Министерство цифрового развития, связи
и массовых коммуникаций Российской Федерации

Сибирский государственный университет
телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

Расчетно-графическое задание

По дисциплине: «Программирование графических процессоров»

Выполнили:

Студенты 3 курса группы ИП-211
Лацук А. Ю.
Оганесян А. С.

Проверил:

Профессор кафедры ПМиК
Малков Е.А.

Новосибирск, 2025

Задание:

1. Реализовать произведение двух матриц, используя:
    - CUDA API
    - CUDA Driver API (C/C++)
    - CUDA Driver API (Python)
    - Numba
    - PyCuda
2. Сравнить время вычислений и производительность.

Цель:

Знакомство с CUDA Driver API и PyCuda

Оборудование:

Видеокарта RTX 3050TI

Выполнение работы:

В группе было проведено распределение работы по написанию программ:

- Оганесян Альберт – реализация на CUDA API (наивное ядро); реализация на CUDA Driver API (C++ и Python) с оптимизацией через shared memory.
- Лацук Андрей – генерация PTX-файла для ядра, реализация с использованием Numba (shared memory, автоматическое управление памятью); интеграция PyCUDA с загрузкой PTX-модуля; написание скрипта run.sh для автоматизации компиляции и запуска всех реализаций.

1. Подготовка окружения

Для упрощения работы и тестирования был написан скрипт run.sh, который компилирует .cu и .cpp файлы в папку obj/, запускает последовательно все реализации и создает виртуальное окружение для Python-зависимостей.

```
#!/bin/bash

if [ ! -d obj ]; then
   mkdir -p obj
```

```
fi

nvcc -ptx -O3 -arch=sm_86 matrix_mul.cu -o obj/matrix_mul.ptx

nvcc cuda_api.cu -o obj/cuda_api -arch=sm_86
./obj/cuda_api

nvcc driver_api.cpp -o obj/driver_api -lcuda -lcudart -arch=sm_86
./obj/driver_api

python3 driver_api.py

python3 -m venv ~/numba_env
source ~/numba_env/bin/activate

python3 numba_impl.py

python3 pycuda_impl.py

deactivate
```

Листинг 1 – скрипт run.sh

Также для работы с Driver API и PyCuda был написан PTX модуль

```
#define BLOCK_SIZE 16

extern "C" __global__ void matrixMul(float *A, float *B, float *C, int N)
{
   __shared__ float sA[BLOCK_SIZE][BLOCK_SIZE];
   __shared__ float sB[BLOCK_SIZE][BLOCK_SIZE];

   int bx = blockIdx.x;
   int by = blockIdx.y;
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   int row = by * BLOCK_SIZE + ty;
   int col = bx * BLOCK_SIZE + tx;

   float sum = 0.0f;

   for (int m = 0; m < (N + BLOCK_SIZE - 1) / BLOCK_SIZE; ++m)
   {
       int tiled_col = m * BLOCK_SIZE + tx;
       int tiled_row = m * BLOCK_SIZE + ty;

       sA[ty][tx] = (row < N && tiled_col < N) ? A[row * N + tiled_col] : 0.0f;
       sB[ty][tx] = (tiled_row < N && col < N) ? B[tiled_row * N + col] : 0.0f;
```

```
        __syncthreads();

        for (int k = 0; k < BLOCK_SIZE; ++k)
        {
            sum += sA[ty][k] * sB[k][tx];
        }

        __syncthreads();
    }

    if (row < N && col < N)
        C[row * N + col] = sum;
}
```

Листинг 2 – matrix_mul.cu

Для тестирования и корректной работы реализаций на Python был модифицирован файл cuda_driver.py

```
# Добавленные строки
cuEventCreate = cuda.cuEventCreate
cuEventCreate.argtypes = [POINTER(c_void_p), c_uint]
cuEventCreate.restype = int

cuEventRecord = cuda.cuEventRecord
cuEventRecord.argtypes = [c_void_p, c_void_p]
cuEventRecord.restype = int

cuEventSynchronize = cuda.cuEventSynchronize
cuEventSynchronize.argtypes = [c_void_p]
cuEventSynchronize.restype = int

cuEventElapsedTime = cuda.cuEventElapsedTime
cuEventElapsedTime.argtypes = [POINTER(c_float), c_void_p, c_void_p]
cuEventElapsedTime.restype = int

cuGetErrorString = cuda.cuGetErrorString
cuGetErrorString.argtypes = [c_int, POINTER(c_char_p)]
cuGetErrorString.restype = int

cuModuleLoadData = cuda.cuModuleLoadData
cuModuleLoadData.argtypes = [POINTER(c_void_p), c_void_p]
cuModuleLoadData.restype = int

cuEventDestroy = cuda.cuEventDestroy
cuEventDestroy.argtypes = [c_void_p]
cuEventDestroy.restype = int
```

```
cuModuleUnload = cuda.cuModuleUnload
cuModuleUnload.argtypes = [c_void_p]
cuModuleUnload.restype = int
```

Листинг 3 – модификация cuda_driver.py

2. Реализации

- **CUDA API (Оганесян):** Наивное ядро без shared memory.
- **CUDA Driver API (Оганесян):**
  - C++: Ручное управление контекстом, памятью и загрузкой PTX.
  - Python: Использование обёртки cuda_driver.py для вызовов CUDA Driver API.
- **Numba (Лацук):** Декоратор @cuda.jit, shared memory, автоматическое копирование данных.
- **PyCUDA (Лацук):** Загрузка PTX-модуля, использование gpuarray для передачи данных.

3. Тестирование

Замер времени выполнения для матрицы 512×512, размер блока – 16.

Объяснение реализаций

CUDA API

Наивное ядро с глобальной памятью. Компиляция через nvcc.

```
#include <cstdio>
#include <cstdlib>
#include <cuda_runtime.h>

#define BLOCK_SIZE 16

__global__ void matrixMul(float *A, float *B, float *C, int N)
{
   int row = blockIdx.y * blockDim.y + threadIdx.y;
   int col = blockIdx.x * blockDim.x + threadIdx.x;

   if (row < N && col < N)
   {
```

```
        float sum = 0.0f;
        for (int k = 0; k < N; k++)
        {
            sum += A[row * N + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}

int main()
{
    int N = 512;
    size_t size = N * N * sizeof(float);

    float *h_A = (float *)malloc(size);
    float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);

    for (int i = 0; i < N * N; i++)
    {
        h_A[i] = 1.0f;
        h_B[i] = 1.0f;
    }

    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    dim3 block(BLOCK_SIZE, BLOCK_SIZE);
    dim3 grid((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (N + BLOCK_SIZE - 1) / BLOCK_SIZE);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    matrixMul<<<grid, block>>>(d_A, d_B, d_C, N);

    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    printf("CUDA API \t\t\tTime: %.3f ms\n", milliseconds);

    cudaFree(d_A);
```

```
    cudaFree(d_B);
    cudaFree(d_C);
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```

Листинг 4 – cuda_api.cu

CUDA Driver API (C++)

Оптимизация через shared memory. Низкоуровневое управление через CUDA API.

```
#include <cstdio>
#include <cuda.h>
#include <cuda_runtime.h>
#include <driver_types.h>
#include <cmath>

#define BLOCK_SIZE 16
const int N = 512;

void checkCUDAError(CUresult res, const char *msg)
{
    if (res != CUDA_SUCCESS)
    {
        const char *errorStr;
        cuGetErrorString(res, &errorStr);
        printf("Ошибка: %s (%s)\n", msg, errorStr);
        exit(1);
    }
}

int main()
{
    CUresult res;

    res = cuInit(0);
    checkCUDAError(res, "cuInit");

    CUdevice device;
    res = cuDeviceGet(&device, 0);
    checkCUDAError(res, "cuDeviceGet");

    CUcontext context;
    res = cuCtxCreate(&context, 0, device);
    checkCUDAError(res, "cuCtxCreate");
```

```
float *h_A = new float[N * N];
float *h_B = new float[N * N];
float *h_C = new float[N * N];
float *h_C_ref = new float[N * N];

for (int i = 0; i < N * N; i++)
{
    h_A[i] = 1.0f;
    h_B[i] = 1.0f;
    h_C_ref[i] = N;
}

CUdeviceptr d_A, d_B, d_C;
res = cuMemAlloc(&d_A, N * N * sizeof(float));
checkCUDAError(res, "cuMemAlloc d_A");
res = cuMemAlloc(&d_B, N * N * sizeof(float));
checkCUDAError(res, "cuMemAlloc d_B");
res = cuMemAlloc(&d_C, N * N * sizeof(float));
checkCUDAError(res, "cuMemAlloc d_C");

res = cuMemcpyHtoD(d_A, h_A, N * N * sizeof(float));
checkCUDAError(res, "cuMemcpyHtoD d_A");
res = cuMemcpyHtoD(d_B, h_B, N * N * sizeof(float));
checkCUDAError(res, "cuMemcpyHtoD d_B");

CUmodule module;
res = cuModuleLoad(&module, "obj/matrix_mul.ptx");
checkCUDAError(res, "cuModuleLoad");

CUfunction kernel;
res = cuModuleGetFunction(&kernel, module, "matrixMul");
checkCUDAError(res, "cuModuleGetFunction");

dim3 block(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid((N + block.x - 1) / block.x, (N + block.y - 1) / block.y);

void *args[] = {(void *)&d_A, (void *)&d_B, (void *)&d_C, (void *)&N};

CUevent start, stop;
cuEventCreate(&start, 0);
cuEventCreate(&stop, 0);
cuEventRecord(start, 0);

res = cuLaunchKernel(kernel,
                     grid.x, grid.y, 1,
                     block.x, block.y, 1,
                     0, 0, args, 0);
checkCUDAError(res, "cuLaunchKernel");
```

```
    cuCtxSynchronize();
    cuEventRecord(stop, 0);
    cuEventSynchronize(stop);

    float ms;
    cuEventElapsedTime(&ms, start, stop);
    printf("CUDA Driver API (C/C++) \tTime: %.3f ms\n", ms);

    res = cuMemcpyDtoH(h_C, d_C, N * N * sizeof(float));
    checkCUDAError(res, "cuMemcpyDtoH");

    cuMemFree(d_A);
    cuMemFree(d_B);
    cuMemFree(d_C);
    delete[] h_A;
    delete[] h_B;
    delete[] h_C;
    delete[] h_C_ref;

    return 0;
}
```

Листинг 5 – driver_api.cpp

CUDA Driver API (Python)

Обёртка над CUDA Driver API на Python. Использует ctypes для вызовов.

```
import numpy as np
from ctypes import c_void_p, c_float, c_int, c_char_p, POINTER, byref, cast
import cuda_driver as cuda  # твоя обёртка над CUDA Driver API

# Константы
CUDA_SUCCESS = cuda.CUDA_SUCCESS
BLOCK_SIZE = 16
N = 512

def check_cuda_error(err_code):
    if err_code != CUDA_SUCCESS:
        err_str = c_char_p()
        cuda.cuGetErrorString(err_code, byref(err_str))
        raise RuntimeError(f"CUDA ошибка: {err_str.value.decode()}")

def main():
    check_cuda_error(cuda.cuInit(0))

    device = c_int()
    check_cuda_error(cuda.cuDeviceGet(byref(device), 0))
```

```
    context = c_void_p()
    check_cuda_error(cuda.cuCtxCreate(byref(context), 0, device))

    A = np.ones((N, N), dtype=np.float32)
    B = np.ones((N, N), dtype=np.float32)
    C_host = np.zeros((N, N), dtype=np.float32)

    d_A = c_void_p()
    d_B = c_void_p()
    d_C = c_void_p()
    data_size = N * N * np.dtype(np.float32).itemsize

    check_cuda_error(cuda.cuMemAlloc(byref(d_A), data_size))
    check_cuda_error(cuda.cuMemAlloc(byref(d_B), data_size))
    check_cuda_error(cuda.cuMemAlloc(byref(d_C), data_size))

    check_cuda_error(cuda.cuMemcpyHtoD(d_A, A.ctypes.data, data_size))
    check_cuda_error(cuda.cuMemcpyHtoD(d_B, B.ctypes.data, data_size))

    module = c_void_p()
    with open("obj/matrix_mul.ptx", "rb") as f:
        ptx_data = f.read()
    check_cuda_error(cuda.cuModuleLoadData(byref(module), ptx_data))

    kernel_func = c_void_p()
    check_cuda_error(cuda.cuModuleGetFunction(byref(kernel_func), module,
b"matrixMul"))

    grid_x = (N + BLOCK_SIZE - 1) // BLOCK_SIZE
    grid_y = (N + BLOCK_SIZE - 1) // BLOCK_SIZE

    N_cint = c_int(N)
    args = [d_A, d_B, d_C, N_cint]

    kernel_params = (c_void_p * len(args))(
        cast(byref(args[0]), c_void_p),
        cast(byref(args[1]), c_void_p),
        cast(byref(args[2]), c_void_p),
        cast(byref(args[3]), c_void_p)
    )

    start_event = c_void_p()
    end_event = c_void_p()
    check_cuda_error(cuda.cuEventCreate(byref(start_event), 0))
    check_cuda_error(cuda.cuEventCreate(byref(end_event), 0))

    check_cuda_error(cuda.cuEventRecord(start_event, 0))
    check_cuda_error(cuda.cuLaunchKernel(
        kernel_func,
        grid_x, grid_y, 1,
```

```
        BLOCK_SIZE, BLOCK_SIZE, 1,
        0, 0,
        kernel_params, 0
    ))
    check_cuda_error(cuda.cuEventRecord(end_event, 0))
    check_cuda_error(cuda.cuEventSynchronize(end_event))

    time_ms = c_float()
    check_cuda_error(cuda.cuEventElapsedTime(byref(time_ms), start_event, end_event))
    print(f"CUDA Driver API (Python) \tTime: {time_ms.value:.3f} ms")

    check_cuda_error(cuda.cuMemcpyDtoH(C_host.ctypes.data, d_C, data_size))

    check_cuda_error(cuda.cuMemFree(d_A))
    check_cuda_error(cuda.cuMemFree(d_B))
    check_cuda_error(cuda.cuMemFree(d_C))
    check_cuda_error(cuda.cuEventDestroy(start_event))
    check_cuda_error(cuda.cuEventDestroy(end_event))
    check_cuda_error(cuda.cuModuleUnload(module))
    check_cuda_error(cuda.cuCtxDestroy(context))

if __name__ == "__main__":
    main()
```

Листинг 6 – driver_api.py

Numba

Автоматизация работы с GPU через декораторы. Использует shared memory.

```
import numpy as np
from numba import cuda, float32
import time
import os

os.environ['NUMBA_ENABLE_CUDASIM'] = '0'
os.environ['NUMBA_CUDA_DEBUGINFO'] = '0'

BLOCK_SIZE = 16

@cuda.jit
def matrixMul_optimized(A, B, C, N):
    sA = cuda.shared.array((BLOCK_SIZE, BLOCK_SIZE), dtype=float32)
    sB = cuda.shared.array((BLOCK_SIZE, BLOCK_SIZE), dtype=float32)

    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bx = cuda.blockIdx.x
    by = cuda.blockIdx.y
```

```
    row = by * BLOCK_SIZE + ty
    col = bx * BLOCK_SIZE + tx
    tmp = 0.0

    for m in range(N // BLOCK_SIZE):
        sA[ty, tx] = A[row, m * BLOCK_SIZE + tx]
        sB[ty, tx] = B[m * BLOCK_SIZE + ty, col]

        cuda.syncthreads()

        for k in range(BLOCK_SIZE):
            tmp += sA[ty, k] * sB[k, tx]

        cuda.syncthreads()

    C[row, col] = tmp


def main():
    N = 512
    assert N % BLOCK_SIZE == 0, "Размер должен быть кратен BLOCK_SIZE"

    A = np.ones((N, N), dtype=np.float32)
    B = np.ones((N, N), dtype=np.float32)
    C = np.zeros((N, N), dtype=np.float32)

    d_A = cuda.to_device(A)
    d_B = cuda.to_device(B)
    d_C = cuda.device_array_like(C)

    threads_per_block = (BLOCK_SIZE, BLOCK_SIZE)
    blocks_per_grid = (N // BLOCK_SIZE, N // BLOCK_SIZE)

    matrixMul_optimized[blocks_per_grid, threads_per_block](d_A, d_B, d_C, N)
    cuda.synchronize()

    start = cuda.event()
    end = cuda.event()
    start.record()

    matrixMul_optimized[blocks_per_grid, threads_per_block](d_A, d_B, d_C, N)

    end.record()
    end.synchronize()
    elapsed_ms = cuda.event_elapsed_time(start, end)

    d_C.copy_to_host(C)
    expected = N
    correct = np.allclose(C, expected, atol=1e-3)
```

```
   print(f"Numba \t\t\t\tTime: {elapsed_ms:.3f} ms")


if __name__ == "__main__":
   main()
```

Листинг 7 – numba_impl.py

PyCUDA

Высокоуровневая библиотека для Python. Интеграция с PTX-модулем.

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy as np
from pycuda import gpuarray

N = 512
BLOCK_SIZE = 16

mod = drv.module_from_file("obj/matrix_mul.ptx")
matrixMul = mod.get_function("matrixMul")

A = np.ones((N, N), dtype=np.float32)
B = np.ones((N, N), dtype=np.float32)
C = np.empty((N, N), dtype=np.float32)

d_A = gpuarray.to_gpu(A)
d_B = gpuarray.to_gpu(B)
d_C = gpuarray.empty_like(d_A)

grid = ((N + BLOCK_SIZE - 1) // BLOCK_SIZE,
       (N + BLOCK_SIZE - 1) // BLOCK_SIZE)

start = drv.Event()
end = drv.Event()
start.record()

matrixMul(d_A.gpudata, d_B.gpudata, d_C.gpudata, np.int32(N),
         block=(BLOCK_SIZE, BLOCK_SIZE, 1),
         grid=(grid[0], grid[1]))

end.record()
end.synchronize()

elapsed = start.time_till(end)
print(f"PyCUDA \t\t\t\tTime: {elapsed:.3f} ms")
```

Листинг 8 – pycuda_impl.py

Результаты и анализ

```
CUDA API                    Time: 0.636 ms
CUDA Driver API (C/C++)     Time: 0.467 ms
CUDA Driver API (Python)    Time: 0.469 ms
Numba                       Time: 5.317 ms
PyCUDA                      Time: 0.730 ms
```

Листинг 9 – вывод скрипта run.sh

Лучший результат (0.467 мс) показала реализация CUDA Driver API (C/C++) благодаря использованию shared memory и минимальным накладным расходам за счет ручного управления контекстом.

CUDA Driver API (Python) близок по производительности (0.469 мс) к CUDA Driver API (C/C++), однако требует аккуратной работы с типами данных через ctypes.

CUDA API показал более медленное время выполнения (0.636 мс) из-за отсутствия оптимизации (глобальная память).

PyCUDA (0.730 мс) оказался медленнее предыдущих реализаций. Высокоуровневые механизмы PyCUDA упрощают код, но накладные расходы увеличивают время.

Numba показала самое большое время (5.317 мс), превышающее время PyCUDA более чем в 7, однако использование Numba обеспечивает простоту разработки.

Выводы

Самой быстрой реализацией оказалась CUDA Driver API (C/C++). Остальные реализации, кроме Numba, показали приемлемую скорость.

Numba проще в использовании, но менее эффективна для задач с частым обменом данными. PyCUDA требует ручной работы с памятью, но позволяет достичь скорости, близкой к C++.