


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

03 Practice
Vector addition

Bastrakov S.I.
Software department

Contents

- Vector addition
- Implementation of axpy

Vector addition

Problem statement

- Addition of two single-precision floating-point vectors.
- We will create CPU and GPU implementations

Implementation on CPU

```
// y += x
void vecAdd(int n, float * x, float * y)
{
    for (int i = 0; i < n; ++i)
        y[i] += x[i];
}
```

Creating kernel

```
__global__ void vecAdd_kernel(int n, float * x, float * y)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        y[i] += x[i];
}
```

Calling kernel

```
void vecAdd_gpu(int n, float * x, float * y)
{
    const int threads_per_block = 256;
    int num_blocks = (n + threads_per_block - 1) /
threads_per_block;
    dim3 grid(num_blocks, 1, 1);
    dim3 blocks(threads_per_block, 1, 1);
    vecAdd_kernel<<<grid, blocks>>>(n, x, y);
}
```

Function main…

```
int main()
{
    const int n = 1000000;
    const int repeats = 1000;
    float * x, * y, * x_gpu, * y_gpu, * y_verify;
    x = new float[n];
    cudaMalloc((void **) &x_gpu, n * incx * sizeof(float));
    y = new float[n];
    y_verify = new float[n];
    cudaMalloc((void **) &y_gpu, n * incy * sizeof(float));
```

Function main…

```
    for (int i = 0; i < n; ++i)
        x[i] = float(rand()) / RAND_MAX;
    for (int i = 0; i < n; ++i)
        y[i] = float(rand()) / RAND_MAX;


    cudaMemcpy(x_gpu, x, n * sizeof(float),
cudaMemcpyHostToDevice);
    cudaMemcpy(y_gpu, y, n * sizeof(float),
cudaMemcpyHostToDevice);
```

Function main…

```
    LARGE_INTEGER before, after, freq;
    QueryPerformanceFrequency(&freq);
    std::cout << std::fixed << std::setprecision(3);
    QueryPerformanceCounter(&before);
    for (int i = 0; i < repeats; ++i)
        vecAdd(n, x, y);
    QueryPerformanceCounter(&after);
    double cpu_time = (after.QuadPart - before.QuadPart) /
double(freq.QuadPart);
    std::cout << "The CPU version took " << cpu_time << "
seconds.\n";
```

Function main…

```
    QueryPerformanceCounter(&before);
    for (int i = 0; i < repeats; ++i)
        vecAdd_gpu(n, x_gpu, y_gpu);
    cudaThreadSynchronize();
    QueryPerformanceCounter(&after);
    double gpu_time = (after.QuadPart - before.QuadPart) /
double(freq.QuadPart);
    std::cout << "The GPU version took " << gpu_time << "
seconds.\n";
    std::cout << "Speedup: " << cpu_time / gpu_time << "
times.\n";
```

Function main

```
    cudaMemcpy(y_verify, y_gpu, n * sizeof(float),
        cudaMemcpyDeviceToHost);
    float difference = 0;
    for (int i = 0; i < n; ++i)
        difference += (y[i] - y_verify[i]) * (y[i] - y_verify[i]);
    if (difference < 1e-5f) std::cout << "Test passed.\n";
    else  std::cout << "Test failed.\n";
    cudaFree(y_gpu);   delete [] y_verify;
    delete [] y;  cudaFree(x_gpu);  delete [] x;
    return 0;
}
```

Implementation of axpy

Problem statement

- Axpy is BLAS level 1 operation

*t*axpy(int n, fp a, fp x[], int incx, fp y[], int incy)

- *t* defines type denoted as fp (s — float, d — double, c — complex float, z — complex double).
- Operation is:
  $\forall i \in 0..n: y[i * incy] \leftarrow y[i * incy] + a * x[i * incx]$

Implementation on CPU

```
void saxpy(int n, float a, float * x, int incx, float * y, int incy)
{
    for (int i = 0; i < n; ++i)
        y[i * incy] += a * x[i * incx];
}
```

Creating kernel

```
__global__ void saxpy_kernel(int n, float a, float * x, int incx,
float * y, int incy)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        y[i * incy] += a * x[i * incx];
}
```

Calling kernel

```
void saxpy_gpu(int n, float a, float * x, int incx, float * y, int incy)
{
    const int threads_per_block = 256;
    int num_blocks = (n + threads_per_block - 1) /
threads_per_block;
    dim3 grid(num_blocks, 1, 1);
    dim3 blocks(threads_per_block, 1, 1);
    saxpy_kernel<<<grid, blocks>>>(n, a, x, incx, y, incy);
}
```

Function main…

```
int main()
{
    const int n = 1000000;
    const int incx = 2;
    const int incy = 3;
    const float alpha = float(rand()) / RAND_MAX;
    const int repeats = 1000;
    float * x, * y, * x_gpu, * y_gpu, * y_verify;
    x = new float[n * incx];
    cudaMalloc((void **) &x_gpu, n * incx * sizeof(float));
```

Function main…

```
    y = new float[n * incy];
    y_verify = new float[n * incy];
    cudaMalloc((void **) &y_gpu, n * incy * sizeof(float));
    for (int i = 0; i < n * incx; ++i)
        x[i] = float(rand()) / RAND_MAX;
    for (int i = 0; i < n * incy; ++i)
        y[i] = float(rand()) / RAND_MAX;
    cudaMemcpy(x_gpu, x, n * incx * sizeof(float),
cudaMemcpyHostToDevice);
    cudaMemcpy(y_gpu, y, n * incy * sizeof(float),
cudaMemcpyHostToDevice);
```

Function main…

```
    LARGE_INTEGER before, after, freq;
    QueryPerformanceFrequency(&freq);
    std::cout << std::fixed << std::setprecision(3);

    QueryPerformanceCounter(&before);
    for (int i = 0; i < repeats; ++i)
        saxpy(n, alpha, x, incx, y, incy);
    QueryPerformanceCounter(&after);
    double cpu_time = (after.QuadPart - before.QuadPart) /
double(freq.QuadPart);
    std::cout << "The CPU version took " << cpu_time << "
seconds.\n";
```

Function main…

```
    QueryPerformanceCounter(&before);
    for (int i = 0; i < repeats; ++i)
        saxpy_gpu(n, alpha, x_gpu, incx, y_gpu, incy);
    cudaThreadSynchronize();
    QueryPerformanceCounter(&after);
    double gpu_time = (after.QuadPart - before.QuadPart) /
double(freq.QuadPart);
    std::cout << "The GPU version took " << gpu_time << "
seconds.\n";
    std::cout << "Speedup: " << cpu_time / gpu_time << "
times.\n";
```

Function main…

```
cudaMemcpy(y_verify, y_gpu, n * incy * sizeof(float),
    cudaMemcpyDeviceToHost);
float difference = 0;
for (int i = 0; i < n * incy; ++i)
    difference += (y[i] - y_verify[i]) * (y[i] - y_verify[i]);
if (difference < 1e-5f)
    std::cout << "Test passed.\n";
else
    std::cout << "Test failed.\n";
```

Function main

```
    cudaFree(y_gpu);
    delete [] y_verify;
    delete [] y;
    cudaFree(x_gpu);
    delete [] x;
    return 0;
}
```

Individual work

- Implement axpy for other types of data: double, complex single, complex double.
- Modify axpy kernel so that it will work for any amount of block and threads per block.

References

- Sanders J., Kandrot E. CUDA by Example: An Introduction to General-Purpose GPU Programming. – Addison-Wesley Professional, 2010. – 312 p.
- NVIDIA CUDA C Programming Guide. [http://docs.nvidia.com/cuda/cuda-c-programming-guide/].

Authors

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