Київський національний університет імені Тараса Шевченка

Лабораторна робота №1

«Визначення швидкодії обчислювальної системи»

Виконав студент першого курсу
Факультету комп'ютерних наук і кібернетики
Волік Федір Михайлович (група К-15)

Київ, 06.02.23

Зміст

1 Умова лабораторної роботи

Мета лабороторної роботи полягає у порівнянні швидкості виконання базових операцій над числовими типами для певної обчислювальної системи. Для досягнення цієї мети пропонується розробити програму, яка вимірює кількість виконуваних базових операцій за певний проміжок часу. Вимірювання "чистої"команди процесора не вимагається.

2 Постановка задачі

Для реалізації програми буде використовуватись мова програмування C++, яка передбачає роботу з низькорівневими операціями.

Мова програмування C++ пропонує для використання широкий вибір числових типів та декілька способів їх запису.

2.1 Цілочисельні типи

Цілочисельні типи різняться за наявністю знака(signed та unsigned модифікатори) та розміром(кількістю зайнятих бітів). Стандарт C++11 ввів ряд назв для цих типів, які дозволяють чітко розуміти їх структуру. Відоповідно, назва типу має форма {u}int{SIZE}_t. Якщо символ u присутній у записі, то тип "беззнаковий"(unsigned), інаше "знаковий"(signed). SIZE може приймати значення 8, 16, 32, 64 — відповідно, кількість бітів типа. Отже, усі цілочисельні типи можна побачити у таблиці:

Знаковість × Розмір	8	16	32	64
signed	int8_t	int16_t	int32_t	int64_t
unsigned	uint8_t	uint16_t	uint32_t	uint64_t

2.2 Дробові типи

Дробові типи представлені типами float та double — відповідно, 32-бітні та 64-бітні дробові числа.

2.3 Базові операції

В мові програмування C++ існує принаймні 4 бінарні операції, які визначені для усіх числових типів: +(додавання), -(віднімання), *(множення) та /(ділення). Крім того, операція %(ділення націло) визначена для усіх цілочисельних типів. Виходячи з цих міркувань, базовими називатимемо саме зазначені 5 операцій.

3 Побудова моделі

Для виміру часу необхідно зафіксувати два моменти — відповідно до та після виконання операції. Однак окрема операція виконується занадто швидко, тому необхідно виконувати її багато разів.

Операції, з якими ми працюємо — бінарні, отже виникає необхідність генерувати для кожної операції 2 операнди. Я бачу декілька варіантів підходу цього питання:

1. Оперувати двома константними значеннями — зовсім не підходить, адже компілятор мови C++ навіть з опцією -O0, яка вимикає усі оптимізації, усе одно підраховує значення усіх операцій над константними операндами на етапі компіляції.

2. Оперувати двома неконстантними, але сталими значеннями — вважається, що робота з різними значеннями змінних одного типу займає однакову кількість часу.

3. Оперувати двома випадковими значеннями. Цей спосіб частково вирішує проблему варіанту №2, але вимагає значної за розміром тестової бази.

4. Виконати операцію над кожною можливою парою значень типу, або ж над кожною можливою парою з рівномірної вибірки значень типу.

Після декількох експериментів, я дійшов висновку, що найкращим насправді є спосіб 3, адже він пов'язаний з найменшою кількістю "зайвих"операції під час тестування. Генерація випадкових чисел у сучасній мові C++ виявилася напрочуд стабільною.

Важливою частиною цієї роботи є аналіз даних тестування, і для цієї задачі, на мою думку, середовище мови C++ не є найкращою опцією. Відповідно, тестуваняя відбуватиметься в виконуваних файлах написаних мовою C++, а вихідні дані тестів пізніше оброблятимуться іншими інструментами.

4 Проєктування та реалізація алгоритму

4.1 Багаторазове виконання

Алгоритм має передбачати багаторазове виконання усіх операцій для покращення точності результатів тестування. Цього можна досягти простим циклом while. Однак через таке "обгортання"у заміри часу потраплятимуть "мусорні"операції — робота з циклами. Отже, окрім цикла while, пропоную виконувати операції усередині циклів велику кількість разів. Для генерації коду в цьому випадку використовуватимо можливості препроцессору мови C++ — макроси.

```
#define x10(ex) ex; ex; ex; ex; ex; ex; ex; ex; ex; ex
#define x100(ex) x10(x10(ex))
#define x1000(ex) x10(x10(x10(ex)))
#define xtimes(ex) x1000(ex) // macros of choice
#define XTIMES_LABEL ("1000")
```

Таким чином, викликом макросу $xtimes(expression)$; ми вставляємо в код 1000 викликів коду $expression$, а $XTIMES_LABEL$ можемо використовувати серед вихідних даних програми.

4.2 Визначення операції

Операцію, яку ми тестуємо, також можна визначити за допомогою системи макросів.

```
#define OP_NOTHING 0
#define OP_ADD 1
#define OP_SUBTRACT 2
#define OP_MULTIPLY 3
#define OP_DIVIDE 4
#define OP_MODULO 5

#if OP_TYPE == OP_NOTHING
```

```
#define OP(r, a, b) ;
#define OP_LABEL ("nothing")
#elif OP_TYPE == OP_ADD
#define OP(r, a, b) (r = (a) + (b))
#define OP_LABEL ("add")
#elif OP_TYPE == OP_SUBTRACT
#define OP(r, a, b) (r = (a) - (b))
#define OP_LABEL ("subtract")
#elif OP_TYPE == OP_MULTIPLY
#define OP(r, a, b) (r = (a) * (b))
#define OP_LABEL ("multiply")
#elif OP_TYPE == OP_DIVIDE
#define OP(r, a, b) (r = (a) / (b))
#define OP_LABEL ("divide")
#elif OP_TYPE == OP_MODULO
#define OP(r, a, b) (r = (a) % (b))
#define OP_LABEL ("modulo")
#endif
```

g++ може отримати серед аргументів флаг -DNAME=VAL, який еквівалентний запису $\#define\ NAME\ VAL$ у мові препроцессора. Таким чином, запис $OP(result, a, b)$ компілятор перетворить на відповідну операцію.

4.3 Випадкові числа

Генерація (псевдо-)випадкових чисел відбувається за допомогою функціоналу модулів <random> та <limits>.

```
#include <random>
#include <limits>
// ...
// necessary initialization
std::random_device device;
std::mt19937_64 gen(device());
std::uniform_int_distribution<T>
   distribution(std::numeric_limits<T>::min(),
```

```
   std::numeric_limits<T>::max())
// actual generation
auto value = distribution(gen);
```

Для генерації дробових чисел достатньо замінити *uniform_int_distribution* на *uniform_real_distribution*.

4.4 Замір часу

Для виміру часу використовується модуль <chrono>. В його просторі імен визначено об'єкт high_resolution_clock з функцією now(), яка повертає поточний час системи типу high_resolution_clock::time_point. Два зафіксовані "моменти"часу можна відняти один від одного, і отримати різницю в часі, яку можна привести до певного формату за допомогою об'єкту std::chrono::duration_cast<T> з викликом функції count(), де T — тип одиниць виміру часу, наприклад std::chrono::milliseconds або std::crhono::microseconds.

4.5 Тестуюча функція

Нарешті, збираючи усі компоненти до купи, ми можемо записати тестуючу функцію для цілих типів:

```
template<typename T>
time_unit test_int(uint64_t repeat) {
    std::random_device device;
    std::mt19937_64 gen(device());
    std::uniform_int_distribution<T>
       distribution(std::numeric_limits<T>::min(),
       std::numeric_limits<T>::max());

    std::chrono::high_resolution_clock::time_point start, finish;
    T left, right;
    T result;
```

```
    start = std::chrono::high_resolution_clock::now();
    while (repeat--) {
        left = distribution(gen);
        right = distribution(gen);
        #if OP_TYPE == OP_DIVIDE || OP_TYPE == OP_MODULO
        if (right == 0) {
            right = 1;
        }
        #endif
        xtimes(OP(result, left, right));
    }
    finish = std::chrono::high_resolution_clock::now();

    return std::chrono::duration_cast<time_unit>(finish - start);
}
```

Слід зазначити, що для того щоб уникнути проблему ділення на 0 в цілих типах використовується ще одна директива препроцессора($\#if\ OP\_TYPE\ ==\ ...$)

Відповідно, для роботи з дробовими числами необхідно замінити в цьому коді $uniform\_int\_distribution$ на $uniform\_real\_distribution$, а "захисний"макрос можна прибрати.

4.6 Обробка результатів тесту

Наступним кроком після виконання тесту є організація та передача результатів. Для цієї мети застосуємо наступну функцію:

```
template<typename T>
void test_int_wrapper(const std::string& type_label, const
   uint64_t& repeat) {
    auto time = test_int<T>(repeat).count();

    std::forward_list<std::string> data {
        type_label,
        std::to_string(sizeof(T) * 8),
```

```
        XTIMES_LABEL,
        std::to_string(repeat),
        OP_LABEL,
        std::to_string(time),
    };

    std::cout << data.front();
    data.pop_front();
    for (auto item: data) {
        std::cout << "," << item;
    }
    std::cout << std::endl;
}
```

Ми використовуємо *forward_list* для того щоб зберегти необхідні вихідні дані, перед тим як вивести їх за допомогою просто цикла *for*.

Аналогічна функція існує для дробових типів.

4.7 Функція *main*

Функція *main* є керуючою функцією програми, де контролюються усі параметри тестування. Надзвичайно зручними для виконання програми є інтерфейс аргументів командного рядка, тому наша функція була зпрограмована для підтримки такого інтерфейсу. Кінцева програма приймає будь-яку кількість аргументів, які визначають, які тести будуть виконані, і які додаткові дані будуть продемонстровані. Це реалізовано за допомогою просто циклу *for*, який ітерується по наданим аргументам, і виконує відповідні дії.

```
int main(int argc, const char *argv[]) {
    for (int i = 1; i < argc; i++) {
        auto arg = std::string(argv[i]);

        if (!arg.compare("labels")) {
            std::cout << "type,size,x,repeat,operation,time" <<
```

```
        std::endl;
    continue;
}

// const uint64_t repeat = (1ULL << 24);
const uint64_t repeat = (1ULL << 12);

if (!arg.compare("uint8_t")) {
    test_int_wrapper<uint8_t>(arg, repeat);
    continue;
}
if (!arg.compare("int8_t")) {
    test_int_wrapper<int8_t>(arg, repeat);
    continue;
}

if (!arg.compare("uint16_t")) {
    test_int_wrapper<uint16_t>(arg, repeat);
    continue;
}
if (!arg.compare("int16_t")) {
    test_int_wrapper<int16_t>(arg, repeat);
    continue;
}

if (!arg.compare("uint32_t")) {
    test_int_wrapper<uint32_t>(arg, repeat);
    continue;
}
if (!arg.compare("int32_t")) {
    test_int_wrapper<int32_t>(arg, repeat);
    continue;
}

if (!arg.compare("uint64_t")) {
    test_int_wrapper<uint64_t>(arg, repeat);
    continue;
```

```
        }
        if (!arg.compare("int64_t")) {
            test_int_wrapper<int64_t>(arg, repeat);
            continue;
        }

        if (!arg.compare("float")) {
            test_real_wrapper<float>(arg, repeat);
            continue;
        }
        if (!arg.compare("double")) {
            test_real_wrapper<double>(arg, repeat);
            continue;
        }
    }

    return 0;
}
```

Для комфортної взаємодії з даними, які генерує програма, використовується декілька "скриптів"мовою Python та Bash. Однак їх структура, на мою, думку, знаходиться поза межами тематики цієї роботи. Вміст скриптів та інших файлів можна знайти у репозиторії проєкту.

5 Приклади та аналіз виконання програми

Отже, було згенеровано виконувані файли(підпрограми) — по одному на кожну операцію, після чого кожна з підпрограм запускалася на виконання з кожним цільовим типом. Виявилось, що дані, які ми отримуємо, є досить стабільними при різних параметрах виконання. Дані з найбільшого тесту представлені у таблиці нижче. За стовпчиками розрізняємо операції, за рядками — типи. У клітинках знаходяться значення виду $speed/max\_speed$ — швидкість операції на типі ділене на максимальну швидкість з усієї таблиці.

	add	subtract	multiply	divide	modulo
$uint8\_t$	0.99	0.79	0.93	0.17	0.17
$int8\_t$	0.79	0.65	0.49	0.17	0.17
$uint16\_t$	0.92	0.75	0.87	0.17	0.17
$int16\_t$	0.76	0.57	0.78	0.17	0.17
$uint32\_t$	0.99	0.80	0.97	0.17	0.17
$int32\_t$	1.00	0.80	0.98	0.17	0.17
$uint64\_t$	0.81	0.74	0.81	0.10	0.10
$int64\_t$	0.81	0.75	0.81	0.10	0.10
$float$	0.72	0.68	0.72	0.33	
$double$	0.69	0.69	0.71	0.25	

Судячих з цих даних, можна зробити припущення, що операції *divide* та *modulo* фактично використовують одну й ту саму реалізацію у кожному типі окремо, і що типи з $uint8\_t$ по $int32\_t$ взагалі мають однакову реалізацію цих операцій. Не можна не помітити попарну схожість типів $uint32\_t$ й $int32\_t$, $uint64\_t$ й $int64\_t$, $float$ й *double*, з чого можна робити припущення про спорідненість цих типів. Переважна кількість інших даних не демонструють таких схожостей, однак можна помітит загальні тенденції. Операції ділення та ділення націло є найповільнішими в усих типах. Як не

дивно, на першому місці за швидкістю в частині випадків знаходяться додавання та множення, хоча я б очікував завжди бачити в цьому контексті саме додавання та віднімання.

6 Висновки

В результаті виконання лобораторної роботи вдалося досягти стабільних результатів тестування, а отже остаточно порівняти швидкість виконання різних операцій з різними типами на моїй обчислювальній системі. Акцент на зв'язок між конкретною системою та результатами тестування є важливим, адже при іншій реалізації швидкість виконання певних операцій може помітно відрізнятися.

Під час розробки програми модель алгоритму та її реалізація змінювалися декілька разів. Це виявилося необхідним для розуміння проблем та дослідженням можливих розв'язків, пов'язаних з тематикою роботи. Досягнути кінцевої мети можна було б швидше, якби менше уваги приділялося повній реалізації окремої ідеї, і більше часу було використано не експериментування з різними концепціями.

7 Джерела та корисні посилання

1. Умова лабораторної роботи:
   https://sites.google.com/site/byvkyiv1/arhiteom_stac/arhiteom_lab_01

2. Вебсайт з документацією до мови C++ — cppreference.com:
   https://en.cppreference.com/w/

3. Вебсайт з онлайн-інструментом для роботи з LaTeX, який використовувався для написання цього звіту — overleaf.com:
   https://www.overleaf.com/

4. Репозиторій з лабораторною роботою:
   https://github.com/WizardPlatypus/ACS