Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Архитектура процессоров и технология CUDA

ОТЧЕТ
по лабораторной работе №1
на тему

# ИЗУЧЕНИЕ ОСОБЕННОСТЕЙ РАБОТЫ КЭШ-ПАМЯТИ

Выполнили: Д.И. Скачков
А.Д. Семков

Проверил: Т.С. Жук

МИНСК 2024

## 1 ЦЕЛЬ РАБОТЫ

В ходе выполнения данной лабораторной работы необходимо изучить особенности работы кэш-памяти процессора.

## 2 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1) Получить у преподавателя задания по лабораторной работе.
2) Реализовать заданные алгоритмы умножения матриц.
3) Получить и проанализировать результаты.
3) Оформить отчет.

## 3 ХОД РАБОТЫ

### 3.1 Листинг кода

```
use std::time::Instant;
use std::cmp::min;
use rand::Rng;

// Размеры кэша
enum CacheSize {
    L1 = 320 * 1024,     // Размер L1 кэша в байтах
    L2 = 5 *1024 * 1024,    // Размер L2 кэша в байтах
    L3 = 12 * 1024 * 1024, // Размер L3 кэша в байтах
}

// Структура матрицы
struct Matrix {
    data: Vec<Vec<f32>>,
    size: usize,
}

impl Matrix {
    fn new(size: usize) -> Matrix {
        let mut rng = rand::thread_rng();
        let data = (0..size)
            .map(|_| (0..size)
                .map(|_| rng.gen::<f32>() * 100.0)
                .collect())
            .collect();
        Matrix { data, size }
    }

    fn zero(size: usize) -> Matrix {
        let data = vec![vec![0.0; size]; size];
        Matrix { data, size }
    }

    fn compare(&self, other: &Matrix) -> bool {
        if self.size != other.size {
            println!("Размеры не равны");
```

```
                    return false;
                }
                for i in 0..self.size {
                    for j in 0..self.size {
                        if (self.data[i][j] - other.data[i][j]).abs() >
1e-6 {
                            return false;
                        }
                    }
                }
                true
            }

            // Стандартное умножение матриц
            fn standard_multiply(&mut self, a: &Matrix, b: &Matrix) {
                for i in 0..self.size {
                    for j in 0..self.size {
                        self.data[i][j] = 0.0;
                        for k in 0..self.size {
                            self.data[i][j] += a.data[i][k] *
b.data[k][j];
                        }
                    }
                }
            }

            // Блочное умножение матриц
            fn block_multiply(&mut self, a: &Matrix, b: &Matrix,
block_size: usize) {
                for i in (0..self.size).step_by(block_size) {
                    for j in (0..self.size).step_by(block_size) {
                        for k in (0..self.size).step_by(block_size) {
                            for ii in i..min(i + block_size, self.size) {
                                for jj in j..min(j + block_size,
self.size) {
                                    let mut sum = 0.0;
                                    for kk in k..min(k + block_size,
self.size) {
                                        sum += a.data[ii][kk] *
b.data[kk][jj];
                                    }
                                    self.data[ii][jj] += sum;
                                }
                            }
                        }
                    }
                }
            }
        }

        // Вычисление размера блока
        fn calculate_block_size(cache_size: usize) -> usize {
            let mut max_block_size = (cache_size as f64 / 3.0 * 0.9).sqrt()
/ std::mem::size_of::<f32>() as f64;
            let mut block_size = max_block_size as usize;
```

```
        if block_size % 64 != 0 {
            block_size -= block_size % 64;
        }

        let min_block_size = 16;
        if block_size < min_block_size {
            block_size = min_block_size;
        }

        block_size
    }

    fn main() {
        let l3_cache_size = CacheSize::L3 as usize;
        let l2_cache_size = CacheSize::L2 as usize;
        let l1_cache_size = CacheSize::L1 as usize;

        let n = ((l3_cache_size / std::mem::size_of::<f32>()) as
f64).sqrt() as usize * 2;
        println!("Размер матрицы: {}x{}", n, n);

        let a = Matrix::new(n);
        let b = Matrix::new(n);
        let mut c1 = Matrix::zero(n);
        let mut c2 = Matrix::zero(n);
        let mut c3 = Matrix::zero(n);
        let mut c4 = Matrix::zero(n);

        let start1 = Instant::now();
        c1.standard_multiply(&a, &b);
        let duration1 = start1.elapsed();
        println!("Время стандартного умножения: {:?}", duration1);

        let block_size_l3 = calculate_block_size(l3_cache_size);
        println!("Размер блока для L3 кэша: {}x{}", block_size_l3,
block_size_l3);

        let start2 = Instant::now();
        c2.block_multiply(&a, &b, block_size_l3);
        let duration2 = start2.elapsed();
        println!("Время блочного умножения (L3 кэша): {:?}",
duration2);

        let block_size_l2 = calculate_block_size(l2_cache_size);
        println!("Размер блока для L2 кэша: {}x{}", block_size_l2,
block_size_l2);

        let start3 = Instant::now();

        c3.block_multiply(&a, &b, block_size_l2);

        let duration3 = start3.elapsed();
        println!("Время блочного умножения (L2 кэша): {:?}",
duration3);
```

### 3.2 Анализ результатов

Матрица размером 3596x3596:
Обычный способ: 4.03475 с
L3-кэш: 0.159958 с
Разница во времени вычисления в 25,22 раза.

Матрица разбивается на блоки, которые могут поместиться в кэш-памяти и перемножаются по стандартному алгоритму. Такой способ уменьшает количество обращений к оперативной памяти и повышает эффективность умножения.

## 4 ВЫВОД

В ходе лабораторной работы были изучены особенности работы кэш-памяти и получены навыки эффективного ее использования для ускорения и оптимизации вычислений больших объемов данных.