

# Задание:

1. Реализовать произведение двух матриц, используя:

* CUDA API
* CUDA Driver API (C/C++)
* CUDA Driver API (Python)
* Numba
* PyCuda

1. 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++.