);

    convolve(a, b, n, result);
```

Function main…

```
    convolve_gpu(a_gpu, b_gpu, n, result_gpu);
    cudaThreadSynchronize();
    cudaMemcpy(result_verify, result_gpu, n * sizeof *result_gpu,
        cudaMemcpyDeviceToHost);

    float difference = 0;
    for (int i = 0; i < n; ++i)
        difference += (result[i].x - result_verify[i].x) * (result[i].x -
result_verify[i].x) +
                      (result[i].y - result_verify[i].y) * (result[i].y -
result_verify[i].y);
```

Function main

```
if (difference < 1e-4f)
    std::cout << "Test passed.\n";
else
    std::cout << "Test failed.\n";
delete [] a;  delete [] b;
delete [] result;
delete [] result_verify;
cudaFree(a_gpu);
cudaFree(b_gpu);
cudaFree(result_gpu);
return 0;
}
```

Individual work

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

References

- NVIDIA CUBLAS Documentation [http://docs.nvidia.com/cublas/index.html#axzz3JRcPurfl]
- NVIDIA CUFFT Documentation [http://docs.nvidia.com/cufft/index.html#axzz3JRcPurfl]

Authors

- Bastrakov S.I.,
  Assistant of the Software department of CMC faculty.
  bastrakov@vmk.unn.ru