Лабораторная работа

2023-12-11

Данная лабораторная работа подготовлена Кириленко Яковом Александровичем по заказу [Альянса RISC‑V](https://riscv-alliance.ru/), допускается к использованию под лицензией [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/deed.ru).

Приведённые ниже команды рассчитаны на выполнение в рабочей среде *Syntacore Kit*, распространяемой [Альянсом RISC‑V](https://riscv-alliance.ru/) в образовательных целях. Обратите внимание, что в другом окружении приведённые примеры команд могут не работать или работать иначе.

Рассмотрите программу, считающую скалярное произведение векторов.

**main.c**

#include <stdlib.h>  
#include <stdio.h>  
#include <memory.h>  
#include <time.h>  
#include <malloc.h>  
  
const size\_t N = 200000;  
const size\_t K = 2000;  
  
typedef float cell\_t;  
typedef cell\_t \* cell\_ptr\_t;  
  
cell\_t dot\_product(cell\_ptr\_t a, cell\_ptr\_t b, size\_t n) {  
 cell\_t r = 0;  
 for (size\_t i = 0; i < n; ++i) {  
 r += a[i] \* b[i];  
 }  
 return r;  
}  
  
cell\_ptr\_t allocate() {  
 cell\_ptr\_t r = (cell\_t\*) malloc(sizeof(cell\_t) \* N);  
 for (size\_t i = 1; i < N; ++i) {  
 r[i] = (cell\_t) (N \* rand() / RAND\_MAX);  
 }  
 return r;  
}  
  
int main() {  
 srand(0); // reproducible PRNG  
 cell\_ptr\_t a = allocate();  
 cell\_ptr\_t b = allocate();  
 cell\_t r1 = 0;  
 clock\_t t = clock();  
 for (size\_t i = 0; i < K; ++i) {  
 int x = (t % (i + 1)); // some number to force dot\_product call  
 r1 += dot\_product( (x & 2) ? a : b  
 , (x & 1) ? b : a  
 , N);  
 }  
 double time1 = (double)(clock()-t)/CLOCKS\_PER\_SEC;  
  
 printf("%.3g\n%.3g\n"  
 , r1  
 , time1);  
  
 free((void \*)a);  
 free((void \*)b);  
 return 0;  
}

# Кросс-компиляция и запуск

Перед выполнением команд проверьте, определены ли переменные GCC\_ROOT или RISCV и указан ли в переменной PATH путь /opt/syntacore/sc-dt/2023.08/tools/ (2023.08 — версия Syntacore Kit):

env | grep 'GCC\_ROOT\|RISCV'  
echo $PATH

Если нет, добавьте в файл ~/.bashrc следующие строки (**пути зависят от версии используемой рабочей среды**):

**Если переменая RISCV не определена**

export GCC\_ROOT="/opt/syntacore/sc-dt/2023.08/riscv-gcc"  
export PATH=$PATH:/opt/syntacore/sc-dt/2023.08/tools/bin

**Если переменая RISCV определена**

export GCC\_ROOT=$RISCV  
export PATH=$PATH:/opt/syntacore/sc-dt/2023.08/tools/bin

Скомпилируйте программу под RISC-V. В *Syntacore Kit* для систем компиляции GCC и CLang это делается следующими способами:

|  |  |
| --- | --- |
| GCC | CLang |
| riscv64-unknown-linux-gnu-gcc main.c -o main | clang -target riscv64-unknown-linux-gnu --sysroot="$GCC\_ROOT/sysroot" --gcc-toolchain="$GCC\_ROOT" -o main main.c |

Убедитесь, что полученный исполняемый файл предназначен для RISC‑V:

file ./main

./main: ELF 64-bit LSB executable, UCB RISC-V, RVC, double-float ABI, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-riscv64-lp64d.so.1, for GNU/Linux 4.15.0, not stripped

Запустите скомпилированную программу в эмуляторе *qemu*:

qemu-riscv64 -L "$GCC\_ROOT/sysroot" ./main

# Отладка

Скомпилируйте программу под RISC-V c отладочной информацией, выставив флаг -g:

|  |  |
| --- | --- |
| GCC | CLang |
| riscv64-unknown-linux-gnu-gcc -g main.c -o main | clang -target riscv64-unknown-linux-gnu --sysroot="$GCC\_ROOT/sysroot" --gcc-toolchain="$GCC\_ROOT" -g -o main main.c |

Запустите *qemu* с сервером GDB на произвольном порту, например 12345:

qemu-riscv64 -L $GCC\_ROOT/sysroot -g 12345 ./main

Он будет ждать подключения GDB. Затем запустите GDB из системы компиляции для RISC-V (с использованием отладочных символов из файла ./main):

riscv64-unknown-linux-gnu-gdb ./main

В консоли GDB подключитесь к GDB-серверу, запущенном в *qemu*:

target remote localhost:12345

Установите sysroot, чтобы GDB мог подгрузить символы используемых динамических библиотек:

set sysroot /opt/syntacore/sc-dt/2023.08/riscv-gcc/sysroot

Затем выставьте точку останова:

b main

Продолжите выполнение программы:

с

Теперь можно отлаживать программу, запущенную в эмуляторе.

**Листинг консоли GDB после проделанных операций**

GNU gdb (GDB) 13.2  
Copyright (C) 2023 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
Type "show copying" and "show warranty" for details.  
This GDB was configured as "--host=x86\_64-pc-linux-gnu --target=riscv64-unknown-linux-gnu".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<https://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
 <http://www.gnu.org/software/gdb/documentation/>.  
  
For help, type "help".  
Type "apropos word" to search for commands related to "word"...  
Reading symbols from ./main...  
(No debugging symbols found in ./main)  
(gdb) target remote localhost:12345  
Remote debugging using localhost:12345  
warning: remote target does not support file transfer, attempting to access files from local filesystem.  
warning: Unable to find dynamic linker breakpoint function.  
GDB will be unable to debug shared library initializers  
and track explicitly loaded dynamic code.  
0x00007f5ae4fdfb40 in ?? ()  
(gdb) set sysroot /opt/syntacore/sc-dt/2023.08/riscv-gcc/sysroot  
Reading symbols from /opt/syntacore/sc-dt/2023.08/riscv-gcc/sysroot/lib/ld-linux-riscv64-lp64d.so.1...  
(No debugging symbols found in /opt/syntacore/sc-dt/2023.08/riscv-gcc/sysroot/lib/ld-linux-riscv64-lp64d.so.1)  
(gdb) b main  
Breakpoint 1 at 0x1077e  
(gdb) c  
Continuing.  
  
Breakpoint 1, 0x000000000001077e in main ()  
(gdb)

# Оптимизация

Рассмотрите листинги ассемблерного кода, полученные Clang при разных уровнях оптимизации, выполнив следующие команды:

* без оптимизаций
* clang -target riscv64-unknown-linux-gnu --sysroot="$GCC\_ROOT/sysroot" --gcc-toolchain="$GCC\_ROOT" -S -o main.s main.c
* с флагом -O1
* clang -target riscv64-unknown-linux-gnu --sysroot="$GCC\_ROOT/sysroot" --gcc-toolchain="$GCC\_ROOT" -S -o main-O1.s main.c -O1
* с флагом -O2
* clang -target riscv64-unknown-linux-gnu --sysroot="$GCC\_ROOT/sysroot" --gcc-toolchain="$GCC\_ROOT" -S -o main-O2.s main.c -O2

Будет получен следующий ассемблерный код для функции dot\_product:

**Компиляция без оптимизаций**

dot\_product: # @dot\_product  
# %bb.0:  
 addi sp, sp, -64  
 sd ra, 56(sp) # 8-byte Folded Spill  
 sd s0, 48(sp) # 8-byte Folded Spill  
 addi s0, sp, 64  
 sd a0, -24(s0)  
 sd a1, -32(s0)  
 sd a2, -40(s0)  
 li a0, 0  
 sw a0, -44(s0)  
 sd a0, -56(s0)  
 j .LBB0\_1  
.LBB0\_1: # =>This Inner Loop Header: Depth=1  
 ld a0, -56(s0)  
 ld a1, -40(s0)  
 bgeu a0, a1, .LBB0\_4  
 j .LBB0\_2  
.LBB0\_2: # in Loop: Header=BB0\_1 Depth=1  
 ld a0, -24(s0)  
 ld a1, -56(s0)  
 slli a1, a1, 2  
 add a0, a0, a1  
 flw ft0, 0(a0)  
 ld a0, -32(s0)  
 add a0, a0, a1  
 flw ft1, 0(a0)  
 flw ft2, -44(s0)  
 fmadd.s ft0, ft0, ft1, ft2  
 fsw ft0, -44(s0)  
 j .LBB0\_3  
.LBB0\_3: # in Loop: Header=BB0\_1 Depth=1  
 ld a0, -56(s0)  
 addi a0, a0, 1  
 sd a0, -56(s0)  
 j .LBB0\_1  
.LBB0\_4:  
 flw fa0, -44(s0)  
 ld ra, 56(sp) # 8-byte Folded Reload  
 ld s0, 48(sp) # 8-byte Folded Reload  
 addi sp, sp, 64  
 ret

В строках 3 — 12 происходит формирование кадра стека функции dot\_product: выделяется необходимое для аргументов и локальных переменных место на стеке (строка 3), на стеке сохраняется адрес возврата и адрес предыдущего кадра (строки 4 — 5), в регистр сохраняется адрес текущего кадра (строка 6), переданные аргументы a, b и n загружаются на стек (строки 7 — 9), локальные переменные r и i инициализируются нулями (строки 10 — 12).

В строках 15 — 17 вычисляется, нужно ли выполнять очередную итерацию цикла: со стека в регистры загружаются значения переменных i и n (строки 15 — 16), а затем сравниваются (строка 17).

В строках 20 — 30 происходит вычисление очередной итерации цикла: со стека в регистр загружается значение переменной i (строка 21), вычисляются адреса в памяти значений a[i] и b[i] и они загружаются в регистры (строки 20, 22 — 27), со стека в регистр загружается значение переменной r (строка 28), к значению r прибавляется результат a[i] \* b[i] (строка 29), новое значение r записывается на стек (строка 30).

В строках 33 — 35 происходит увеличение счётчика цикла i после выполнения очередной итерации: со стека в регистр загружается значение переменной i (строка 33), значение переменной i увеличивается на 1 (строка 34), новое значение переменной i записывается на стек (строка 35).

В строках 38 — 42 происходит возврат результата после выполнения цикла: со стека в регистр, через который возвращается результат, загружается значение переменной r (строка 38), со стека в регистры загружаются адрес возврата и адрес предыдущего кадра стека (строки 39 — 40), очищается кадр стека (строка 41), происходит возврат из функции dot\_product (строка 42).

**Компиляция с флагом -O1**

dot\_product: # @dot\_product  
# %bb.0:  
 fmv.w.x fa0, zero  
 beqz a2, .LBB0\_2  
.LBB0\_1: # =>This Inner Loop Header: Depth=1  
 flw ft0, 0(a0)  
 flw ft1, 0(a1)  
 fmadd.s fa0, ft0, ft1, fa0  
 addi a2, a2, -1  
 addi a1, a1, 4  
 addi a0, a0, 4  
 bnez a2, .LBB0\_1  
.LBB0\_2:  
 ret

Полученный ассемблерный код значительно короче, полученного без применения оптимизаций. Это получается за счёт того, что на стеке не выделяется место под аргументы и локальные переменные, и все вычисления производятся с регистрами без обращений к памяти.

В строке 3 происходит инициализация регистра, в котором хранится значение r, нулём.

В строке 4 происходит сравнение регистра, в котором хранится значение n, с нулём, чтобы начать выполнение цикла.

В строках 6 — 7 происходит загрузка в регистры значений a[0] и b[0].

В строке 8 к значению r прибавляется результат a[0] \* b[0].

В строке 9 значение n уменьшается на 1.

В строках 10 — 11 увеличиваются значения регистров, в которых хранятся адреса массивов a и b, чтобы на следующей итерации a[0] и b[0] соответствовали следующим элементам массивов.

В строке 12 происходит сравнение регистра, в котором хранится значение n, с нулём, чтобы узнать, нужно ли выполнять очередную итерацию цикла.

В строке 14 происходит возврат из функции dot\_product.

Таким образом, данный код работает аналогично неоптимизированному, однако выполняет гораздо меньше «дорогих» обращений к памяти.

**Компиляция с флагом -O2**

dot\_product: # @dot\_product  
# %bb.0:  
 beqz a2, .LBB0\_4  
# %bb.1:  
 li a3, 8  
 andi a6, a2, 7  
 bgeu a2, a3, .LBB0\_5  
# %bb.2:  
 fmv.w.x fa0, zero  
 li a2, 0  
 bnez a6, .LBB0\_8  
.LBB0\_3:  
 ret  
.LBB0\_4:  
 fmv.w.x fa0, zero  
 ret  
.LBB0\_5:  
 andi a2, a2, -8  
 fmv.w.x fa0, zero  
 li a4, 0  
 neg a2, a2  
 addi a5, a1, 16  
 addi a3, a0, 16  
.LBB0\_6: # =>This Inner Loop Header: Depth=1  
 flw ft0, -16(a3)  
 addi a4, a4, -8  
 flw ft1, -16(a5)  
 flw ft2, -12(a5)  
 fmadd.s ft0, ft0, ft1, fa0  
 flw ft1, -12(a3)  
 fmadd.s ft0, ft1, ft2, ft0  
 flw ft1, -8(a3)  
 flw ft2, -8(a5)  
 fmadd.s ft0, ft1, ft2, ft0  
 flw ft1, -4(a3)  
 flw ft2, -4(a5)  
 fmadd.s ft0, ft1, ft2, ft0  
 flw ft1, 0(a3)  
 flw ft2, 0(a5)  
 fmadd.s ft0, ft1, ft2, ft0  
 flw ft1, 4(a3)  
 flw ft2, 4(a5)  
 fmadd.s ft0, ft1, ft2, ft0  
 flw ft1, 8(a3)  
 flw ft2, 8(a5)  
 fmadd.s ft0, ft1, ft2, ft0  
 flw ft1, 12(a3)  
 flw ft2, 12(a5)  
 addi a5, a5, 32  
 addi a3, a3, 32  
 fmadd.s fa0, ft1, ft2, ft0  
 bne a2, a4, .LBB0\_6  
# %bb.7:  
 neg a2, a4  
 beqz a6, .LBB0\_3  
.LBB0\_8:  
 slli a2, a2, 2  
 add a3, a0, a2  
 flw ft0, 0(a3)  
 add a3, a1, a2  
 flw ft1, 0(a3)  
 li a3, 1  
 fmadd.s fa0, ft0, ft1, fa0  
 beq a6, a3, .LBB0\_3  
# %bb.9:  
 addi a3, a2, 4  
 add a4, a0, a3  
 add a3, a3, a1  
 flw ft1, 0(a3)  
 li a3, 2  
 flw ft0, 0(a4)  
 fmadd.s fa0, ft0, ft1, fa0  
 beq a6, a3, .LBB0\_3  
# %bb.10:  
 addi a3, a2, 8  
 add a4, a0, a3  
 add a3, a3, a1  
 flw ft1, 0(a3)  
 li a3, 3  
 flw ft0, 0(a4)  
 fmadd.s fa0, ft0, ft1, fa0  
 beq a6, a3, .LBB0\_3  
# %bb.11:  
 addi a3, a2, 12  
 add a4, a0, a3  
 add a3, a3, a1  
 flw ft1, 0(a3)  
 li a3, 4  
 flw ft0, 0(a4)  
 fmadd.s fa0, ft0, ft1, fa0  
 beq a6, a3, .LBB0\_3  
# %bb.12:  
 addi a3, a2, 16  
 add a4, a0, a3  
 add a3, a3, a1  
 flw ft1, 0(a3)  
 li a3, 5  
 flw ft0, 0(a4)  
 fmadd.s fa0, ft0, ft1, fa0  
 beq a6, a3, .LBB0\_3  
# %bb.13:  
 addi a3, a2, 20  
 add a4, a0, a3  
 add a3, a3, a1  
 flw ft1, 0(a3)  
 li a3, 6  
 flw ft0, 0(a4)  
 fmadd.s fa0, ft0, ft1, fa0  
 beq a6, a3, .LBB0\_3  
# %bb.14:  
 addi a2, a2, 24  
 add a0, a0, a2  
 flw ft0, 0(a0)  
 add a0, a1, a2  
 flw ft1, 0(a0)  
 fmadd.s fa0, ft0, ft1, fa0  
 ret

В данном случае порождается гораздо больше кода, чем при уровне оптимизаций -O1.

Сначала проверяется случай n == 0 (строка 3), если это так, регистр, через который происходит возврат результата, инициализируется нулём и происходит возврат из функции dot\_product (строки 15 — 16).

Затем проверяется случай, когда n >= 8 (строки 5 и 7), если это не так (n < 8), происходит переход к строкам 57 — 117. Эти строки содержат 7 блоков, в каждом происходит вычисление r += a[i] \* b[i] и проверяется, нужно ли закончить и вернуть результат.

Если же n >= 8, то вычисления производятся блоками по 8 операций (строки 25 — 51):

r += a[i + 0] \* b[i + 0]; r += a[i + 1] \* b[i + 1]; ... r += a[i + 7] \* b[i + 7];

То есть компилятор произвёл «раскрутку цикла». Важно отметить, что в таком случае проверять, нужно ли остановиться, достаточно один раз на весь блок (строка 52), а не на каждую операцию r += a[i] \* b[i]. Это положительно сказывается на производительности, так как условные переходы — «дорогая» операция. В момент, когда осталось выполнить меньше 8 операций, проверяется, выполнены ли все вычисления (строки 54 — 55), если да, происходит возврат из функции dot\_product (строка 13), иначе — происходит переход к случаю, когда надо вычислить менее 8 операций (строки 57 — 117).

Поскольку блоки вычислений в строках 25 — 51 однотипны и данные лежат в памяти упорядоченно, вычисления могут быть векторизованы. Однако на момент создания этой лабораторной работы версия компилятора в *Syntacore Kit* не векторизует вычисления с числами с плавающей запятой.

Немного изменим пример, чтобы продемонстрировать, как компилятор оптимизирует код с помощью векторных инструкций.

Измените следующие строки в main.c:

* 10 — typedef int cell\_t;
* 43 — printf("%i\n%.3g\n"

Чтобы получить векторные инструкции, необходимо указать векторное расширение в архитектуре, передав опцию -march=rv64gcv. Таким образом, получаем следующую команду:

clang -target riscv64-unknown-linux-gnu --sysroot="$GCC\_ROOT/sysroot" --gcc-toolchain="$GCC\_ROOT" -S -o main-O3v.s main.c -march=rv64gcv -O3

**CLang с опциями -O3 и -march=rv64gcv**

dot\_product: # @dot\_product  
# %bb.0:  
 beqz a2, .LBB0\_3  
# %bb.1:  
 csrr a3, vlenb  
 srli t0, a3, 1  
 bgeu a2, t0, .LBB0\_4  
# %bb.2:  
 li a7, 0  
 li a3, 0  
 j .LBB0\_7  
.LBB0\_3:  
 li a0, 0  
 ret  
.LBB0\_4:  
 addi a4, t0, -1  
 slli t1, a3, 1  
 and a6, a2, a4  
 add t2, a0, a3  
 add t3, a1, a3  
 vsetvli a3, zero, e32, m1, ta, ma  
 sub a7, a2, a6  
 li a5, 0  
 vmv.v.i v8, 0  
 mv a3, a7  
 vmv.v.i v9, 0  
.LBB0\_5: # =>This Inner Loop Header: Depth=1  
 add a4, a0, a5  
 add t4, t2, a5  
 vl1re32.v v10, (a4)  
 add a4, a1, a5  
 vl1re32.v v11, (a4)  
 add a4, t3, a5  
 vl1re32.v v12, (t4)  
 vl1re32.v v13, (a4)  
 sub a3, a3, t0  
 add a5, a5, t1  
 vmacc.vv v8, v11, v10  
 vmacc.vv v9, v13, v12  
 bnez a3, .LBB0\_5  
# %bb.6:  
 vadd.vv v8, v9, v8  
 vmv.s.x v9, zero  
 vredsum.vs v8, v8, v9  
 vmv.x.s a3, v8  
 beqz a6, .LBB0\_9  
.LBB0\_7:  
 slli a4, a7, 2  
 sub a2, a2, a7  
 add a1, a1, a4  
 add a0, a0, a4  
.LBB0\_8: # =>This Inner Loop Header: Depth=1  
 lw a4, 0(a0)  
 addi a2, a2, -1  
 lw a5, 0(a1)  
 addi a1, a1, 4  
 addi a0, a0, 4  
 mulw a4, a5, a4  
 addw a3, a3, a4  
 bnez a2, .LBB0\_8  
.LBB0\_9:  
 mv a0, a3  
 ret

В данном ассемблерном коде инструкции и регистры, начинающиеся с буквы v относятся к [векторному расширению (V) RISC‑V](https://github.com/riscv/riscv-v-spec/tree/master).

Сначала на основании соотношения длины векторных регистров в байтах и величины n принимается решение об использовании векторов (строки 5 — 7).

Если векторы не используются, то вычисления производятся с помощью обычного цикла (строки 53 — 60).

Если надо использовать векторы, то в 21 строке выставляется максимальная длина для используемых векторов, а в строках 24 и 26 инициализируются векторные регистры, в которых будет аккумулироваться вычисляемые значения. Далее в цикле происходят вычисления с использованием векторов (строки 28 — 40). После окончания цикла накопленные в векторных регистрах результаты складываются и записываются в обычный регистр (строки 42 — 45). Если ещё остались необработанные элементы исходных массивов, то они обрабатываются обычным циклом (строки 48 — 60).

Попробуйте скомпилировать программу теми же опциями (-O3 и -march=rv64gcv), используя систему компиляции GCC.

**GCC с опциями -O3 и -march=rv64gcv**

dot\_product:  
 beq a2,zero,.L4  
 slli a2,a2,2  
 mv a5,a0  
 add a2,a0,a2  
 li a0,0  
.L3:  
 lw a3,0(a5)  
 lw a4,0(a1)  
 addi a5,a5,4  
 addi a1,a1,4  
 mulw a4,a4,a3  
 addw a0,a4,a0  
 bne a2,a5,.L3  
 ret  
.L4:  
 li a0,0  
 ret

GCC не породил векторизованный код. Различные системы компиляции имеют неодинаковую степень поддержки различных расширений RISC‑V. Обратите на это внимание при выборе системы компиляции для своих проектов.