Технология CUDA для высокопроизводительных вычислений на кластерах с GPU

Лихогруд Николай
n.lihogrud@gmail.com

Задание

Однокубитовая квантовая операция

Однокубитовая квантовая операция

- Операция применения квантового вентиля к одному из кубитов квантового регистра
  - Квантовый вентиль задается унитарной матрицей размера 2×2
  - Квантовый регистр из N кубитов задается квантовой суперпозицией его состояний
- На выходе операции – новая суперпозиция состояний регистра

Унитарная матрица

- Квадратная матрица с *комплексными элементами*, результат умножения которой на эрмитово сопряжённую равен единичной матрице

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} \bar{a} & \bar{c} \\ \bar{b} & \bar{d} \end{bmatrix} = \begin{bmatrix} |a|^2 + |b|^2 & a\bar{c} + b\bar{d} \\ c\bar{a} + d\bar{b} & |c|^2 + |d|^2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Квантовая суперпозиция

- Обозначим состояние квантового регистра, при котором кубиты находятся в состояниях $|b_i>,\ i = 0..N-1$, как булевый вектор $\bar{B}_s$:

$$\bar{B}_s = b_0 b_1 \dots b_{N-1} : \sum_{i=0}^{N-1} b_i 2^i = s\,, b_i = 0 \text{ или } 1$$

- Квантовая суперпозиция состояний квантового регистра из N кубит задается 2^N комплексными числами, такими, что

$$\sum_{s=0}^{2^N-1} |p_{\bar{B}_s}|^2 = 1, p_{\bar{B}_s} \text{ – амплитуда состояния } \bar{B}_s$$

  - $|p_{\bar{B}_s}|^2$ - вероятность при измерении получить состояние $\bar{B}_s$

Квантовая операция

- Вход:

$M = \{m_{ij}\} \in C^{2\times 2}$ - квантовый вентиль (унитарная матрица)

$\bar{P} = \{p_{\bar{B}_0}, p_{\bar{B}_1}, \dots, p_{\bar{B}_{2^N-1}}\}$ – суперпозиция состояний регистра из *N* кубитов

i – индекс кубита, к которому нужно применить вентиль

- Выход

$\bar{P'} = \{p'_{\bar{B}_0}, p'_{\bar{B}_1}, \dots, p'_{\bar{B}_{2^N-1}}\}$ – новая суперпозиция состояний:

$$p'_{b_0 b_1 \dots b_i \dots b_{N-1}} = m_{b_i 0} * p_{b_0 b_1 \dots 0_i \dots b_{N-1}} + m_{b_i 1} * p_{b_0 b_1 \dots 1_i \dots b_{N-1}}$$

Пример

- Регистр из одного кубита:

$$\begin{bmatrix} p'_0 \\ p'_1 \end{bmatrix} = \begin{bmatrix} m_{00} * p_0 + m_{01} * p_1 \\ m_{10} * p_0 + m_{11} * p_1 \end{bmatrix} = M \times P^T$$

Пример

- Регистр из двух кубитов, $i = 1$:

$$\begin{bmatrix} p'_{00} \\ p'_{01} \\ p'_{10} \\ p'_{11} \end{bmatrix} = \begin{bmatrix} m_{00} * p_{00} + m_{01} * p_{01} \\ m_{10} * p_{00} + m_{11} * p_{01} \\ m_{00} * p_{10} + m_{01} * p_{11} \\ m_{10} * p_{10} + m_{11} * p_{11} \end{bmatrix}$$

Пример

- Регистр из двух кубитов, $i = 0$:

$$\begin{bmatrix} p'_{00} \\ p'_{01} \\ p'_{10} \\ p'_{11} \end{bmatrix} = \begin{bmatrix} m_{00} * p_{00} + m_{01} * p_{10} \\ m_{00} * p_{01} + m_{01} * p_{11} \\ m_{10} * p_{00} + m_{11} * p_{10} \\ m_{10} * p_{01} + m_{11} * p_{11} \end{bmatrix}$$

Пример

- Регистр из трех кубитов, $i = 1$:

$$\begin{bmatrix} q_{000} \\ q_{001} \\ q_{010} \\ q_{011} \\ q_{100} \\ q_{101} \\ q_{110} \\ q_{111} \end{bmatrix} = \begin{bmatrix} m_{00} * p_{000} + m_{01} * p_{010} \\ m_{00} * p_{001} + m_{01} * p_{011} \\ m_{10} * p_{000} + m_{11} * p_{010} \\ m_{10} * p_{001} + m_{11} * p_{011} \\ m_{00} * p_{100} + m_{01} * p_{110} \\ m_{00} * p_{101} + m_{01} * p_{111} \\ m_{10} * p_{100} + m_{11} * p_{110} \\ m_{10} * p_{101} + m_{11} * p_{111} \end{bmatrix}$$

Одиночная операция

Суть задания

- Предоставляется C++ реализация:
  - Парсинга параметров
  - Генерации, проверки корректности, вывода суперпозиции состояний квантового регистра и квантового вентиля (унитарной матрицы)
  - Вспомогательных типов для упрощения CUDA-ядер
- Требуется реализовать однокубитовую квантовую операцию операций на CPU и GPU

Описание программы

- Параметры программы:

`-n <number of qubits>`	- число кубитов в регистре
`-i <qubit index>`	- номер кубита, над которым требуется провести операцию
`-check`	- флаг проверки результата
`-v`	- флаг вывода в консоль регистров и квантового вентиля

- Вывод программы:
  - Время работы на GPU
  - Время работы на CPU, если включен флаг проверки результата

Используемые типы данных

- в `definitions.h`:

```
typedef double ComplexAmplitudePart;
```

- Тип мнимой/вещественной частей комплексной амплитуды.Вынесена в отдельный typedef для быстрого переключения между `float`/`double`

```
typedef std::complex<ComplexAmplitudePart> ComplexAmplitude;
```

- Комплексная амплитуда

```
typedef std::vector<ComplexAmplitude> QuantumSuperposition;
```

- Суперпозиция состояний квантового регистра

```
typedef ComplexAmplitude QuantumGate[2][2];
```

- Квантовый вентиль

Вспомогательные функции

- В `definitions.h` и `helpers.cpp`:

```
void intializeRandomQuantumRegisterStatesSuperposition(
    QuantumSuperposition &quantumRegisterStatesSuperposition);
```

- Инициализация квантового регистра. Сумма квадратов амплитуд равна единице.

```
void intializeRandomQuantumGate(QuantumGate &quantumGate);
```

- Инициализация квантового вентиля - унитарной комплексной матрицы

```
void printQuantumRegisterStatesSuperposition(const
    QuantumSuperposition &quantumRegisterStatesSuperposition);
```

- Вывести амплитуды состояний квантовой суперпозиции

Вспомогательные типы для GPU

- В `kernel.cu`:

Комплексное число с операторами, работающими на GPU

```
struct ComplexAmplitudeGPU {
     ComplexAmplitudePart real;
     ComplexAmplitudePart imag;

     __device__ ComplexAmplitudeGPU() {}
     __device__ ComplexAmplitudeGPU(ComplexAmplitudePart real,
                                    ComplexAmplitudePart imag)
          : real(real), imag(imag) {}

     __device__ ComplexAmplitudeGPU operator*(ComplexAmplitudeGPU other);
     __device__ ComplexAmplitudeGPU operator+(ComplexAmplitudeGPU other);
};
```

Вспомогательные типы для GPU

- В `kernel.cu`:

  Класс для раздельного хранения вещественных и мнимых частей, см. лекцию по глобальной памяти

```
struct QuantumSuperpositionGPU {
    int numberOfStates;
    ComplexAmplitudePart *real;
    ComplexAmplitudePart *imag;

    __device__ ComplexAmplitudeGPU operator[] (uint64_t index);
    __device__ void setAmplitudeAtIndex(uint64_t index,
                                        ComplexAmplitudeGPU amplitude);
};
```

Требуется реализовать

```
void applyQuantumGateToQuantumSuperpositionOnCPU(
    QuantumSuperposition &quantumSuperposition,
    QuantumGate quantumGate, int qubitIndex);
```

- Применить на **CPU** вентиль `quantumGate` к кубиту с индексом `qubitIndex` в квантовом регистре `quantumSuperposition`
- Использовать OpenMP, т.е. накинуть `#pragma omp parallel for` на внешний цикл

***Результат применения операции записывается в тот же вектор**

Требуется реализовать

```
void applyQuantumGateToQuantumSuperpositionOnGPU(
    QuantumSuperposition &quantumSuperposition,
    QuantumGate quantumGate, int qubitIndex);
```

- Применить на **GPU** вентиль `quantumGate` к кубиту с индексом `qubitIndex` в квантовом регистре `quantumSuperposition`
- Функция выделяет память на GPU, копирует на GPU квантовую суперпозицию и вентиль, запускает ядро, копирует результат с GPU, замеряет время выполнения копирований и счета ядра

***Результат применения операции записывается в тот же вектор**

Требования к CUDA-части

- Квантовый вентиль располагаем в константной памяти

```
__constant__ ComplexAmplitudeGPU quantumGateOnGPU[2][2];
```

- Каждая нить рассчитывает амплитуды двух связанных состояний. Общее число нитей 2^{N-1}
- Блоки грида линейные, по 512 нитей - `dim3(512)`
- Размеры грида рассчитывать в зависимости от параметра устройства `int cudaDeviceProp::maxGridSize[3]`
  - Если число блоков превышает максимальную ширину грида, то высота грида будет больше одного блока

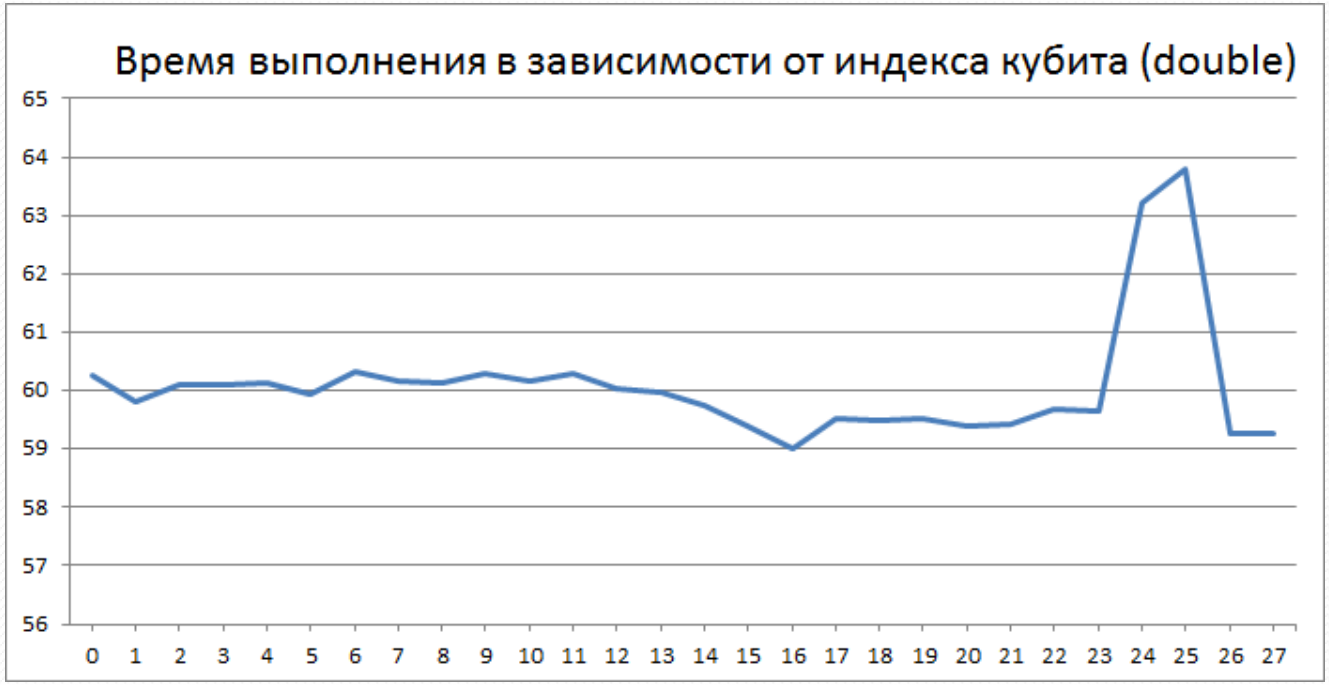
Переход от CPU к GPU

- На CPU – цикл из 2^{N-1} итераций, на каждой итерации рассчитываются амплитуды двух связанных состояний регистра

- Для перехода к GPU-реализации нужно распределить итерации по нитям
  - Ядро будет состоять из вычисления индекса итерации, которую должна выполнить нить, и тела цикла из CPU-реализации

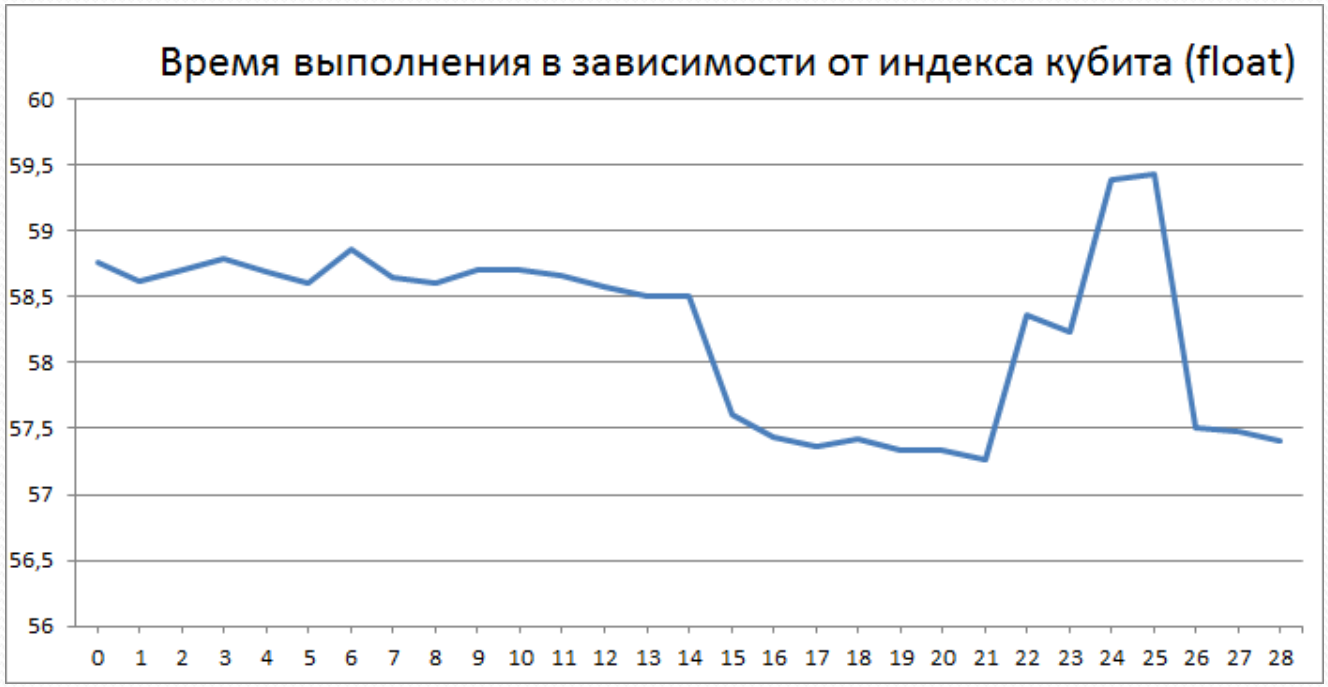
Тесты

- Продемонстрировать результаты тестирования в виде графика (для `double`):



Тесты

- Продемонстрировать результаты тестирования в виде графика (для `float`):



Последовательность операций + общая память

0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	0	1	1
0	0	0	1	0	0
0	0	0	1	0	1
0	0	0	1	1	0
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	0	1
0	0	1	0	1	0
0	0	1	0	1	1
0	0	1	1	0	0
0	0	1	1	0	1
0	0	1	1	1	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	0	0	1
0	1	0	0	1	0
0	1	0	0	1	1
0	1	0	1	0	0
0	1	0	1	0	1
0	1	0	1	1	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	0	1
0	1	1	0	1	0
0	1	1	0	1	1
0	1	1	1	0	0
0	1	1	1	0	1
0	1	1	1	1	0
0	1	1	1	1	1

Независимость блоков

- Пусть блок состоит из 2^b нитей
  - Например, $b = 3$, в блоке 8 нитей
- Расстояние между отрезками суперпозиции, рассчитываемыми блоком, уменьшается с увеличением индекса кубита
- При $i \geq n - (b + 1)$ блок считает непрерывный отрезок суперпозиции
  - n - число кубитов, i – индекс кубита, каждая нить считает две амплитуды,
- Отрезки разных блоков не пересекаются

0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	0	1	1
0	0	0	1	0	0
0	0	0	1	0	1
0	0	0	1	1	0
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	0	1
0	0	1	0	1	0
0	0	1	0	1	1
0	0	1	1	0	0
0	0	1	1	0	1
0	0	1	1	1	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	0	0	1
0	1	0	0	1	0
0	1	0	0	1	1
0	1	0	1	0	0
0	1	0	1	0	1
0	1	0	1	1	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	0	1
0	1	1	0	1	0
0	1	1	0	1	1
0	1	1	1	0	0
0	1	1	1	0	1
0	1	1	1	1	0
0	1	1	1	1	1

0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	0	1	1
0	0	0	1	0	0
0	0	0	1	0	1
0	0	0	1	1	0
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	0	1
0	0	1	0	1	0
0	0	1	0	1	1
0	0	1	1	0	0
0	0	1	1	0	1
0	0	1	1	1	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	0	0	1
0	1	0	0	1	0
0	1	0	0	1	1
0	1	0	1	0	0
0	1	0	1	0	1
0	1	0	1	1	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	0	1
0	1	1	0	1	0
0	1	1	0	1	1
0	1	1	1	0	0
0	1	1	1	0	1
0	1	1	1	1	0
0	1	1	1	1	1

0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	0	1	1
0	0	0	1	0	0
0	0	0	1	0	1
0	0	0	1	1	0
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	0	1
0	0	1	0	1	0
0	0	1	0	1	1
0	0	1	1	0	0
0	0	1	1	0	1
0	0	1	1	1	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	0	0	1
0	1	0	0	1	0
0	1	0	0	1	1
0	1	0	1	0	0
0	1	0	1	0	1
0	1	0	1	1	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	0	1
0	1	1	0	1	0
0	1	1	0	1	1
0	1	1	1	0	0
0	1	1	1	0	1
0	1	1	1	1	0
0	1	1	1	1	1

0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	0	1	1
0	0	0	1	0	0
0	0	0	1	0	1
0	0	0	1	1	0
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	0	1
0	0	1	0	1	0
0	0	1	0	1	1
0	0	1	1	0	0
0	0	1	1	0	1
0	0	1	1	1	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	0	0	1
0	1	0	0	1	0
0	1	0	0	1	1
0	1	0	1	0	0
0	1	0	1	0	1
0	1	0	1	1	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	0	1
0	1	1	0	1	0
0	1	1	0	1	1
0	1	1	1	0	0
0	1	1	1	0	1
0	1	1	1	1	0
0	1	1	1	1	1

Суть задания

- Последовательно применить квантовый вентиль к кубитам с индексами i:

$$\left(n - (\log_2(BLOCK\_SIZE) + 1)\right) \leq i \leq (n - 1)$$

- $BLOCK\_SIZE$ – число нитей в блоке, $BLOCK\_SIZE = 2^b$

$$\left(n - (b + 1)\right) \leq i \leq (n - 1)$$

  - Последовательно применить квантовый вентиль к последним $b + 1$ кубитам

Суть задания

- $BLOCK\_SIZE$ – число нитей в блоке,
  $BLOCK\_SIZE = 2^b$, $\big(n - (b+1)\big) \leq i \leq (n-1)$

- На каждой итерации блоки работают с непересекающимися отрезками квантовой суперпозиции

  - Нет необходимости в синхронизации между блоками в конце итерации, каждый блок работает отдельно от других
    - **Можем хранить промежуточные результаты в общей памяти**

  - Нужна синхронизация внутри блока

Описание программы

- Параметры программы:

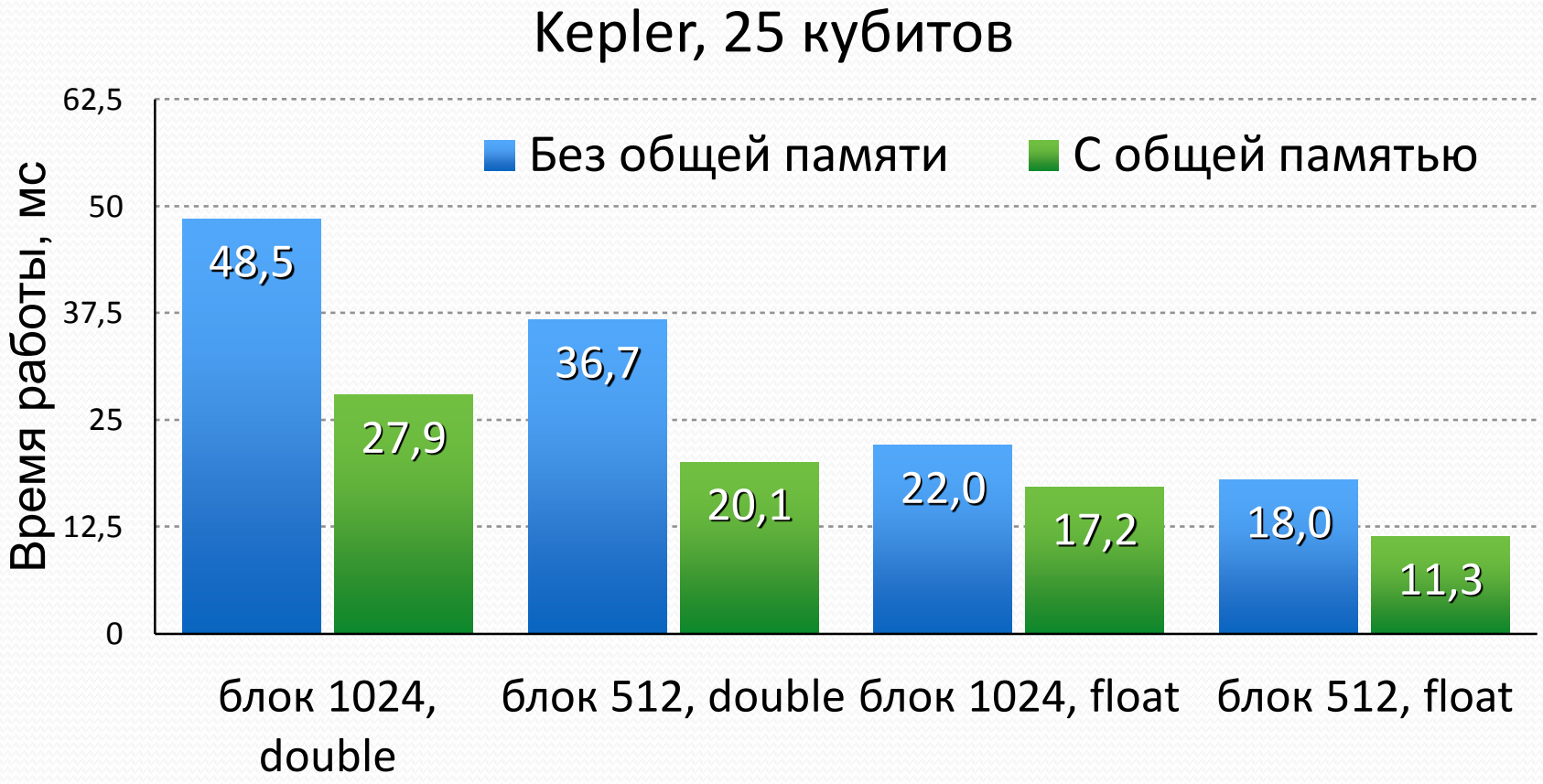
`-n <number of qubits>`	- число кубитов в регистре
`-block <block size>`	- размер блока нитей, степень двойки
`-check`	- флаг проверки результата
`-v`	- флаг вывода в консоль регистров и квантового вентиля

- Вывод программы:
  - Время работы на GPU с общей памятью и без
  - Время работы на CPU, если включен флаг проверки результата

Требуется реализовать

- Последовательное применение вентиля к последним $b + 1$ кубитам на **CPU**
- **CUDA-ядро**, последовательно применяющее вентиль к последним $b + 1$ кубитам
- **CUDA-ядро**, последовательно применяющее вентиль к последним $b + 1$ кубитам. Промежуточные результаты **сохранять в общей памяти**
- CPU-функцию для управления памятью устройства, запуска двух ядер и замера времени их выполнения

Тесты


Kepler, 25 кубитов
Время работы, мс
62,5
50
37,5
25
12,5
0
Без общей памяти
С общей памятью
48,5
27,9
36,7
20,1
22,0
17,2
18,0
11,3
блок 1024, double
блок 512, double
блок 1024, float
блок 512, float

Тесты

- На Fermi разница в производительности с общей памятью и без неё меньше, чем на Kepler, т.к. промежуточные результаты полностью влезают в кеш
  - Лично у меня на Fermi нет ускорения на `float` и есть небольшое на `double`
- На Kepler ускорение больше, т.к. на нем запросы в глобальную память не кешируются (см. в cuda-programming-guide)
- На Fermi можно замедлить вариант без общей памяти, указав `-Xptxas -dlcm=cg` при компиляции (выключает кеширование)

The end