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»

ОТЧЕТ
по практикуму
Задание №1

Тема практикума «Обработка и визуализация графов.»

Название «Разработка и отладка программ в вычислительном комплексе Тераграф с помощью библиотеки leonhard x64 xrt»

Дисциплина «Архитектура элекронно-вычислительных» машин

Студент: _____ Мансуров В. М.
подпись, дата Фамилия, И.О.

Преподаватель: _____ Ибрагимов С. В.
подпись, дата Фамилия, И. О.

Москва — 2022 г.

Содержание

Цель работы

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 Основные теоретические сведения

Основная вычислительная системы (так называемая хост-подсистема) берет на себя функции управления запуском вычислительных задач, поддержкой сетевых подключений, обработкой и балансировкой нагрузки. В хост-подсистему входят два многоядерных ЦПУ по 26 ядер каждый, оперативная память на 1 Тбайт и дополнительная энергонезависимая память на 8 Тбайт, где хранятся атрибуты вершин и ребер графа, буферизируются поступающие запросы на обработку и визуализацию графов, хранятся временные данные об изменениях в графах. В хост-подсистеме используется процессор с архитектурой x86 для обеспечения сетевого взаимодействия и связи системы с внешним миром. Указанные функции реализованы в Программном ядре хост-подсистемы (host software kernel) – программном обеспечении, взаимодействующим с подсистемой обработки графов через шину PCIe.

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. На рисунке 7 представлена диаграмма последовательностей первого сценария работы – вызов обработчика с передачей параметров и возвратом значения через очередь сообщений.

2 Экспериментальная часть

2.1 Индивидуальное задание

Задание практикума выполнялось 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.

2.2 Результаты выполнения задания

2.2.1 Host

Листинг 2.1 – Измененный код хост-системы под индивидульное задание

```
#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"
```

```
20
21 #define BURST 256
22
23 union uint64 {
24     uint64_t    u64;
25     uint32_t    u32[2];
26     uint16_t    u16[4];
27     uint8_t     u8[8];
28 };
29
30 uint64_t rand64() {
31     uint64 tmp;
32     tmp.u32[0] =  rand();
33     tmp.u32[1] =  rand();
34     return tmp.u64;
35 }
36
37 static void usage()
38 {
39     std::cout << "usage: <xclbin> <sw_kernel>\n\n";
40 }
41
42 #include <set>
43 using namespace std;
44
45
46 set<uint64_t> getExcept(set<uint64_t> &a, set<uint64_t> &b){
47     set<uint64_t> result;
48     printf("Ожидаемый результат C:\n");
49     for (uint64_t el:a){
50         if (b.find(el) == b.end()) {
51             printf("\t%u\n", el);
52             result.insert(el);
53         }
54     }
55
56     return result;
57 }
58
59 int main(int argc, char** argv)
60 {
```

```
61      set<uint64_t> a,b,c;
62      unsigned int cores_count = 0;
63      float LNH_CLOCKS_PER_SEC;
64
65      __foreach_core(group, core) cores_count++;
66
67      //Assign xclbin
68      if (argc < 3) {
69          usage();
70          throw std::runtime_error("FAILED_TEST\nNo␣xclbin␣
                specified");
71      }
72
73      //Open device #0
74      leonhardx64 lnh_inst = leonhardx64(0,argv[1]);
75      __foreach_core(group, core)
76      {
77          lnh_inst.load_sw_kernel(argv[2], group, core);
78      }
79
80
81      // /*
82      //  *
83      //  * Запись множества из BURST key-value и его последовательн
            ое чтение через Global Memory Buffer
84      //  *
85      //  */
86
87
88      //Выделение памяти под буферы gpc2host и host2gpc для каждого
            ядра и группы
89      uint64_t
            *host2gpc_buffer[LNH_GROUPS_COUNT][LNH_MAX_CORES_IN_GROUP];
90      __foreach_core(group, core)
91      {
92          host2gpc_buffer[group][core] = (uint64_t*)
                malloc(4*BURST*sizeof(uint64_t));
93      }
94      uint64_t
            *gpc2host_buffer[LNH_GROUPS_COUNT][LNH_MAX_CORES_IN_GROUP];
95      __foreach_core(group, core)
```

```
96      {
97          gpc2host_buffer[group][core] = (uint64_t*)
               malloc(2*BURST*sizeof(uint64_t));
98      }
99
100     //Создание массива ключей и значений для записи в lnh64
101     __foreach_core(group, core)
102     {
103         printf("Generate data\n");
104         printf("\tIndex — Value\n");
105         uint64_t random_num;
106         for (int i=0;i<2*BURST;i++) {
107             //Первый элемент массива uint64_t - key
108             random_num = rand64() % 1024;
109
110             if (i < BURST){
111                 host2gpc_buffer[group][core][2*i+1] = i;
112                 //random_num = i + 250;
113                 a.insert(random_num);
114             } else {
115                 host2gpc_buffer[group][core][2*i+1] = i - BURST;
116                 //random_num = i;
117                 b.insert(random_num);
118             }
119
120             if (i == 0)
121             printf("----------A SET----------\n");
122             if (i == BURST)
123             printf("----------B SET----------\n");
124             host2gpc_buffer[group][core][2*i] = random_num;
125             //Второй uint64_t - value
126
127             printf("\t%u, %u\n",
                   host2gpc_buffer[group][core][2*i+1],
                   host2gpc_buffer[group][core][2*i]);
128         }
129     }
130
131     //Запуск обработчика insert_burst
132     __foreach_core(group, core) {
133         lnh_inst.gpc[group][core]->start_async(__event__(insert_burst));
```

```
134     }
135
136     //DMA запись массива host2gpc_buffer в глобальную память
137     __foreach_core(group, core) {
138         lnh_inst.gpc[group][core]->buf_write(BURS *4 *
                sizeof(uint64_t), (char*)
                host2gpc_buffer[group][core]);
139     }
140
141     //Ожидание завершения DMA
142     __foreach_core(group, core) {
143         lnh_inst.gpc[group][core]->buf_write_join();
144     }
145
146     //Передать количество key-value
147     __foreach_core(group, core) {
148         lnh_inst.gpc[group][core]->mq_send(BURST);
149     }
150
151     //Запуск обработчика для последовательного обхода множества кл
            ючей
152     __foreach_core(group, core) {
153         lnh_inst.gpc[group][core]->start_async(__event__(search_burst));
154     }
155
156     //Получить количество ключей
157     unsigned int count[LNH_GROUPS_COUNT][LNH_MAX_CORES_IN_GROUP];
158
159     __foreach_core(group, core) {
160         count[group][core] =
                lnh_inst.gpc[group][core]->mq_receive();
161     }
162
163
164     //Прочитать количество ключей
165     __foreach_core(group, core) {
166         lnh_inst.gpc[group][core]->buf_read(count[group][core] *
                2 * sizeof(uint64_t),
                (char*)gpc2host_buffer[group][core]);
167     }
168
```

```
169     //Ожидание завершения DMA
170     __foreach_core(group, core) {
171         lnh_inst.gpc[group][core]->buf_read_join();
172     }
173
174
175     bool error = false;
176     set<uint64_t> res;
177     //Проверка целостности данных
178     __foreach_core(group, core) {
179         printf("Count: %u\n", count[group][core]);
180         printf("Результат полученный из SPE:\n");
181         printf("\tIndex — Value\n");
182         for (int i=0; i<count[group][core]; i++) {
183             uint64_t key = gpc2host_buffer[group][core][2*i];
184             uint64_t value = gpc2host_buffer[group][core][2*i+1];
185             res.insert(key);
186             //uint64_t orig_key =
                    host2gpc_buffer[group][core][2*value];
187             printf("\tC = %u, %u\n", value, key);
188             //          if (key != orig_key) {
189                 //              error = true;
190                 //          }
191         }
192
193         c = getExcept(a, b);
194         if (res != c)
195         error = true;
196     }
197
198
199     __foreach_core(group, core) {
200         free(host2gpc_buffer[group][core]);
201         free(gpc2host_buffer[group][core]);
202     }
203
204     if (!error)
205     printf("Тест пройден успешно!\n");
206     else
207     printf("Тест завершен с ошибкой!\n");
208
```

```
209
210     return 0;
211 }
```

2.2.2 sw_kernel

Листинг 2.2 – Измененный код sw_kernel под индивидульное задание

```
1  /*
2  * gpc_test.c
3  *
4  * sw_kernel library
5  *
6  *  Created on: April 23, 2021
7  *      Author: A.Popov
8  */
9
10 #include <stdlib.h>
11 #include <unistd.h>
12 #include "lnh64.h"
13 #include "gpc_io_swk.h"
14 #include "gpc_handlers.h"
15
16 #define SW_KERNEL_VERSION 26
17 #define DEFINE_LNH_DRIVER
18 #define DEFINE_MQ_R2L
19 #define DEFINE_MQ_L2R
20 #define __fast_recall__
21
22 #define TEST_STRUCTURE 1
23 #define A 1
24 #define B 2
25 #define C 3
26
27 extern lnh lnh_core;
28 extern global_memory_io gmio;
29 volatile unsigned int event_source;
30
31 int main(void) {
32     ////////////////////////////////////////////////////
```

```
33      //                      Main Event Loop
34      ////////////////////////////////////////////////////////
35      //Leonhard driver structure should be initialised
36      lnh_init();
37      //Initialise host2gpc and gpc2host queues
38      gmio_init(lnh_core.partition.data_partition);
39      for (;;) {
40          //Wait for event
41          while (!gpc_start());
42          //Enable RW operations
43          set_gpc_state(BUSY);
44          //Wait for event
45          event_source = gpc_config();
46          switch(event_source) {
47              ////////////////////////////////////////////
48              //  Measure GPN operation frequency
49              ////////////////////////////////////////////
50              case __event__(insert_burst) : insert_burst(); break;
51              case __event__(search_burst) : search_burst(); break;
52          }
53          //Disable RW operations
54          set_gpc_state(IDLE);
55          while (gpc_start());
56
57      }
58  }
59
60  //-------------------------------------------------------------------
61  //      Получить пакет из глобальной памяти и аписат в lnh64
62  //-------------------------------------------------------------------
63
64  void insert_burst() {
65
66      //Удаление данных из структур
67      lnh_del_str_sync(1);
68      lnh_del_str_sync(2);
69      lnh_del_str_sync(3);
70      //Объявление переменных
71      unsigned int count = mq_receive();
72      unsigned int size_in_bytes = 4*count*sizeof(uint64_t);
73      //Создание буфера для приема пакета
```

```
74      uint64_t *buffer = (uint64_t*)malloc(size_in_bytes);
75      //Чтение пакета в RAM
76      buf_read(size_in_bytes, (char*)buffer);
77      //Обработка пакета – запись
78
79      // insert A set
80      int i=0;
81      for (; i<count; i++) {
82          lnh_ins_sync(1,buffer[2*i],buffer[2*i+1]);
83      }
84
85      // insert B set
86      for (; i<2*count; i++) {
87          lnh_ins_sync(2,buffer[2*i],buffer[2*i+1]);
88      }
89      lnh_sync();
90      free(buffer);
91  }
92
93
94  //-------------------------------------------------------------------
95  //      Обход структуры lnh64 и запись в глобальную память
96  //-------------------------------------------------------------------
97
98  void search_burst() {
99
100     //Ожидание завершения предыдущих команд
101     lnh_sync();
102     // C = A not B
103     lnh_not_sync(A, B, C);
104     //Объявление переменных
105     // count of C set
106     unsigned int count = lnh_get_num(C);
107     unsigned int size_in_bytes = 2*count*sizeof(uint64_t);
108     //Создание буфера для приема пакета
109     uint64_t *buffer = (uint64_t*)malloc(size_in_bytes);
110     //Выборка минимального ключа
111     lnh_get_first(C);
112     //Запись ключа и значения в буфер
113     for (int i=0; i<count; i++) {
114         buffer[2*i] = lnh_core.result.key;
```

```
115             buffer[2*i+1] = lnh_core.result.value;
116             lnh_next(C,lnh_core.result.key);
117         }
118         //Запись глобальной памяти из RAM
119         buf_write(size_in_bytes, (char*)buffer);
120         mq_send(count);
121         free(buffer);
122 
123 }
```