


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

07 Practice
Matrix multiplication

Bastrakov S.I.
Software department

Contents

- Naive algorithm
- Block algorithm

Problem statement

- Matrix multiplication of two rectangular matrices: C = AB
- Size of A is m x n, size of B is n x k
- m, n, k are multiples of 16

Naive algorithm

Implementation 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]; // c(i, j) += a(i, l) *
b(l, j)
        }
}
```

Creating kernel

```
__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
}
```

Calling kernel

```
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);
}
```

Function main…

```
int main()
{
    const int m = BLOCK_SIZE * 12, n = BLOCK_SIZE * 8, k =
BLOCK_SIZE * 14;
    const int repeats = 10;
    float * a = new float[m * n];
    float * a_gpu;
    cudaMalloc((void **) &a_gpu, m * n * sizeof(float));
    float * b = new float[n * k];
    float * b_gpu;
    cudaMalloc((void **) &b_gpu, n * k * sizeof(float));
```

Function main…

```
float * c = new float[m * k];
float * c_verify = new float[m * k];
float * c_gpu;
cudaMalloc((void **) &c_gpu, m * k * sizeof(float));

for (int i = 0; i < m * n; ++i)
    a[i] = float(rand()) / RAND_MAX;
for (int i = 0; i < n * k; ++i)
    b[i] = float(rand()) / RAND_MAX;
```

Function main…

```
    cudaMemcpy(a_gpu, a, m * n * sizeof(float),
cudaMemcpyHostToDevice);
    cudaMemcpy(b_gpu, b, n * k * sizeof(float),
cudaMemcpyHostToDevice);

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

Function main…

```
    QueryPerformanceCounter(&before);
    for (int i = 0; i < repeats; ++i)
        mmult(m, n, k, a, b, c);
    QueryPerformanceCounter(&after);
    double cpu_time = (after.QuadPart - before.QuadPart) /
double(freq.QuadPart);
    std::cout << "The CPU version took " << cpu_time << "
seconds.\n\n";
```

Function main…

```
    QueryPerformanceCounter(&before);
    for (int i = 0; i < repeats; ++i)
        mmult_gpu(m, n, k, a_gpu, b_gpu, c_gpu);
    cudaThreadSynchronize();
    QueryPerformanceCounter(&after);
    double gpu_time = (after.QuadPart - before.QuadPart) /
double(freq.QuadPart);
    std::cout << "The naive GPU version took " << gpu_time << "
seconds.\n";
    std::cout << "Speedup: " << cpu_time / gpu_time << "
times.\n";
```

Function main…

```
cudaMemcpy(c_verify, c_gpu, m * k * sizeof *c_gpu,
    cudaMemcpyDeviceToHost);
float difference = 0;
for (int i = 0; i < m * k; ++i)
    difference += (c[i] - c_verify[i]) * (c[i] - c_verify[i]);
if (difference < 1e-5f)
    std::cout << "Test passed.\n";
else
    std::cout << "Test failed (diff = " << difference << ").\n";
std::cout << "\n";
```

Function main

```
    cudaFree(c_gpu);
    delete [] c_verify;
    delete [] c;
    cudaFree(b_gpu);
    delete [] b;
    cudaFree(a_gpu);
    delete [] a;
}
```

Block algorithm

Analysis of naive algorithm

- Each thread reads a row of matrix A and a column of matrix B, overall $2n$ numbers.
- Thread block reads overall $2n$ * BLOCK_SIZE^2 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.

Block algorithm

$$\begin{pmatrix} A_{I1} & A_{I2} & \cdots & A_{IN} \end{pmatrix} \times \begin{pmatrix} B_{1J} \\ B_{2J} \\ \vdots \\ B_{NJ} \end{pmatrix} = \begin{pmatrix} C_{IJ} \end{pmatrix}$$

$$C_{IJ} = \sum_{p=0}^{N} A_{Ip} \times B_{pJ}$$

Creating kernel…

```
__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;
```

Creating kernel

```
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
}
```

Calling kernel

```
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);
}
```

Modification on function main…

```
    QueryPerformanceCounter(&before);
    for (int i = 0; i < repeats; ++i)
        mmult_gpu_opt(m, n, k, a_gpu, b_gpu, c_gpu);
    cudaThreadSynchronize();
    QueryPerformanceCounter(&after);
    double opt_gpu_time = (after.QuadPart - before.QuadPart) /
double(freq.QuadPart);
    std::cout << "The optimized GPU version took " <<
opt_gpu_time << " seconds.\n";
    std::cout << "Speedup: " << cpu_time / opt_gpu_time << "
times.\n";
```

Modification on function main

```
 cudaMemcpy(c_verify, c_gpu, m * k * sizeof *c_gpu,
    cudaMemcpyDeviceToHost);
difference = 0;
for (int i = 0; i < m * k; ++i)
    difference += (c[i] - c_verify[i]) * (c[i] - c_verify[i]);
if (difference < 1e-5f)
    std::cout << "Test passed.\n";
else
    std::cout << "Test failed (diff = " << difference << ").\n";
```

Individual work

- Modify block algorithm so that each thread computes several elements instead of 1.
- Empirically find optimal block size.
- 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.

References

- Farber R. CUDA Application Design and Development. – Morgan Kaufmann, 2011. – 336 p.
- GPU Computing Gems Emerald Edition, ed. Wen-mei W. Hwu. – Morgan Kaufmann, 2011. – 886 p.
- NVIDIA CUDA C Best Practices Guide [http://docs.nvidia.com/cuda/cuda-c-best-practices-guide#axzz3JRcPurfl]

Authors

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