Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчёт по практикуму №1 по курсу «Архитектура ЭВМ»

Тема Разработка и отладка программ в вычислительном комплексе

Тераграф с помощью библиотеки leonhard x64 xrt

Студент Княжев А. В.

Группа ИУ7-52Б

Преподаватель Дубровин Е.Н.

Москва — 2022 г.

Оглавление

Введение

Практикум посвящен освоению принципов работы вычислительного комплекса Тераграф и получению практических навыков решения задач обработки множеств на основе гетерогенной вычислительной структуры. В ходе практикума необходимо ознакомиться с типовой структурой двух взаимодействующих программ: хост-подсистемы и программного ядра sw_kernel. Участникам предоставляется доступ к удаленному серверу с ускорительной картой и настроенными средствами сборки проектов, конфигурационный файл для двухъядерной версии микропроцессора Леонард Эйлер, а также библиотека leonhard x64 xrt с открытым исходным кодом.

1 Аналитическая часть

1.1 Структура вычислительного комплекса Тераграф

Комплекс «Тераграф» предназначен для хранения и обработки графов сверхбольшой размерности и будет применяться для моделирования биологических систем, анализа финансовых потоков в режиме реального времени, для хранения знаний в системах искусственного интеллекта, создания интеллектуальных автопилотов с функциями анализа дорожной обстановки, и в других прикладных задачах. Он способен обрабатывать графы сверхбольшой размерности до 1012 (одного триллиона) вершин и 2·1012 ребер. Комплекс состоит из 3-х однотипных гетерогенных узлов, которые взаимодействуют между собой через высокоскоростные сетевые подключения 100Gb Ethernet. Каждый узел состоит из хост-подсистемы, подсистемы хранения графов, подсистемы коммутации узлов, а также подсистемы обработки графов. Структурная схема одного узла представлена на рисунке (1.1).

1.2 Принципы взаимодействия микропроцессора Леонард Эйлер и хост-подсистемы

Основу взаимодействия подсистем при обработке графов составляет передача блоков данных и коротких сообщений между GPC и хост-подсистемой. Для передачи сообщений для каждого GPC реализованы два аппаратных FIFO буфера на 512 записей: Host2GPC для передачи от хост-подсистемы к ядру, и GPC2Host для передачи в обратную сторону.

Обработка начинается с того, что собранное программное ядро (software kernel) загружается в локальное ОЗУ одного или нескольких CPE (микропроцессора riscv32im). Для этого используется механизм прямого доступа к памяти со стороны хост-подсистемы. В свою очередь, GPC (один или несколько) получают сигнал о готовности образа software kernel в Глобальной памяти, после чего вызывается загрузчик, хранимый в ПЗУ CPE. Загрузчик выполняет копирование программного ядра из Глобальной памяти в ОЗУ CPE и передает управление на начальный адрес программы обработки. Предусмотрен режим работы GPC, при котором во время обработки

происходит обмен данными и сообщениями. Эти два варианта работы реализуется через буферы и очереди соответственно.

Если код программного ядра уже загружен в ОЗУ CPE, хост-подсистема может вызвать любой из содержащихся в нем обработчиков. Для этого в GPC передает оговоренный UID обработчика (handler), после чего передается сигнал запуска (сигнал START). В ответ CPE устанавливает состояние BUSY и начинает саму обработку. В ходе обработки ядро может обмениваться сообщениями с хост-подсистемой через очереди (команды mq_send и mq_receive). По завершении обработки устанавливается состояние IDLE и вырабатывается прерывание, которое перехватывается хост-подсистемой. Далее, пользовательское приложение хост-подсистемы уведомляется о завершении обработки и готовности результатов.

Если во время работы над кодом обработчика программному ядру software kernel требуется осуществить передачу больших блоков данным между CPE и хост-подсистемой, то может быть задействована Глобальная память и внешняя память большого размера (External Memory, до 16ГБ).

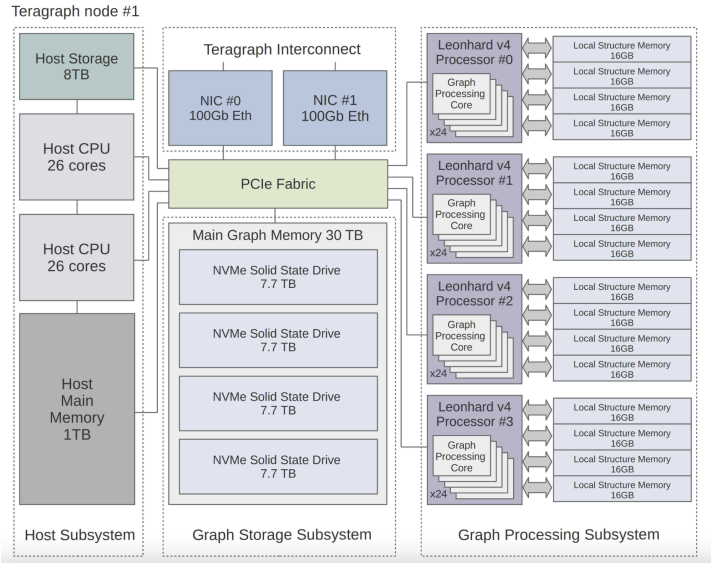

Рисунок 1.1 – Схема одного узла Телеграф

1.3 Взаимодействие CPE(riscv32im) и SPE(lnh64)

Микропроцессор lnh64 с набором команд дискретной математики (Discrete Mathematics Instruction Set Computer) является ассоциативным процессором, т.е. устройством, выполняющим операции обработки над данными, хранящимися в ассоциативной памяти (так называемой Локальной памя-

ти структур). В качестве таковой выступает адресная память DDR4, причем для каждого ядра lnh64 доступны 2.5 ГБ адресного пространства в ней. Для организации ассоциативного способа доступа к адресному устройству микропроцессор lnh64 организует на аппаратном уровне структуру B+дерева. Причем 512МБ занимает древовидая структура от верхнего и до предпоследнего уровня, 2048МБ занимает последний уровень дерева, на котором и хранятся 64х разрядные ключи и значения. Каждый микропроцессор lnh64 может хранить и обрабатывать до 117 миллионов ключей и значений.

Исходя из этого, обработка множеств или графов представляется в DISC наборе команд, как работа со структурами ключей и значений (key-value). Однако, как было показано ранее при описании набора команд DISC, в отличие от общепринятых key-value хранилищ, доступны такие операции как ближайший больший (NGR), ближайший меньший (NSM), команды объединения множеств (OR) и ряд других. Это и позволяет использовать lnh64 в качестве устройства, хранящего большие множества (для графов это множества вершин и ребер).

Доступ к микропроцессору lnh64 (Structure Processing Element) осуществляется чтением и записью в пространство памяти микропроцессора riscv32im (Computing Processing Element) в диапазоне 0x60000000 - 0x60001000.

2 Ход выполнения работы

Все задания практикума выполнялись по варианту 11.

Устройство формирования индексов SQL EXCEPT. Сформировать в хост-подсистеме и передать в SPE 256 записей множества A (случайные числа в диапазое 0..1024) и 256 записей множества B (случайные числа в диапазоне 0..1024). Сформировать в SPE множество C = A not B. Выполнить тестирование работы SPE, сравнив набор ключей в множестве C с ожидаемым.

В листинге 2.1 представлен код программы по индивидуальному варианту из файла *host_main.cpp*. В листинге 2.2 представлен код программы по индивидуальному варианту из файла *sw_kernel_main.c*.

Листинг 2.1 – Код программы по индивидуальному варианту

host_main.cpp

```
#include <iostream>
#include <stdio.h>
#include <stdexcept>
#include <iomanip>
#ifdef _WINDOWS
#include <io.h>
#else
#include <unistd.h>
#endif


#include "experimental/xrt_device.h"
#include "experimental/xrt_kernel.h"
#include "experimental/xrt_bo.h"
#include "experimental/xrt_ini.h"

#include "gpc_defs.h"
#include "leonhardx64_xrt.h"
#include "gpc_handlers.h"

#define BURST 256

union uint64 {
    uint64_t    u64;
    uint32_t    u32[2];
    uint16_t    u16[4];
    uint8_t     u8[8];
};

```

```
uint64_t rand64() {
    uint64 tmp;
    tmp.u32[0] =  rand();
    tmp.u32[1] =  rand();
    return tmp.u64;
}

static void usage()
{
    std::cout << "usage: <xclbin> <sw_kernel>\n\n";
}

int main(int argc, char** argv)
{

    unsigned int cores_count = 0;
    float LNH_CLOCKS_PER_SEC;

    __foreach_core(group, core) cores_count++;

    if (argc < 3) {
        usage();
        throw std::runtime_error("FAILED_TEST\nNo xclbin specified");
    }

    leonhardx64 lnh_inst = leonhardx64(0,argv[1]);
    __foreach_core(group, core)
    {
        lnh_inst.load_sw_kernel(argv[2], group, core);
    }

    uint64_t *host2gpc_buffer[LNH_GROUPS_COUNT][LNH_MAX_CORES_IN_GROUP];
    __foreach_core(group, core)
    {
        host2gpc_buffer[group][core] = (uint64_t*)
           malloc(2*BURST*sizeof(uint64_t));
    }
    uint64_t *gpc2host_buffer[LNH_GROUPS_COUNT][LNH_MAX_CORES_IN_GROUP];
    __foreach_core(group, core)
    {
        gpc2host_buffer[group][core] = (uint64_t*)
           malloc(2*BURST*sizeof(uint64_t));
    }


    __foreach_core(group, core)
    {
        for (int i=0;i<BURST;i++) {
```

```
            uint64_t a = rand64()%1025;
            uint64_t b = rand64() %1025;
            host2gpc_buffer[group][core][2*i] = a;
            host2gpc_buffer[group][core][2*i+1] = b;

        }
    }

    __foreach_core(group, core) {
        lnh_inst.gpc[group][core]->start_async(__event__(insert_burst));
    }

    __foreach_core(group, core) {
        lnh_inst.gpc[group][core]->buf_write(BURST*2*sizeof(uint64_t),(char*)host2
    }

    __foreach_core(group, core) {
        lnh_inst.gpc[group][core]->buf_write_join();
    }

    __foreach_core(group, core) {
        lnh_inst.gpc[group][core]->mq_send(BURST);
    }

    __foreach_core(group, core) {
        lnh_inst.gpc[group][core]->start_async(__event__(search_burst));
    }

    unsigned int count[LNH_GROUPS_COUNT][LNH_MAX_CORES_IN_GROUP];

    __foreach_core(group, core) {
        count[group][core] = lnh_inst.gpc[group][core]->mq_receive();
    }



    __foreach_core(group, core) {
        lnh_inst.gpc[group][core]->buf_read(count[group][core]*2*sizeof(uint64_t)
    }


    __foreach_core(group, core) {
        lnh_inst.gpc[group][core]->buf_read_join();
    }


    bool error = false;

```

```
    __foreach_core(group, core) {
        for(int j = 0; j < 2 * BURST;j++)
        {
            uint64_t src = host2gpc_buffer[group][core][j];
            int c = 0;

            for (int i=0; i<count[group][core]; i++) {
                uint64_t dst = gpc2host_buffer[group][core][i];
                if (dst == src) {
                    c++;
                }
            }

            if (c < 1) {
                error = true;
            }
        }
    }

    if (!error)
        printf("Tests passed!\n");
    else
        printf("Tests failed!\n");

    __foreach_core(group, core) {
        free(host2gpc_buffer[group][core]);
        free(gpc2host_buffer[group][core]);
    }


    return 0;
}
```

Листинг 2.2 – Код программы по индивидуальному варианту

sw_kernel_main.c

```

#include <stdlib.h>
#include <unistd.h>
#include "lnh64.h"
#include "gpc_io_swk.h"
#include "gpc_handlers.h"

#define SW_KERNEL_VERSION 26
#define DEFINE_LNH_DRIVER
#define DEFINE_MQ_R2L
#define DEFINE_MQ_L2R
#define __fast_recall__
```

```
13
14 #define A_STRUCTURE 1
15 #define B_STRUCTURE 2
16 #define R_STRUCTURE 3
17
18 extern lnh lnh_core;
19 extern global_memory_io gmio;
20 volatile unsigned int event_source;
21
22 int main(void) {
23     //Leonhard driver structure should be initialised
24     lnh_init();
25     //Initialise host2gpc and gpc2host queues
26     gmio_init(lnh_core.partition.data_partition);
27     for (;;) {
28         //Wait for event
29         while (!gpc_start());
30         //Enable RW operations
31         set_gpc_state(BUSY);
32         //Wait for event
33         event_source = gpc_config();
34         switch(event_source) {
35             case __event__(insert_burst) : insert_burst(); break;
36             case __event__(search_burst) : search_burst(); break;
37         }
38         //Disable RW operations
39         set_gpc_state(IDLE);
40         while (gpc_start());
41
42     }
43 }
44
45 void insert_burst() {
46     lnh_del_str_sync(A_STRUCTURE);
47     lnh_del_str_sync(B_STRUCTURE);
48     lnh_del_str_sync(R_STRUCTURE);
49
50     unsigned int count = mq_receive();
51     unsigned int size_in_bytes = 2*count*sizeof(uint64_t);
52
53     uint64_t *buffer = (uint64_t*)malloc(size_in_bytes);
54
55     buf_read(size_in_bytes, (char*)buffer);
56
57     for (int i=0; i<count; i++) {
58         lnh_ins_sync(A_STRUCTURE,buffer[2*i],0);
59         lnh_ins_sync(B_STRUCTURE, buffer[2*i+1],0);
60     }
```

```
61      lnh_sync();
62      free(buffer);
63  }
64
65  void search_burst() {
66      lnh_sync();
67
68      lnh_or_sync(A_STRUCTURE, B_STRUCTURE, R_STRUCTURE);
69      //
70      unsigned int count = lnh_get_num(R_STRUCTURE);
71      unsigned int size_in_bytes = count*sizeof(uint64_t);
72
73      uint64_t *R = (uint64_t*)malloc(size_in_bytes);
74
75      lnh_get_first(R_STRUCTURE);
76      for (int i=0; i<count; i++) {
77          R[i] = lnh_core.result.key;
78          lnh_next(R_STRUCTURE,lnh_core.result.key);
79      }
80
81      buf_write(size_in_bytes, (char*)R);
82      mq_send(count);
83      free(R);
84  }
```

В листинге 2.3 представлена команда для сборки проекта.

Листинг 2.3 – Код программы по индивидуальному варианту sw_kernel_main.c

```
1  ./host_main leonhard_2cores_267mhz.xclbin sw_kernel_main.rawbinary
```

Заключение

В данном практикуме были получены практические навыки решения задач обработки множеств на основе гетерогенной вычислительной структуры. В ходе практикума было проведено знакомство с типовой структурой двух взаимодействующих программ: хост-подсистемы и программного ядра sw_kernel.