Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра информатики

Отчет по лабораторной работе №5

Кеш

Выполнил: Студент: гр. 053506
Слуцкий Н. С.

Руководитель: ст. преподаватель
Шиманский В.В.

Минск 2022

Содержание

Введение

Цели лабораторной работы:

1 Изучить другие возможности симулятора Venus.

2 Исследовать принцип работы кэша, чем кэш отличается от обычной памяти, изучить виды записей в кэш и память.

3 Выполнить упражнения.

Постановка задачи

В нескольких лабораторных работах мы будем работать с файлами ассемблера RISC-V, каждый из которых имеет расширение файла .s. Для их запуска мы будем использовать Venus, образовательный ассемблер и симулятор RISC-V. Вы можете запустить Venus локально из своего терминала или из браузера Venus, а следующие инструкции помогут вам выполнить все шаги по его настройке. Однако, для наших лабораторных вам может быть удобнее использовать веб-редактор.

Упражнение 1: Симулятор кэша Venus

Ход работы:

Прочитайте cache.s, чтобы понять, что делает программа. Смоделируйте в Venus следующие 3 сценария и ответьте на соответствующие вопросы.

Сценарий 1:

Параметры программы:(установите их, инициализировав регистры “a” в коде)
Размер массива (a0): **128 (байт)**
Размер шага (a1): **8**
Количество повторений (a2): **4**
Вариант (a3): **0**
Параметры кеша: (установите их на вкладке Cache)
Уровни кеша: **1**
Размер блока: **8**
Количество блоков: **4**
Включить?: **Должно быть отмечено зеленым**
Политика размещения: **Прямое отображение**
Ассоциативность: **1** (Venus не позволяет вам изменить этот параметр с политикой размещения прямого отображения, почему?)
Политика замены блоков: **LRU**

Вопросы

1. Какая комбинация параметров обеспечивает наблюдаемую вами частоту попаданий? - «Благодаря тому, что размер шага **a1** в байтах точно равен размеру блока кэша в байтах».

2. Какова будет наша частота попаданий, если мы произвольно увеличим количество

повторений? - Частота попаданий равна 0, потому что изначально она не зависела от количества повторений, а зависит от шага (регистра **a1**).

3. Как мы можем изменить один параметр программы, чтобы получить максимально возможную частоту попаданий? - Давайте изменим «**a1, 1**» и тогда получим частоту попаданий равную 0.5.

Сценарий 2:

Параметры программы: (установите их, инициализировав регистры "a" в коде)

Размер массива (a0)**: 256 (байт)**
Размер шага (a1)**: 2**
Количество повторений (a2)**: 1**
Вариант (a3)**: 1**
Параметры кеша: (задайте их на вкладке Cache)
Уровни кеша: **1**
Размер блока: **16**
Количество блоков: **16**
Включить?: **Должен быть зеленым**
Политика размещения: **Ассоциативный кеш с N-образным набором**
Ассоциативность: **2**
Политика замены блоков: **LRU**

Вопросы

. Сколько обращений к памяти приходится на одну итерацию внутреннего цикла (не связанно с числом повторений)? - На одну итерацию внутреннего цикла приходится 2 обращения к памяти.

. Какой повторяющийся шаблон попаданий/промахов? - Самый короткий шаблон попаданий/промахов (попаданий — **H**, промахов — **M**) это шаблон **MH,** попадается при обращению к каждому «новому» блоку кэша.

. Сохраняя все остальное без изменений, к чему приближается наша частота попаданий, когда количество повторений достигает бесконечности? - Частота попаданий приближается к 99/100.

Сценарий 3

Параметры программы: (установите их, инициализировав регистры a в коде)
Размер массива (a0): **128 (байт)**
Размер шага (a1): **1**
Количество повторений (a2): **1**
Вариант (a3): **0**

Параметры кеша: (задайте их на вкладке Cache)
L1:
Уровни кеша: **2**
Размер блока: **8**

Количество блоков: **8**
Включить?: **Должен быть зеленым**
Политика размещения: **Прямое отображение**
Ассоциативность: **1**
Политика замены блоков: **LRU**

L2:
Размер блока: **8**
Количество блоков: **16**
Включить?: **Должен быть зеленым**
Политика размещения: **Прямое отображение**
Ассоциативность: **1**
Политика замены блоков: **LRU**

Вопросы:

1. **Какова частота попаданий в кеш L1? В кеш второго уровня? Общая?**- [L1, 0.5], [L2, 0], [L1+L2, 0.5]

2. **Сколько всего произошло доступов к кешу L1? Сколько из них промахнулись?** - [Обращения, 32], [Количество промахов, 16].

3. **Сколько у произошло доступов к кешу L2?** - [Обращения, 16], [Количество промахов, 16]. К кэшу **L2** произошло в 2 раза меньше доступов, потому что у **L2** общий размер кэша 128 байт, а у **L1** 64 байта, а по нашему сценарию размер исходного массива 128 байт. Вероятно, чтобы мы могли получить доступ к кэшу **L2** у нас не должно быть доступа к данным кэша **L1**.

4. **Какой программный параметр позволил бы нам увеличить частоту попаданий L2, но сохранить частоту попаданий L1 на прежнем уровне?** - Нам нужно увеличить количество повторений в регистре **a2.**

5. **Наши показатели попаданий L1 и L2 уменьшаются (-), остаются прежними (=) или увеличиваются (+), когда мы** (1) **увеличиваем количество блоков в L1 или** (2) **увеличиваем размер блока L1?** -
Во всех протестированных случаях наши показатели попаданий увеличелись. [1_L1_>], [2_L1_>], [1_L2_>], [2_L2_>].

Упражнение 2 — Упорядочивание циклов и умножение матриц.

Ход работы

На нашем локальном компьютере находится файл MatrixMultiply.c, где есть 6 способов реализации умножения матриц. Нам необходимо было сравнить скорость работы каждой из них и ответить на вопросы.

Результаты работы программы:

```
ijk:    n = 1000, 0.764 Gflop/s
ikj:    n = 1000, 0.109 Gflop/s
jik:    n = 1000, 0.707 Gflop/s
jki:    n = 1000, 5.091 Gflop/s
kij:    n = 1000, 0.111 Gflop/s
kji:    n = 1000, 4.760 Gflop/s
```

Вопросы:

1. Какие 2 порядка обхода матриц лучше всего подходят для матриц 1000 на 1000 на вашем компьютере? -

На компьютере с Ryzen 7 4800H лучше всего подходят порядки обхода **jki** и **kji**.

2. Какие 2 порядка обхода матриц работают хуже всего на вашем компьютере? - Хуже всего на этом компьютере работают обходы **ikj** и **kji**.

Упражнение 3 — Блокирование кэша и транспонирование матрицы.

Ход работы

Ваша задача - реализовать блокирование кеша в функции transpose_blocking() в файле **transpose.c.**
Вы НЕ можете считать, что ширина матрицы (n) кратна размеру блока.
По умолчанию функция ничего не делает, поэтому функция тестирования сообщит об ошибке.

Листинг кода:

```c
#include "transpose.h"

/* The naive transpose function as a reference. */
void transpose_naive(int n, int blocksize, int *dst, int *src) {
    for (int x = 0; x < n; x++) {
        for (int y = 0; y < n; y++) {
            dst[y + x * n] = src[x + y * n];
        }
    }
}

void transpose_blocking(int n, int blocksize, int *dst, int *src) {
    for (int i = 0; i < n; i += blocksize) {
        for (int j = 0; j < n; j += blocksize) {
            for (int l = i; l < i + blocksize; l++) {
                for (int k = j; k < j + blocksize; k++) {
                    dst[k + l * n] = src[l + k * n];
                }
            }
        }
    }
}
```

Вопросы:

Часть 1 — Изменение размера массива

1. **В какой момент времени версия транспонирования с разделением на блоки становится быстрее, чем версия без разделения?**

- Начиная размера массива на 2000 элементов.

```
Testing naive transpose: 17.274 milliseconds
Testing transpose with blocking: 4.749 milliseconds
```

2. **Почему разделение на блоки требует, чтобы матрица была определенного размера, прежде чем она превзойдет код без разделения на блоки?**

- Вероятно, дело в скорости алгоритма — и на маленьких значениях массива выходит невыгодно разделять на блоки.

Часть 2 — Изменение размера блока

1. Как изменяется производительность при увеличении размера блоков? Почему так происходит? - Изначально при увеличении размера блоков производительность увеличивается (уменьшается время выполнения программы)

```
Testing naive transpose: 142.437 milliseconds
Testing transpose with blocking: 34.895 milliseconds
```

Время выполнения для массива на 5000 элементов и 200 блоков.

Но с ещё большим увеличением числа блоков время выполнения снова замедляется и нужно искать «золотую середину».

```
Testing naive transpose: 205.788 milliseconds
Testing transpose with blocking: 127.929 milliseconds
```

Время выполнения для массива на 5000 элементов и 2500 блоков.

Это происходит из-за того, что при определённом числе блоков процесс становится трудоёмким в смысле декомпозиции и обратного сбора этих блоков, что отражается на времени выполнения программы.

Выводы

В результате выполнения лабораторной работы №5 были изучены разновидности кэширования памяти в процессоре. А также в каком-то смысле просимулированы какие-то действия из этой темы. Цели лабораторной работы можно считать достигнутыми.

Литература

Харрис, Дэвид; Харрис, Сара «Цифровая схемотехника и архитектура компьютера. RISC-V» ДМК, 2022.