**Домашнее задание №2**

**Липатов Данила МСМТ243**

(пункты 1,2,4,5)

После подключения к HPC был перенесен исходный скрипт (скрипт писался в VSC .cpp, но компилировался и запускался с .cu, локально так же компилировался и запускался с .cu)

Был подключен модуль для CUDA - module load nvidia\_sdk/nvhpc/23.5

Далее файл компилировался :

nvcc hw\_1.cu -o hw\_1.out

И запускался на узле А следующим образом

srun -n 1 --gpus=1 --constraint="type\_a" hw\_1.out

Реализация для каждого из методов лежит в соответствующем файле.

Аналогично запуску выше были запушены и другие реализации с pinned, unified, shared, cuda-stream.

Разница между global, pinned, unified memory следующая:

Global:

double \*h\_A = (double\*)malloc(bytes);

double \*h\_B = (double\*)malloc(bytes);

double \*h\_C = (double\*)malloc(bytes);

double \*d\_A, \*d\_B, \*d\_C;

cudaMalloc(&d\_A, bytes);

cudaMalloc(&d\_B, bytes);

cudaMalloc(&d\_C, bytes);

cudaMemcpy(d\_A, h\_A, bytes, cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_B, bytes, cudaMemcpyHostToDevice);

Pinned:

double \*h\_A, \*h\_B, \*h\_C;

cudaMallocHost((void \*\*)&h\_A, size);

cudaMallocHost((void \*\*)&h\_B, size);

cudaMallocHost((void \*\*)&h\_C, size);

double \*d\_A, \*d\_B, \*d\_C;

cudaMalloc((void \*\*)&d\_A, size);

cudaMalloc((void \*\*)&d\_B, size);

cudaMalloc((void \*\*)&d\_C, size);

cudaMemcpy(d\_A, h\_A, size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_B, size, cudaMemcpyHostToDevice);

Unified:

double \*h\_A, \*h\_B, \*h\_C;

cudaMallocManaged(&h\_A, size);

cudaMallocManaged(&h\_B, size);

cudaMallocManaged(&h\_C, size);

По времени вычисления они одинаковые (+-).

Для shared memory используется pinned memory + переписанная функция умножения матриц как раз для разделения памяти, что позволяет значительно ускорить процесс умножения и получить следующий график:

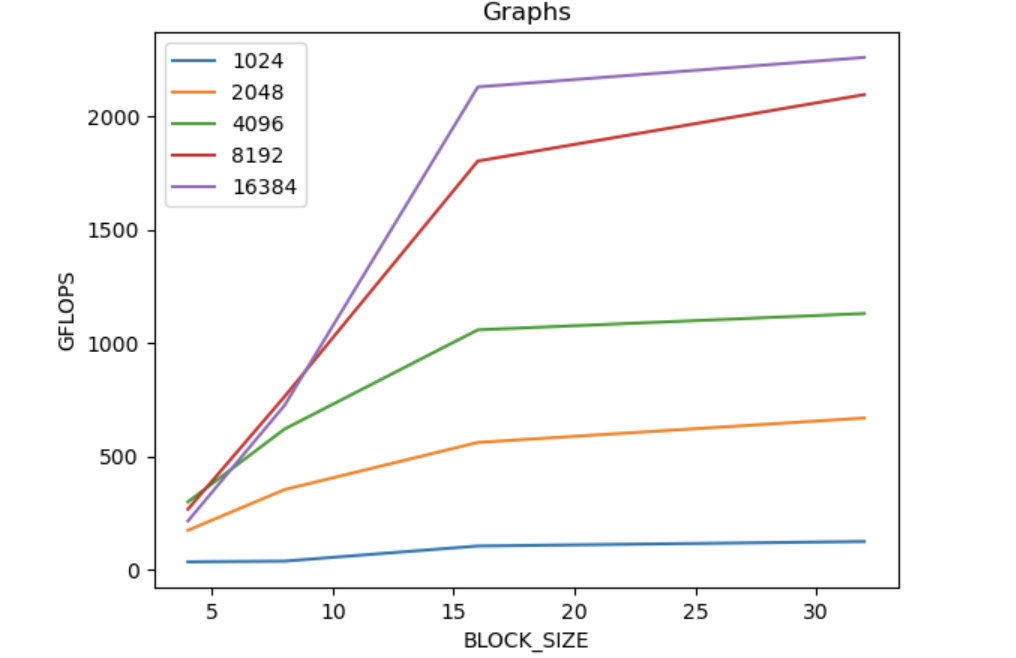


Рис. 1.1 Зависимость GFLOPS от BLOCK\_SIZE

Для CUDA stream так же был произведен анализ, для выяснения оптимального количества потоков (1/2/4) на графике ниже, и произведено сравнение с cuBLAS, CPU вычислениями

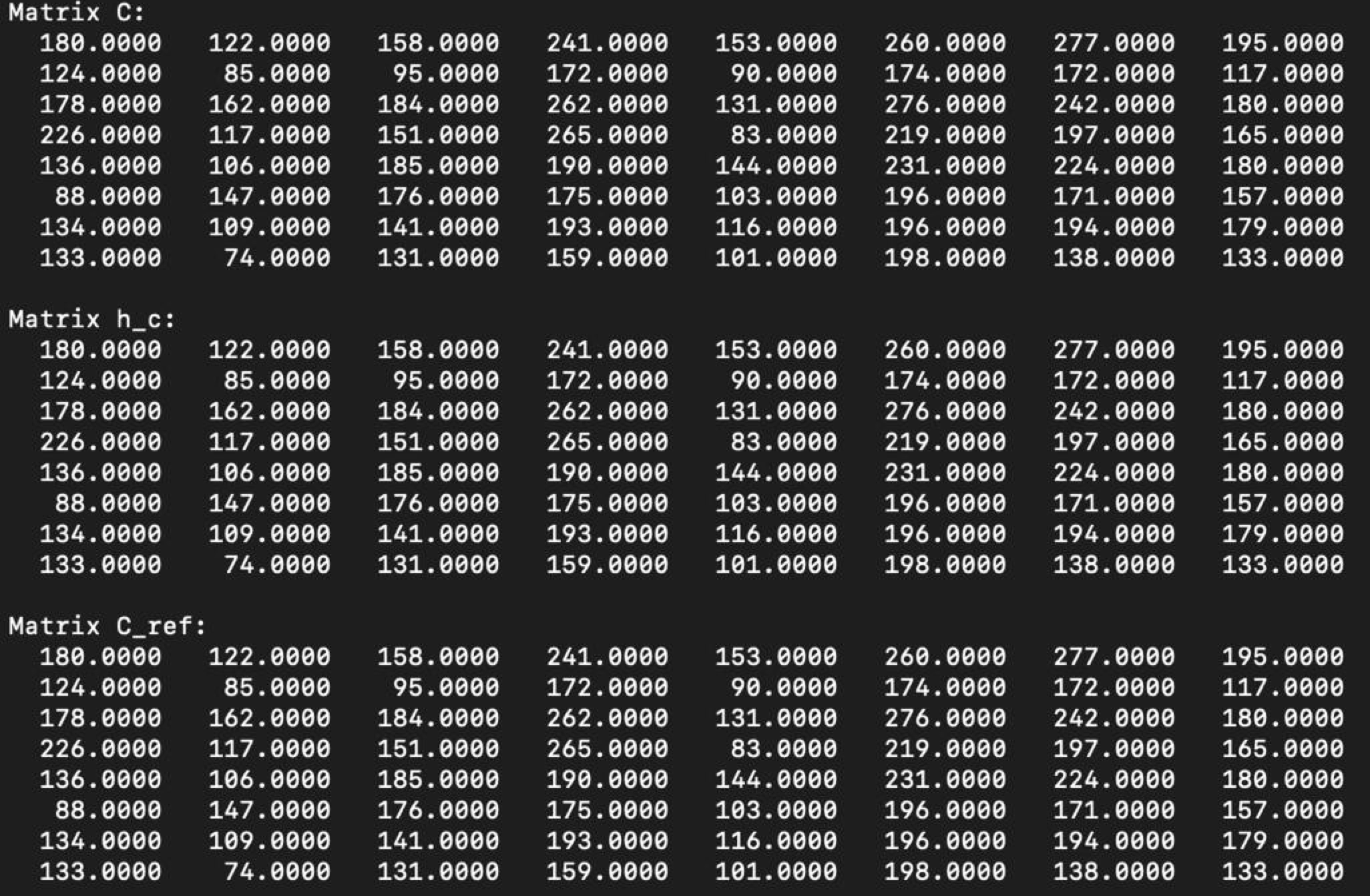


Рис. 1.2 Сравнение cuBLAS, cuda-stream, CPU

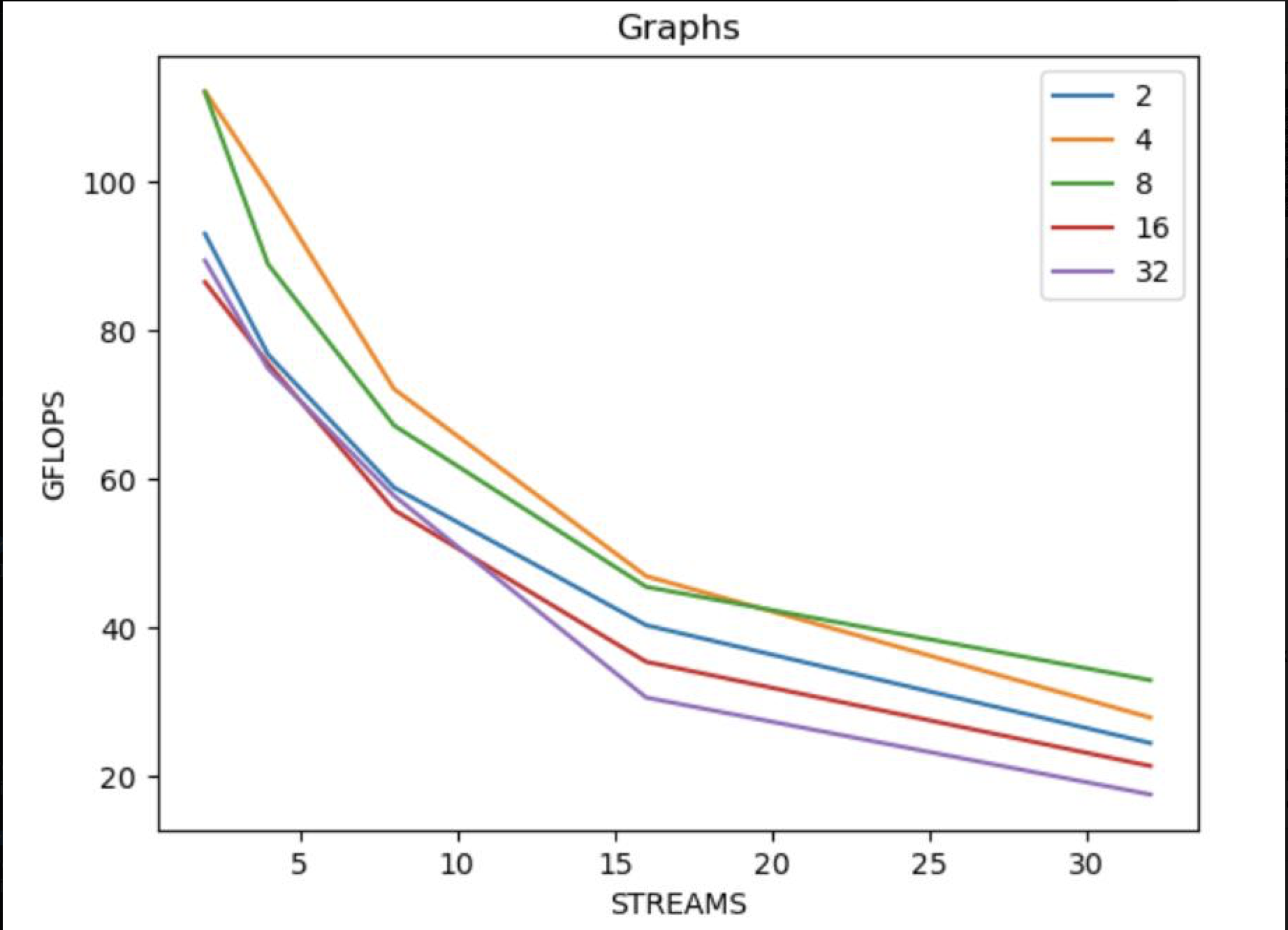


Рис. 1.3 матрица 1024

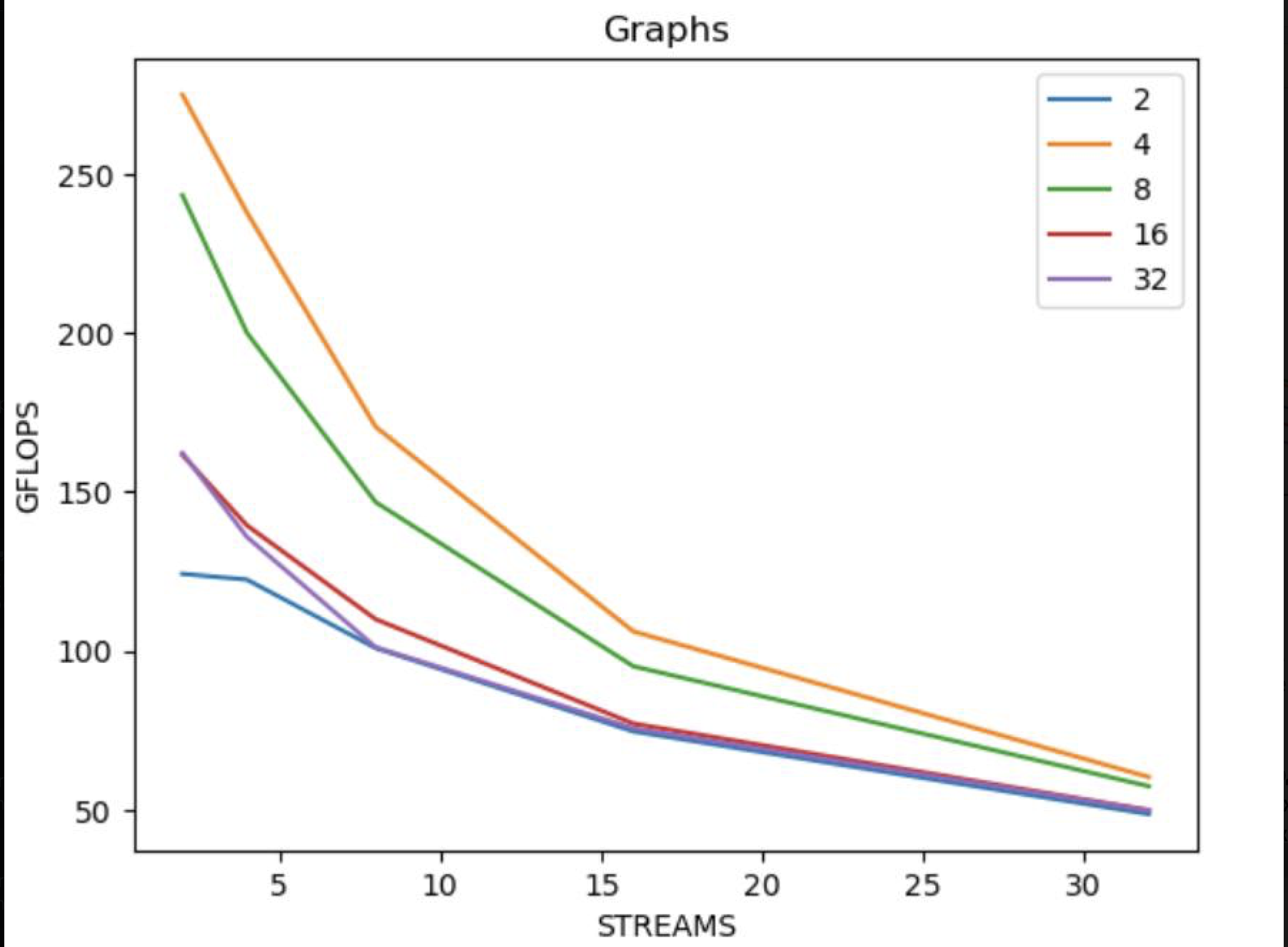


Рис. 1.4 матрица 2048

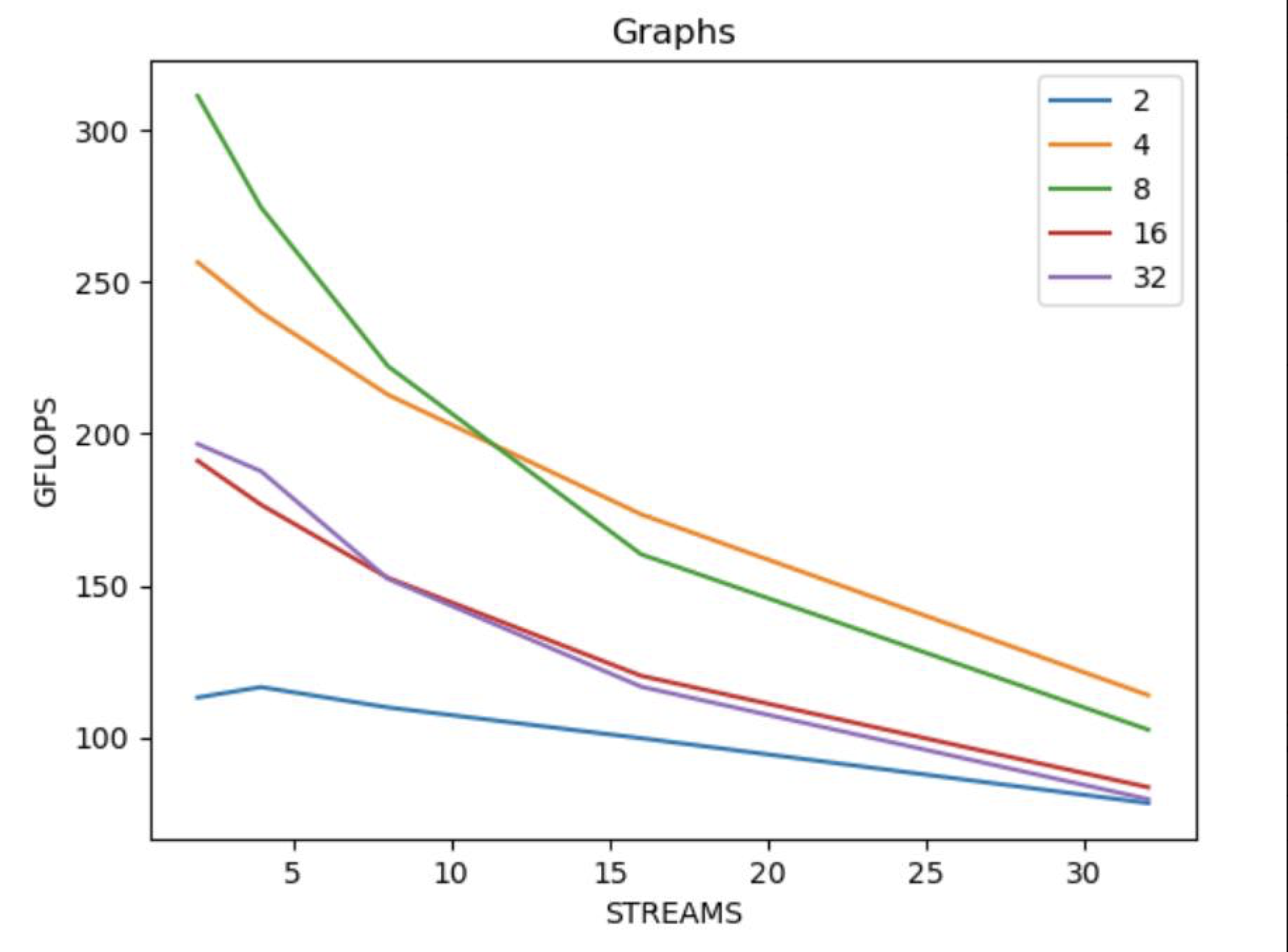


Рис. 1.5 матрица 4096

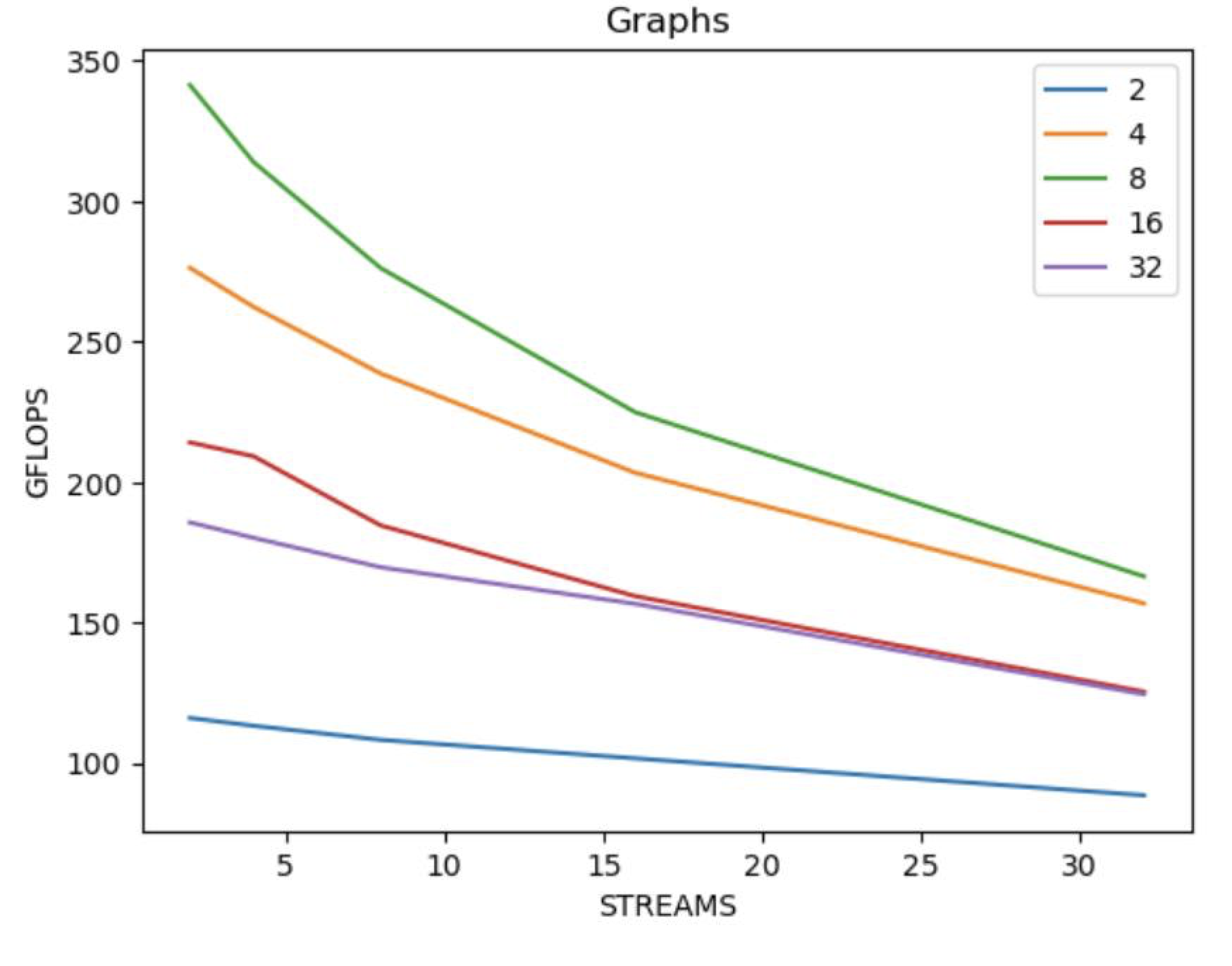


Рис. 1.6 матрица 8192

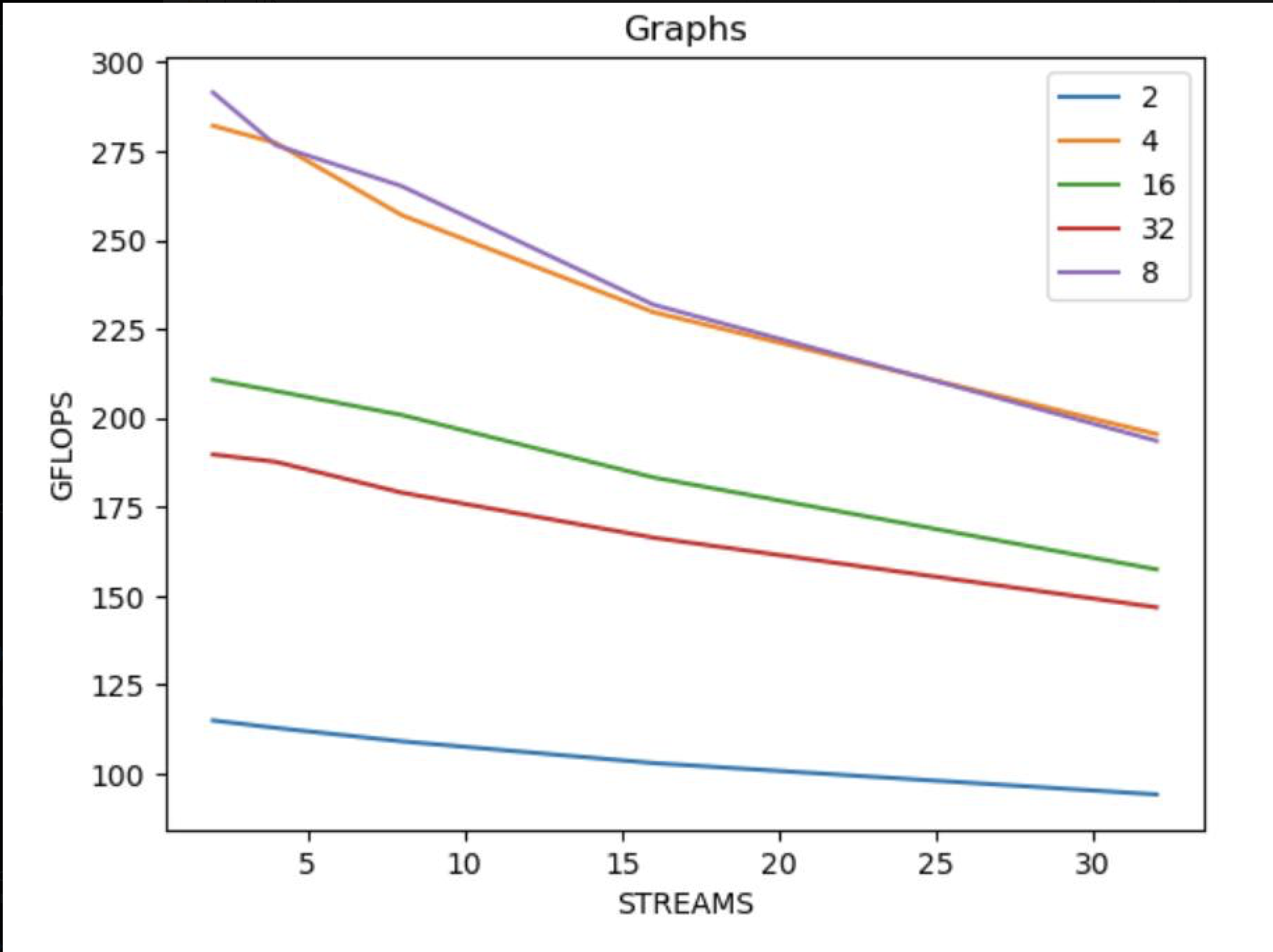


Рис. 1.7 матрица 16384

Потоки были кратны 2, если выбирается некратное 2-ум кол-во потоков, то оно округлялось сверху до кратного (так же чтобы был кратен кол-ву блоков), иначе матрица «обрезалась»

График для cuBLAS приведен ниже, его пиковая производительность в 8к GFLOPS соответствует максимальным мощностям ресурсов, так же было выяснено, что максимальная матрица равна (примерно) 35000, так как после 32768 возможно было еще добавить размеры, хотя матрица больше 40000 уже не проходила.

nvcc hw\_1\_cublas.cu -lcublas -o hw\_1\_cublas.out

График производительности:

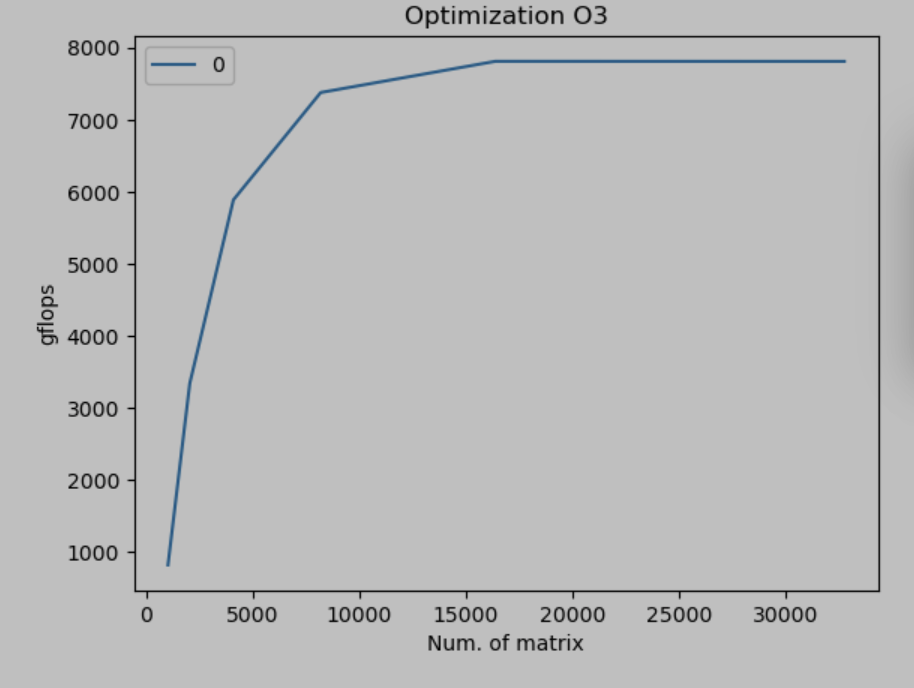


Рис. 1.8 GFLOPS от матрицы

Эмпирически было получено, что пиковая производительность приходится на большие матрицы, размер может превышать 32768 х 32768, но уже при размерах 65530 х 65530 не получается просто так начать расчет.

Для реализации openMP на GPU использовался скрипт из предыдущей лабораторной с pragma:

#pragma omp target data map(to: A[0:N\*N], B[0:N\*N]) map(from: C[0:N\*N])

#pragma omp target teams num\_teams(NUM\_TEAMS) distribute parallel for collapse(2)

И просто рассматривался вариант с

#pragma omp target teams distribute parallel for collapse(2)

Результат расчета сходится с предыдущими:

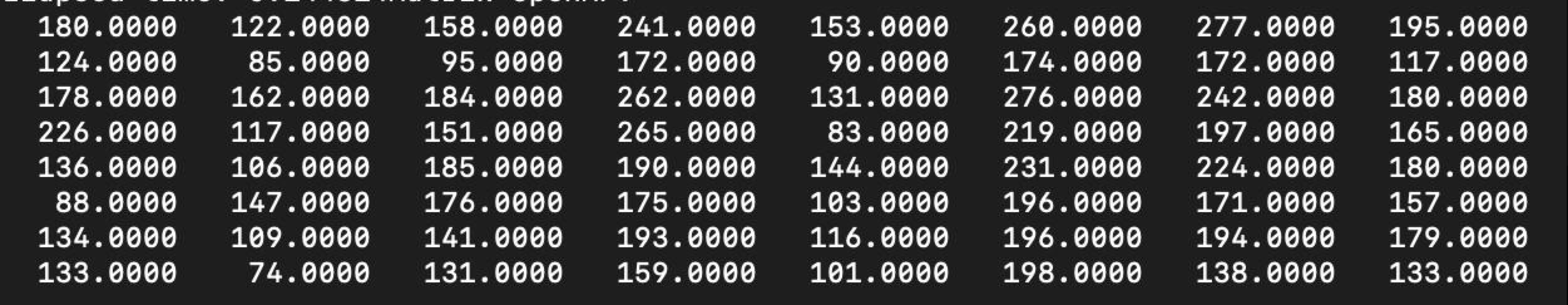


Рис. 1.9 Расчет матрицы

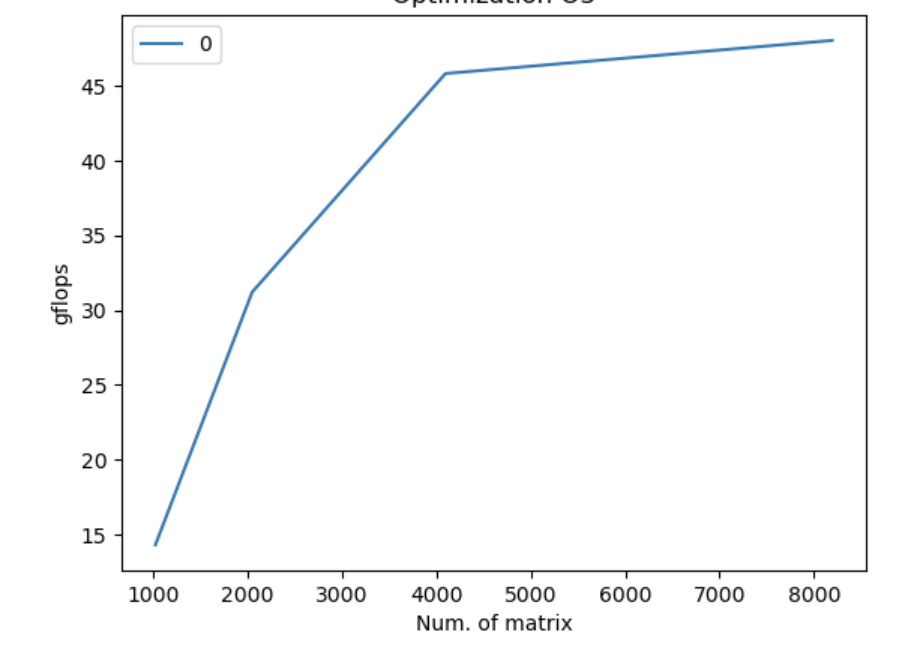


Рис. 1.10 GFLOPS от матрицы

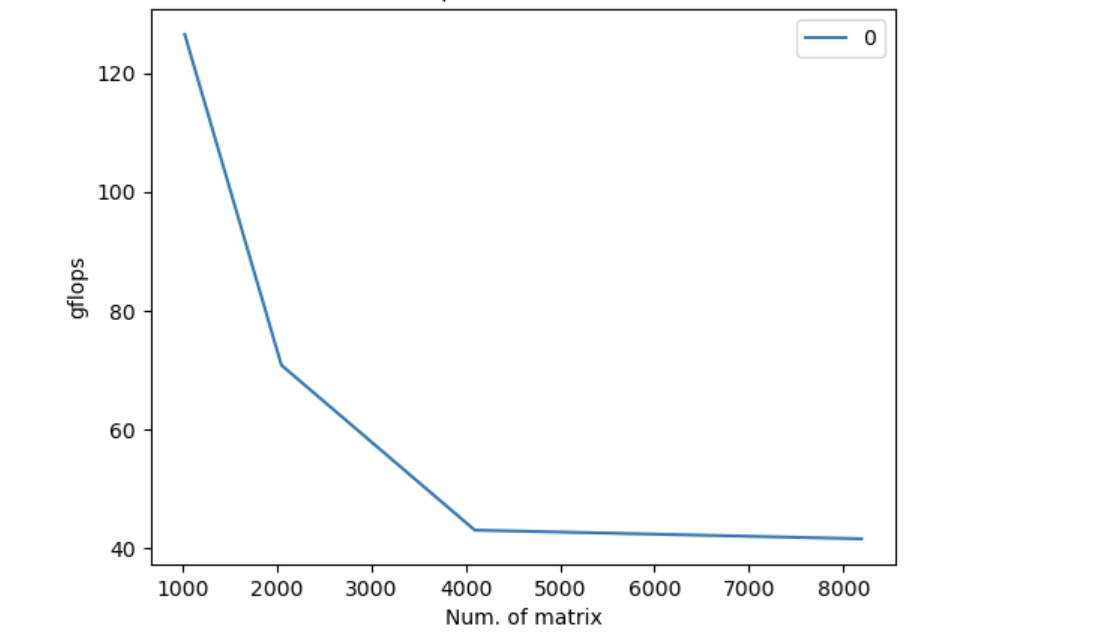


Рис. 1.11 GFLOPS от матрицы