МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»

Институт информатики, математики и электроники
Факультет информатики
Кафедра технической кибернетики

Параллельное программирование

Методические указания
к лабораторной работе №5

САМАРА 2017

Составитель: доц., Козлова Е.С.

УДК 519.681

Методические указания к лабораторным работам

Самарский национальный исследовательский университет имени академика С.П. Королева (Самарский университет)

Составитель: Е.С. Козлова

Самара, 2017. 6 с.

Методические указания предназначены для бакалавров направления 010400.62 «Прикладная математика и информатика»

Печатается по решению редакционно-издательского совета Самарского университета

Рецензент:

1 Библиотека CUBLAS

1.1 Описание библиотеки

CUBLAS — реализация интерфейса программирования приложений для создания библиотек, выполняющих основные операции линейной алгебры BLAS (Basic Linear Algebra Subprograms) для CUDA. Он позволяет получить доступ к вычислительным ресурсам графических процессоров NVIDIA. Библиотека является самодостаточной на уровне API, то есть, прямого взаимодействия с драйвером CUDA не происходит. CUBLAS прикрепляется к одному GPU и автоматически не распараллеливается между несколькими GPU.

Основные функции библиотеки CUBLAS: создание матриц и векторных объектов в пространстве памяти GPU, заполнение их данными, вызов последовательных функций CUBLAS, и загрузка результатов из области памяти GPU обратно к хосту. Чтобы достичь этого, CUBLAS предоставляет вспомогательные функции для создания и уничтожения объектов в памяти GPU, и для записи данных и извлечения информации из этих объектов.

Для максимальной совместимости с существующими средами Fortran, CUBLAS хранит матрицы *по столбцам* и использует индексирование с 1. Так как C и C++ используют построчное хранение, приложения не могут использовать родные семантики массивов для двумерных массивов. Вместо этого, макросы или встроенные функции должны быть определены для того чтобы использовать матрицы при помощи одномерных массивов. Для Fortran'а код портирован на C механическим способом, при котором сохраняется индексирование с 1. В этом случае индекс массива из матричного элемента в строке i и в столбце j могут быть вычислены с помощью следующего макроса:

```
#define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1))
```

Здесь ld — это размерности матрицы, в случае хранения в столбцах, является количеством строк. Для кода изначально написанного на C и C++, можно было бы использовать индексирование с 0, в этом случае макрос выглядит так:

```
#define IDX2C(i,j,ld) (((j)*(ld))+(i))
```

1.2 Основные функции библиотеки CUBLAS

Предоставляя функции для работы с элементами линейной алгебры, CUBLAS позволяет существенно упростить написание программы на CUDA. Более того, CUBLAS-решения самые эффективные на GPU. Рассмотрим основные операции CUBLAS.

```
cublasCreate(cublasHandle_t *handle);
```

инициализация объекта handle.

```
cublasDestroy(cublasHandle_t handle);
```

освобождение объекта handle.

```
cublasSetVector(int n, int elemSize, const void *x, int incx, void *y, int incy);
```

копирование n элементов из вектора x, хранящегося в памяти CPU, в вектор y в память GPU. Каждый элемент представляет собой elemSize байт. incx, incy – расстояния между соседними элементами в векторах x и y соответственно.

```
cublasGetVector(int n, int elemSize, const void *x, int incx, void *y, int incy);
```

копирование n элементов из вектора x, хранящегося в памяти GPU, в вектор y в память CPU. Каждый элемент представляет собой elemSize байт. incx, incy – расстояния между соседними элементами в векторах x и y соответственно.

```
cublasSetMatrix(int rows, int cols, int elemSize, const void *A, int lda, void *B, int ldb);
```

копирование rows × cols элементов матрицы A (CPU) в матрицу B (GPU). Каждый элемент представляет собой elemSize байт. Матрицы хранятся по столбцам. lda, ldb – основные размерности матриц A и B.

```
cublasGetMatrix(int rows, int cols, int elemSize, const void *A, int lda, void *B, int ldb);
```

копирование rows × cols элементов матрицы A (GPU) в матрицу B (CPU). Каждый элемент представляет собой elemSize байт. Матрицы хранятся по столбцам. lda, ldb – основные размерности матриц A и B.

Существуют асинхронные аналоги представленных функций. Для их вызова достаточно к имени соответствующей функции добавить постфикс Async.

```
cublasStatus_t cublas<DEFINE_TYPE>gemm(cublasHandle_t handle, cublasOperation_t transa, cublasOperation_t transb, int m, int n, int k, const TYPE *alpha, const TYPE *A, int lda, const TYPE *B, int ldb, const TYPE *beta, TYPE *C, int ldc);
```

производит операцию $C = \alpha\ transa(A)\ transb(B) + \beta C$, где transa – операция над матрицей A; transb – операция над матрицей B; α, β – некоторые числа; C – результирующая матрица (если $\beta \neq 0$, матрица C перезапишет результат в «саму себя», как следует из формулы); TYPE – тип единичного элемента.

Прочие полезные функции для работы с матрицами и векторами можно найти на официальном сайте NVidia: http://docs.nvidia.com/cuda/cublas.

Следует отметить, что копирование данных из памяти CPU в память GPU и обратно не обязательно осуществлять с использованием CUBLAS-функций для проведения над ними CUBLAS-операций. Допустимо использовать CUDA-функции (`cudaMalloc, cudaFree, cudaMemcpy`) для копирования данных из CPU в GPU и обратно. Главное не забывать, что для CUBLAS матрицы записываются по столбцам.

2 Задание на лабораторную работу

Лабораторная работа 5

С использованием библиотеки CUBLAS произвести умножение квадратных матриц размерностей N×N, сгенерированных случайно, где N = 64, 128, 256. Записать время выполнения умножений матриц в таблицу. Сделать выводы по полученным результатам.

Пример 1. Код main-функции

```
#include <cstdlib>
#include <curand.h>
#include <cublas_v2.h>

//GPU_fill_rand() - Функция случайной генерации матрицы
//gpu_blas_mmul() - Функция умножения матриц
//print_matrix() - Функция вывода матрицы

int main() {
    // Allocate 3 arrays on CPU
    int nr_rows_A, nr_cols_A, nr_rows_B, nr_cols_B, nr_rows_C, nr_cols_C;
    // for simplicity we are going to use square arrays
    nr_rows_A = nr_cols_A = nr_rows_B = nr_cols_B = nr_rows_C = nr_cols_C
= 3;
    float *h_A = (float *)malloc(nr_rows_A * nr_cols_A * sizeof(float));
    float *h_B = (float *)malloc(nr_rows_B * nr_cols_B * sizeof(float));
    float *h_C = (float *)malloc(nr_rows_C * nr_cols_C * sizeof(float));
    // Allocate 3 arrays on GPU
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A,nr_rows_A * nr_cols_A * sizeof(float));
    cudaMalloc(&d_B,nr_rows_B * nr_cols_B * sizeof(float));
    cudaMalloc(&d_C,nr_rows_C * nr_cols_C * sizeof(float));
    // Fill the arrays A and B on GPU with random numbers
    GPU_fill_rand(d_A, nr_rows_A, nr_cols_A);
    GPU_fill_rand(d_B, nr_rows_B, nr_cols_B);
    // Optionally we can copy the data back on CPU and print the arrays
    cudaMemcpy(h_A,d_A,nr_rows_A * nr_cols_A *
sizeof(float),cudaMemcpyDeviceToHost);
    cudaMemcpy(h_B,d_B,nr_rows_B * nr_cols_B *
sizeof(float),cudaMemcpyDeviceToHost);
    std::cout << "A =" << std::endl;
    print_matrix(h_A, nr_rows_A, nr_cols_A);
    std::cout << "B =" << std::endl;
    print_matrix(h_B, nr_rows_B, nr_cols_B);
    // Multiply A and B on GPU
    gpu_blas_mmul(d_A, d_B, d_C, nr_rows_A, nr_cols_A, nr_cols_B);
    // Copy (and print) the result on host memory
    cudaMemcpy(h_C,d_C,nr_rows_C * nr_cols_C *
sizeof(float),cudaMemcpyDeviceToHost);
    std::cout << "C =" << std::endl;
    print_matrix(h_C, nr_rows_C, nr_cols_C);

    //Free GPU memory
    cudaFree(d_A);
```

```
    cudaFree(d_B);
    cudaFree(d_C);

    // Free CPU memory
    free(h_A);
    free(h_B);
    free(h_C);
    return 0;
}
```

Пример 2. Код случайной генерации матрицы

```
// Fill the array A(nr_rows_A, nr_cols_A) with random numbers on GPU
void GPU_fill_rand(float *A, int nr_rows_A, int nr_cols_A) {
    // Create a pseudo-random number generator
    curandGenerator_t prng;
    curandCreateGenerator(&prng, CURAND_RNG_PSEUDO_DEFAULT);
    // Set the seed for the random number generator using the system clock
    curandSetPseudoRandomGeneratorSeed(prng, (unsigned long long)
clock());

    // Fill the array with random numbers on the device
    curandGenerateUniform(prng, A, nr_rows_A * nr_cols_A);
}
```

Пример 3. Код умножения матриц

```
// Multiply the arrays A and B on GPU and save the result in C
// C(m,n) = A(m,k) * B(k,n)
void gpu_blas_mmul(const float *A, const float *B, float *C, const int m,
const int k, const int n) {
    int lda=m,ldb=k,ldc=m;
    const float alf = 1;
    const float bet = 0;
    const float *alpha = &alf;
    const float *beta = &bet;
    // Create a handle for CUBLAS
    cublasHandle_t handle;
    cublasCreate(&handle);
    // Do the actual multiplication
    cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, n, k, alpha, A, lda,
B, ldb, beta, C, ldc);
    // Destroy the handle
    cublasDestroy(handle);
}
```