Московский государственный университет имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра вычислительных методов

Построение разреженной матрицы и решение СЛАУ

Параллельные высокопроизводительные вычисления

выполнил:
Петров Т. П.
группа 504

26 апреля
Москва, 2025

Содержание

1 Описание задания и программной реализации

1.1 Краткое описание задания

Необходимо реализовать многопоточную программу для решения систем линейных алгебраически уравнений (СЛАУ) на неструктурированной сетке с использованием OpenMP. Алгоритм должен состоять из нескольких этапов:

1. Генерация графа сетки и его матричного представления – создание графа, связей элементов и его представление в разреженном формате CSR
2. Заполнение матрицы СЛАУ – построение матрицы коэффициентов и вектора правой части с использованием тестовых формул
3. Решение СЛАУ – реализация итерационного метода сопряженных градиентов для решения уравнения с поддержкой параллелизма
4. Проверка производительности – измерение времени выполнения каждого этапа и анализ многопоточного ускорения и эффективности алгоритма

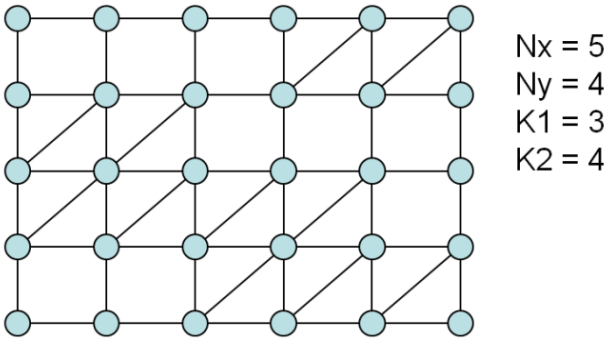


Рис. 1: Пример сетки для генерации графа

1.2 Характеристики вычислительной системы и сборка

	PC	Polus node
CPU	i5-12400F	IBM POWER 8
Cores	6	10
Threads	2	8
TPP	384 GFLOPS	290 GFLOPS
RAM	2xDDR5-5600	4xDDR4-2400
BW	89.6 GB/s	153.6 GB/s

Для того, чтобы собрать программу и скомпилировать все файлы, необходимо выполнить ряд следующих действий:

```
mkdir build && cd build
cmake -DENABLE_TESTS=<On|Off> -DDEBUG_MODE=<On|Off> -DUSE_MPI=<On|Off> ..
make -j $(nproc)
cd ../bin
```

Также на локальных системах можно запустить несколько простых тестов для проверки корректности работы OpenMP программы (при условии включения флага ENABLE_TESTS):

```
cd build/Tests
ctest
```

1.3 Запуск программы

Для запуска на локальных системах достаточно указать количество нитей, а также все необходимые входные данные:

```
OMP_NUM_THREADS=k ./CGSolver --Nx VAL --Ny VAL --K1 VAL --K2 VAL
```

Для запуска на кластере, использующем систему очередей, запускается скрипт со следующими параметрами:

```
mpisubmit.pl -t k CGSolver -- Nx Ny K1 K2
```

Однако желательно редактирование командного файла для запуска на заданных узлах и привязки нитей к ядрам.

```
#BSUB -o out/CGSolver.%J.out
#BSUB -e out/CGSolver.%J.err
#BSUB -m polus-c4-ib
#BSUB -R affinity[core(20)]
OMP_NUM_THREADS=k
/polusfs/lsf/openmp/launchOpenMP.py bin/CGSolver Nx Ny K1 K2
```

Для запуска MPI версии программы локально (обязательно следует указывать число нитей на один MPI процесс OMP_NUM_THREADS=M, иначе по умолчанию будет общее число доступных нитей, что существенно замедлит выполнение):

```
OMP_NUM_THREADS=M mpiexec -n N bin_mpi/CGSolver --Nx VAL --Ny VAL --K1 VAL --K2
  VAL --Px VAL --Py VAL
```

и на кластере:

```
source /polusfs/setenv/setup.SMPI
#BSUB -n N
#BSUB -x
#BSUB -o out/MPI/CGSolverMeasure.%J.out
#BSUB -e out/MPI/CGSolverMeasure.%J.err
#BSUB -R "span[ptile=N/2] affinity[core(1)]"
#BSUB -m "polus-c2-ib polus-c4-ib"
export OMP_NUM_THREADS=M
mpiexec --bind-to core --map-by core bin_mpi/CGSolver Nx Ny K1 K2 Px Py
```

2 OpenMP реализация

2.1 Результаты измерений производительности

2.1.1 Последовательная производительность

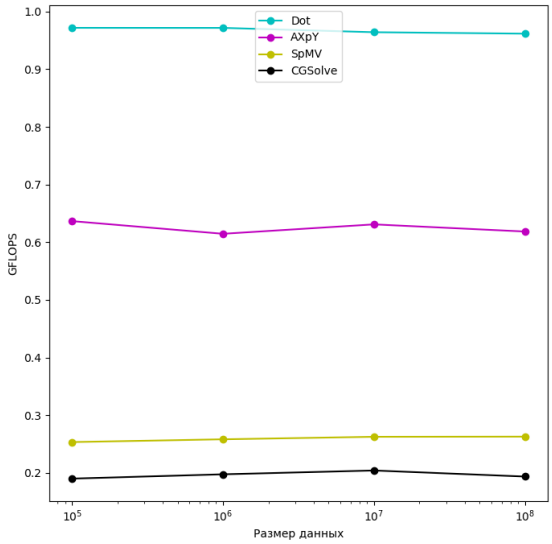
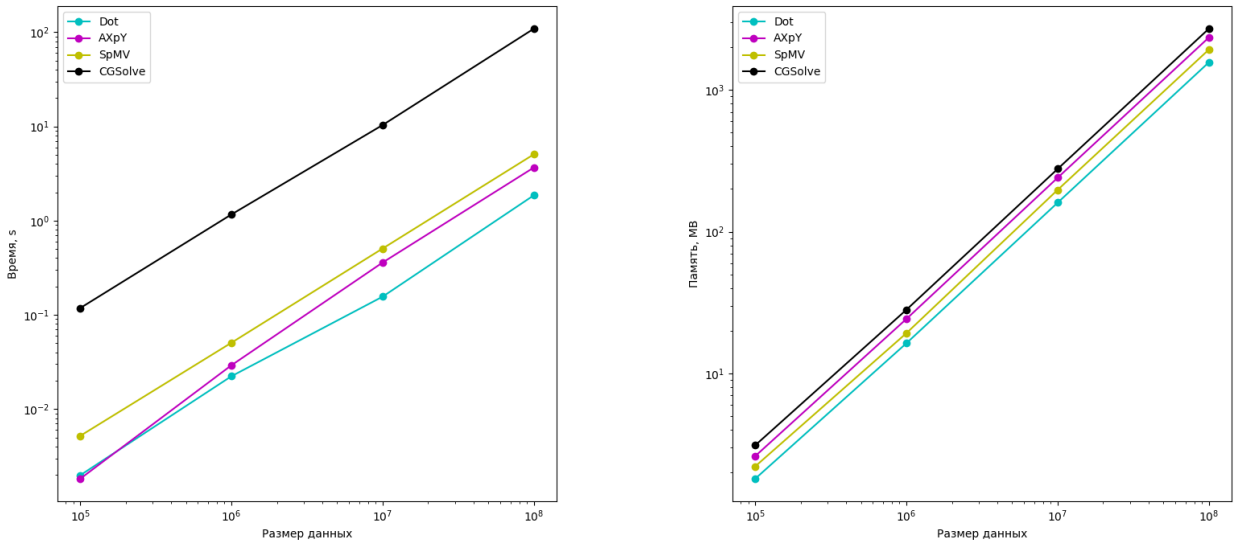


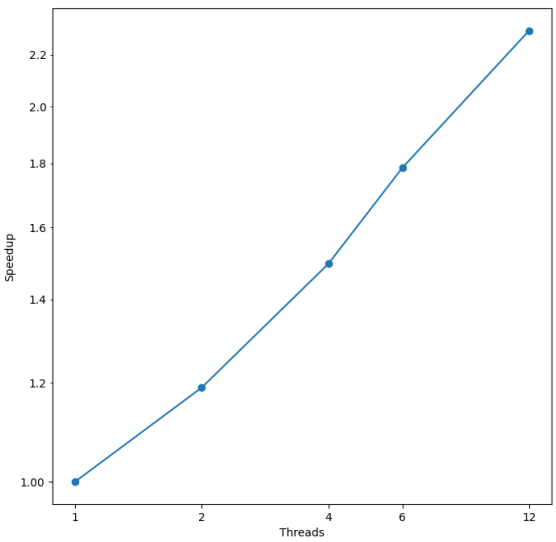
Рис. 2: Зависимость производительности от размера входных данных



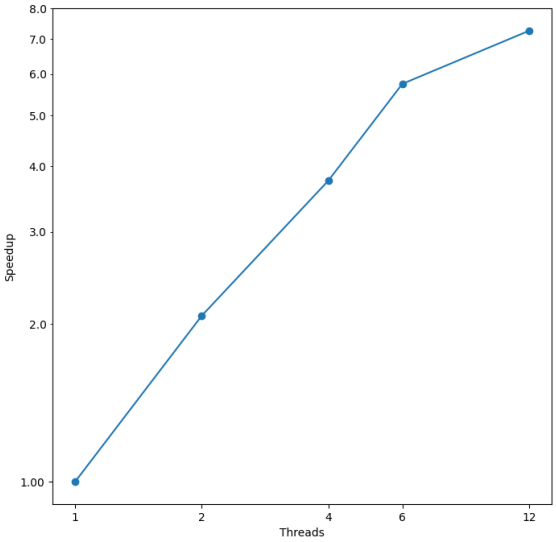
(а) (б)

Рис. 3: Зависимость времени выполнения (а) и выделяемой памяти (б) в зависимости от размера входных данных

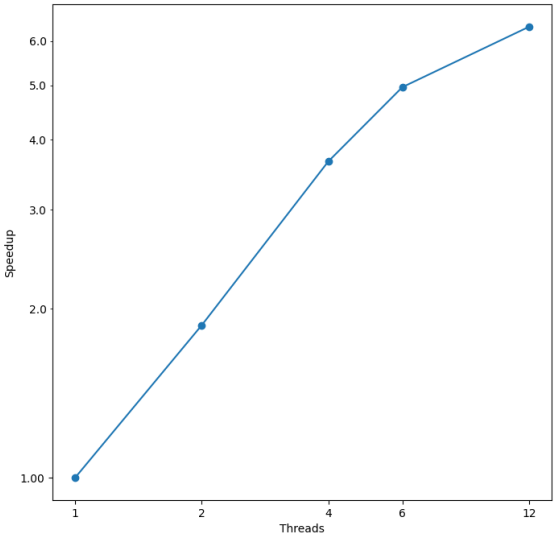
2.1.2 Параллельное ускорение



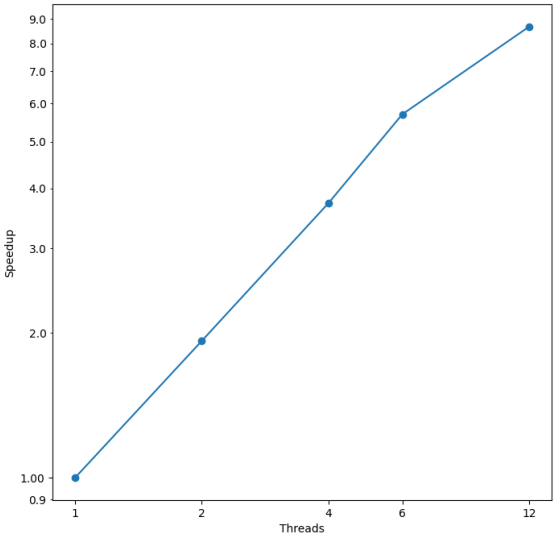
(а) EN



(б) NE

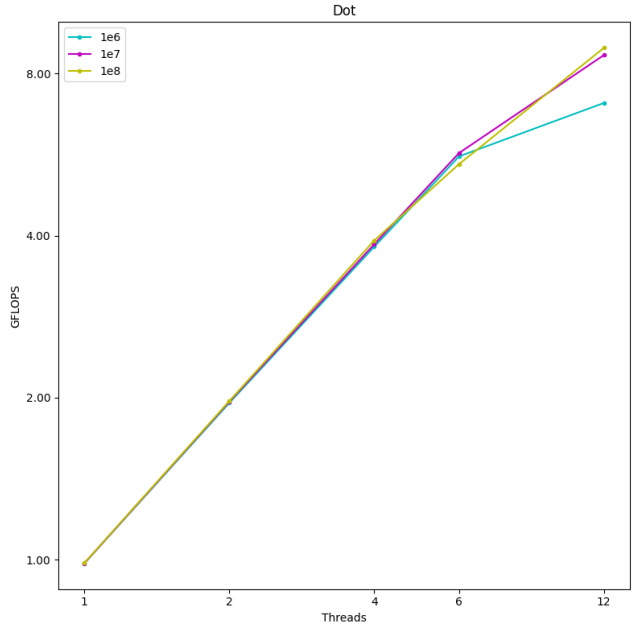


(в) EE

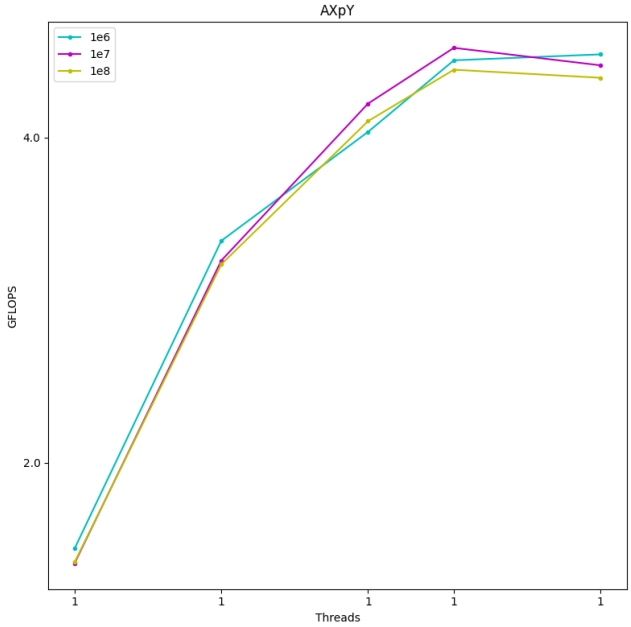


(г) Fill

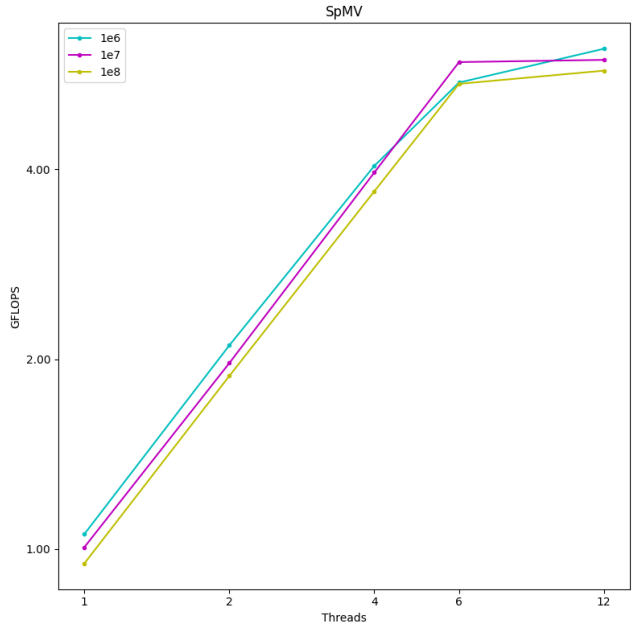
Рис. 4: Ускорение создания (а), транспонирования (б), построения смежности (в) и заполнения (г) в зависимости от числа нитей



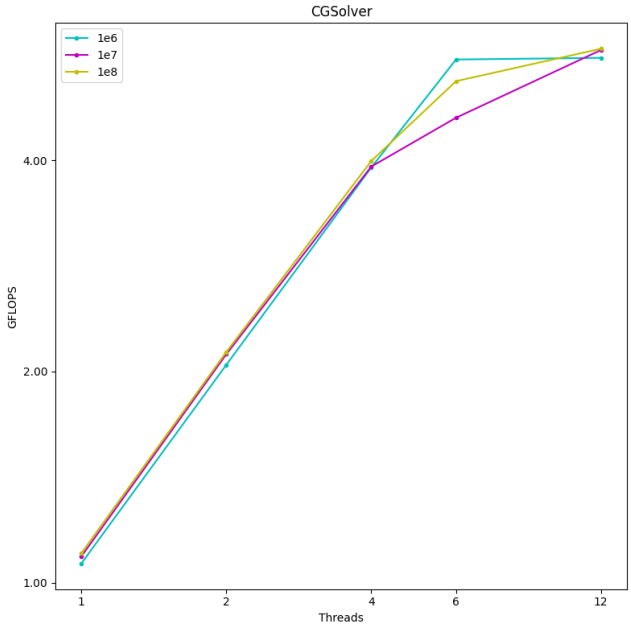
(а) Dot



(б) AXpY



(в) SpMV



(г) CGSolver

Рис. 5: Зависимость производительности от числа нитей

T	2	4	6	12
Dot	2.00	3.98	5.51	9.06
AXpY	1.89	2.56	2.86	2.81
SpMV	1.98	3.89	5.76	6.05
CGSolve	1.94	3.62	4.71	5.24

Рис. 6: Расчеты ускорения для каждой из операций на двухсокетной конфигурации при $N = 10^8$

2.2 Анализ полученных результатов

$TPP_{PC} = 4\ GHz \cdot 6\ Cores \cdot 2\ Threads/Core \cdot 512/64 = 384\ GFLOPS$
$TPP_{Polus} = 290\ GFLOPS$
$BW_{PC} = 2\ Channels \cdot 5600\ MT/s \cdot 8\ bytes = 89,6\ GB/s$
$BW_{Polus} = 8\ Channels \cdot 2400\ MT/s \cdot 8\ bytes = 153,6\ GB/s$

$AI_{dot} = 2\ FLOP/(2 \cdot 8)\ bytes = 1/8$

На каждой итерации считываем из памяти $x[i]$ и $y[i]$, каждый из которых типа 'double' занимает — 8 байт. При этом на каждой итерации происходит 2 операции: умножение $x[i] \cdot y[i]$ и сложение с итоговой суммой

$AI_{AXpY} = 2\ FLOP/(3 \cdot 8)\ bytes = 1/12$

На каждой итерации считываем из памяти $x[i]$ и $y[i]$, каждый из которых типа 'double' занимает — 8 байт, а также записываем в память результат $z[i] = a \cdot x[i] + y[i]$ типа 'double' — тоже 8 байт. Следовательно, всего $3 \cdot 8$ байтов. При этом на каждой итерации происходит 2 операции: умножение $a \cdot x[i]$ и сложение $x[i] + y[i]$

$AI_{SpMV} = 12\ FLOP/(132)\ bytes = 1/11$

За итерацию будем считать умножение строки i матрицы A на столбец x. Для считывания элементов матрицы A необходимо вначале получить номер $col = ia[i]$, с которого в массиве a начинают храниться элементы i-ой строки — 4 байта (ia имеет тип 'int'). Далее будем считать, что в среднем в строке 6 ненулевых элементов. Для каждого такого элемента необходимо считать значение $a[col]$ типа 'double' — 8 байт, считать $j = ja[col]$ номер столбца — 4 байта (ja имеет тип 'int'), а также считать $x[j]$ типа 'double' — 8 байт. В конце вычислений необходимо записать полученный результат в $y[i]$ типа 'double' — 8 байт. Таким образом получаем, что на каждую строку нам понадобится $4+6\cdot(4+8+8)+8 = 132$. При этом для каждого элемента строки потребуется 2 операции (умножение $a[col] \cdot x[j]$ и сложение в общий результат). Следовательно всего $6 \cdot 2 = 12$ операций.

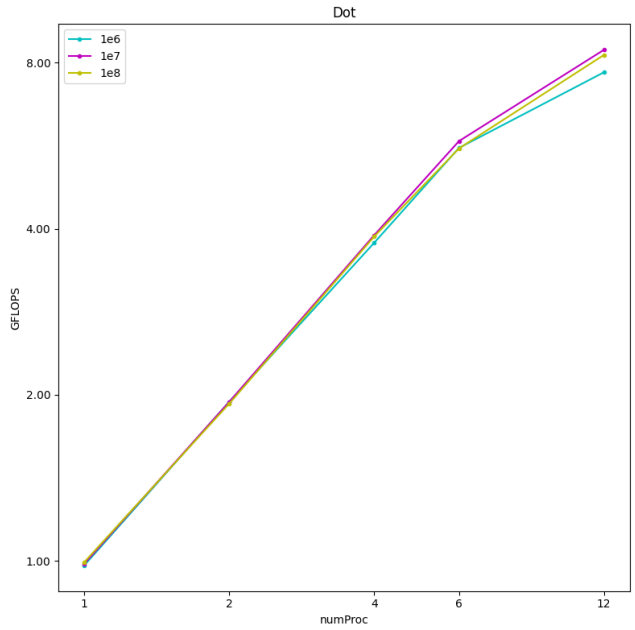
$TBP = \min(TPP, BW \cdot AI)$
$TBP_{PC,dot} = \min(384\ GFLOPS, 89,6\ GB/s \cdot 1/8) = 11.2\ GFLOPS$
$TBP_{PC,AXpY} = \min(384\ GFLOPS, 89,6\ GB/s \cdot 1/12) = 7.5\ GFLOPS$
$TBP_{PC,SpMV} = \min(384\ GFLOPS, 89,6\ GB/s \cdot 1/11) = 8.1\ GFLOPS$

...	Dot	AXpY	SpMV
$PC\ TBP_{Analytical}$	11.2 $GFLOPS$	7.5 $GFLOPS$	8.1 $GFLOPS$
$PC\ RealP$	8.93 $GFLOPS$	4.77 $GFLOPS$	6.21 $GFLOPS$

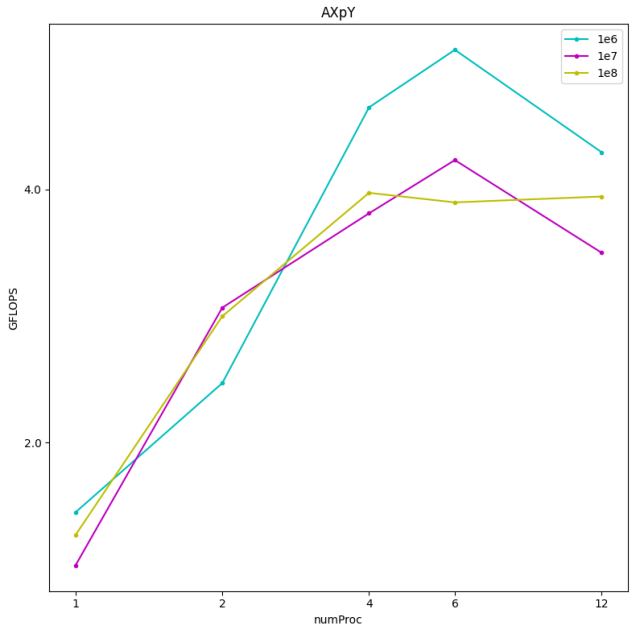
Рис. 7: Аналитические и реальные значение производительности для каждой из операций

3 MPI реализация

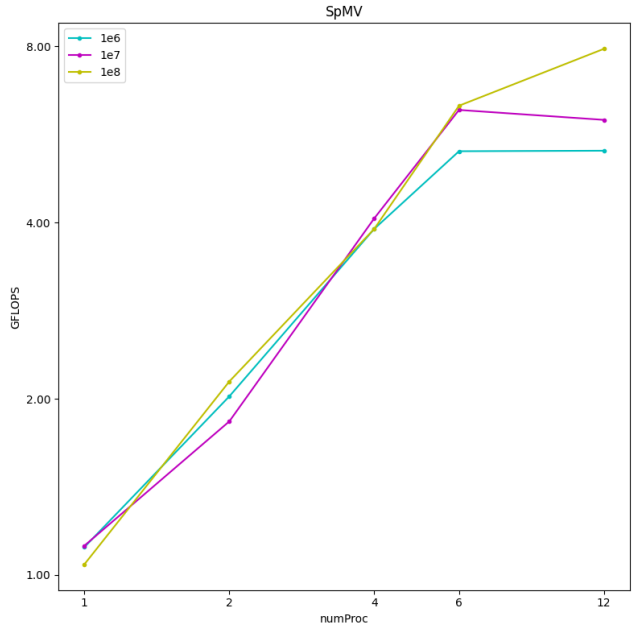
3.1 Параллельное ускорение



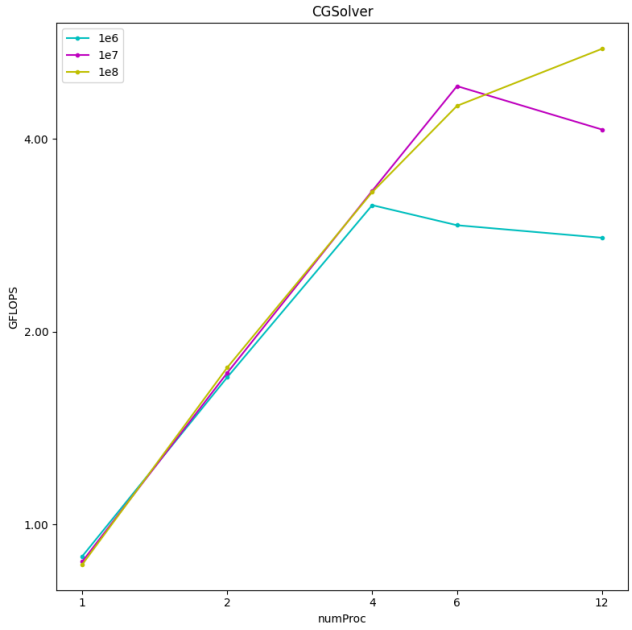
(а) Dot



(б) AXpY



(в) SpMV



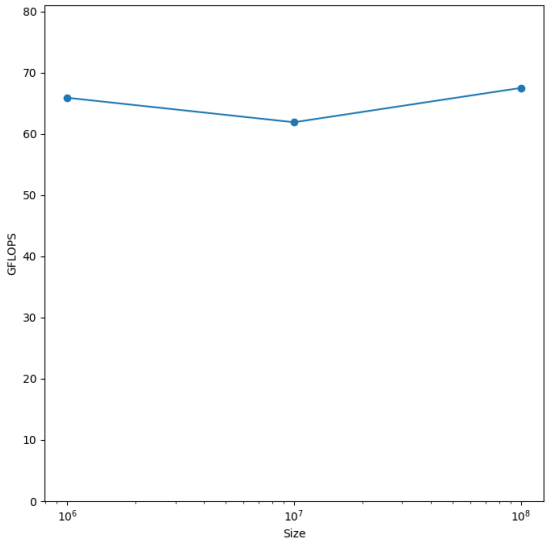
(г) CGSolver

Рис. 8: Зависимость производительности от числа процессов

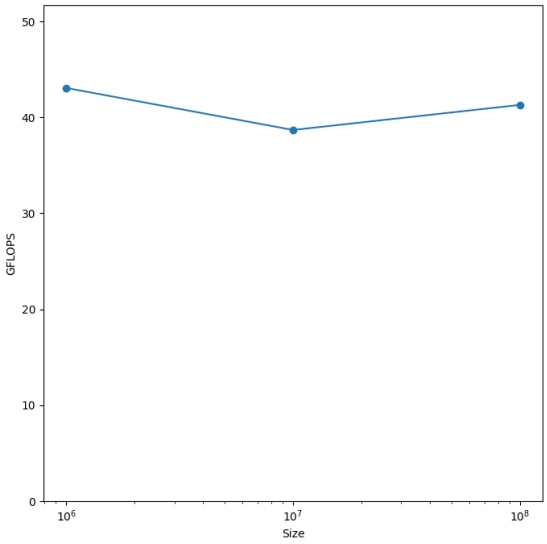
T	2	4	6	12
Dot	1.94	3.89	5.62	8.31
AXpY	1.82	2.55	2.82	2.76
SpMV	2.05	3.74	6.07	7.60
CGSolve	2.03	3.81	5.20	6.38

Рис. 9: Расчеты ускорения в MPI режиме при $N = 10^8$

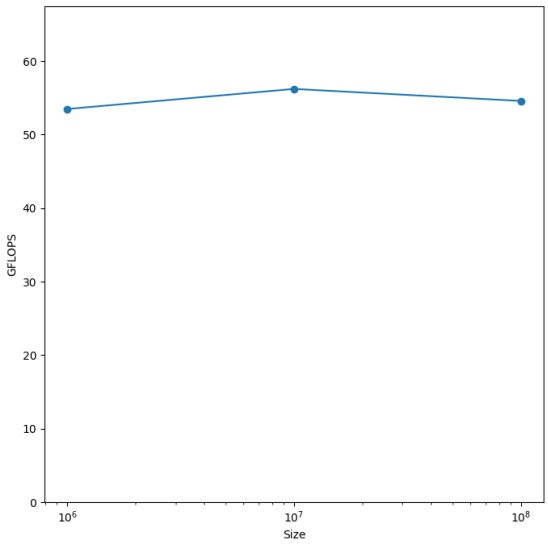
4 CUDA реализация



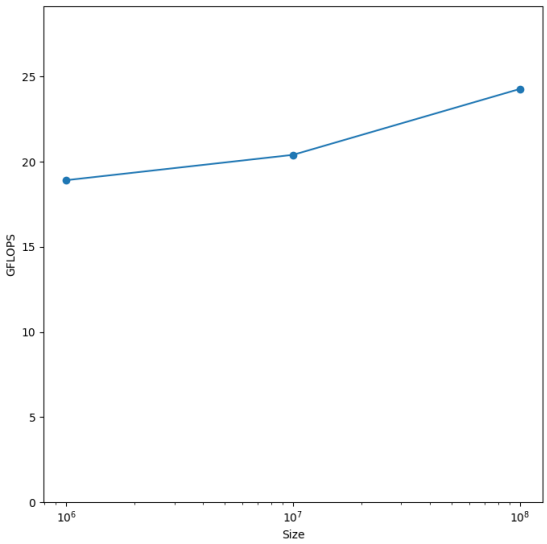
(а) Dot



(б) AXpY



(в) SpMV



(г) CGSolver

Рис. 10: Зависимость производительности от размера матрицы