



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 Карпова Е.О.

Группа ИУ7-52Б

Преподаватель Дубровин Е.Н.

Москва — 2022 г.

Оглавление

Введение	3
1 Аналитическая часть	4
1.1 Структура вычислительного комплекса Тераграф	4
1.2 Принципы взаимодействия микропроцессора Леонард Эйлер и хост-подсистемы	4
1.3 Взаимодействие CPE(riscv32im) и SPE(lnh64)	5
2 Ход выполнения работы	7
Заключение	13

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 10^{12} (одного триллиона) вершин и $2 \cdot 10^{12}$ 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 данных между CPE и хост-подсистемой, то может быть задействована Глобальная память и внешняя память большого размера (External Memory, до 16ГБ).

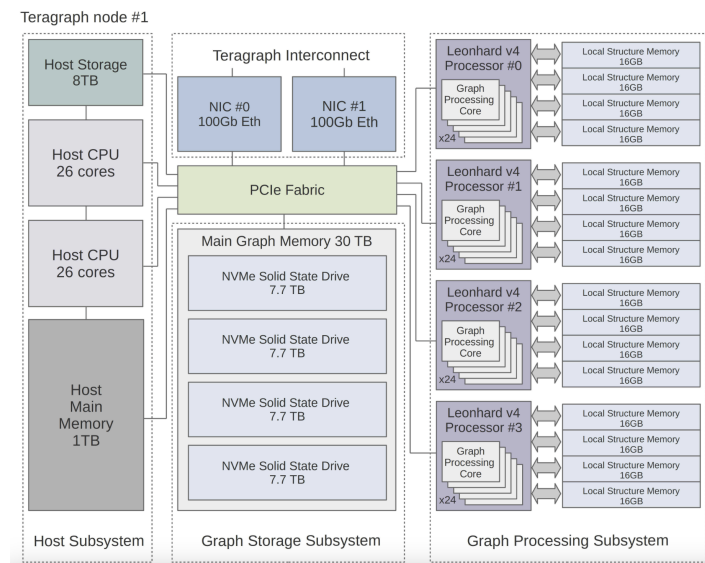


Рисунок 1.1 – Схема одного узла Телеграф

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 В+дерева. Причем 512МБ занимает древовидная 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 8.

Устройство вычисления обратной функции. Сформировать в хост-подсистеме и передать в SPE 256 записей key-value со значениями функции $f(x) = x^2$ в диапазоне значений x от 0 до 1048576 (где $f(x)$ - ключ, x - значение). Выполнить тестирование работы устройства, посылая из хост-подсистемы значение $f(x)$ и получая от *sw_kernel* значение x . Если указанного значения $f(x)$ не сохранено в SPE, выполнить поиск ближайшего (меньшего или большего) значения к $f(x)$ и вернуть соответствующий x . Сравнить результат 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

```
1 #include <iostream>
2 #include <stdio.h>
3 #include <stdexcept>
4 #include <iomanip>
5 #ifdef _WINDOWS
6 #include <io.h>
7 #else
8 #include <unistd.h>
9 #endif
10
11
12 #include "experimental/xrt_device.h"
13 #include "experimental/xrt_kernel.h"
14 #include "experimental/xrt_bo.h"
15 #include "experimental/xrt_ini.h"
16
17 #include "gpc_defs.h"
18 #include "leonhardx64_xrt.h"
19 #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 int main(int argc, char** argv)
43 {
44
45     unsigned int cores_count = 0;
46     float LNH_CLOCKS_PER_SEC;
47
48     __foreach_core(group, core) cores_count++;
49
50     //Assign xclbin
51     if (argc < 3) {
52         usage();
53         throw std::runtime_error("FAILED_TEST\nNo _xclbin_ specified");
54     }
55
56     //Open device 0
57     leonhardx64 lnh_inst = leonhardx64(0, argv[1]);
58     __foreach_core(group, core)
59     {
60         lnh_inst.load_sw_kernel(argv[2], group, core);
61     }
62
63     uint64_t *host2gpc_buffer[LNH_GROUPS_COUNT][LNH_MAX_CORES_IN_GROUP];
64     __foreach_core(group, core)
65     {
66         host2gpc_buffer[group][core] = (uint64_t*)
67             malloc(2*BURST*sizeof(uint64_t));
68     }
69     uint64_t *gpc2host_buffer[LNH_GROUPS_COUNT][LNH_MAX_CORES_IN_GROUP];
70     __foreach_core(group, core)
71     {
72         gpc2host_buffer[group][core] = (uint64_t*)
73             malloc(2*BURST*sizeof(uint64_t));
74     }

```



```

73
74 __foreach_core(group, core)
75 {
76     for (int i=0;i<BURST;i++) {
77         uint64_t key = rand64() % 1048577;
78         host2gpc_buffer[group][core][2*i] = key * key;
79         host2gpc_buffer[group][core][2*i+1] = key;
80     }
81 }
82
83 __foreach_core(group, core) {
84     lnh_inst.gpc[group][core]->start_async(__event__(insert_burst));
85 }
86
87 __foreach_core(group, core) {
88     lnh_inst.gpc[group][core]->buf_write(BURST*2*sizeof(uint64_t),
89     (char*)host2gpc_buffer[group][core]);
90 }
91
92 __foreach_core(group, core) {
93     lnh_inst.gpc[group][core]->buf_write_join();
94 }
95
96 __foreach_core(group, core) {
97     lnh_inst.gpc[group][core]->mq_send(BURST);
98 }
99
100 uint64_t* user_key = (uint64_t*)malloc(sizeof(uint64_t));
101 user_key[0] = 158404;
102
103 __foreach_core(group, core) {
104     lnh_inst.gpc[group][core]->buf_write(sizeof(uint64_t),
105     (char*)user_key);
106 }
107
108 __foreach_core(group, core) {
109     lnh_inst.gpc[group][core]->buf_write_join();
110 }
111
112 __foreach_core(group, core) {
113     lnh_inst.gpc[group][core]->start_async(__event__(search_burst));
114 }
115
116 unsigned int count[LNH_GROUPS_COUNT][LNH_MAX_CORES_IN_GROUP];
117
118 __foreach_core(group, core) {
119     count[group][core] = lnh_inst.gpc[group][core]->mq_receive();
120

```

```

121     }
122
123
124     __foreach_core(group, core) {
125         lnh_inst.gpc[group][core]->buf_read(count[group]
126         [core]*2*sizeof(uint64_t),
127         (char*)gpc2host_buffer[group][core]);
128     }
129
130     __foreach_core(group, core) {
131         lnh_inst.gpc[group][core]->buf_read_join();
132     }
133
134
135
136     bool error = false;
137
138     __foreach_core(group, core) {
139         uint64_t value = gpc2host_buffer[group][core][0];
140         printf("flag_%llu, val_%llu\n", count[group][core], value);
141     }
142
143
144     __foreach_core(group, core) {
145         free(host2gpc_buffer[group][core]);
146         free(gpc2host_buffer[group][core]);
147     }
148
149     if (!error)
150         printf("Тест пройден успешно!\n");
151     else
152         printf("Тест завершен с ошибкой!\n");
153
154
155     return 0;
156 }

```

Листинг 2.2 – Код программы по индивидуальному варианту
sw_kernel_main.c

```

1 #include <stdlib.h>
2 #include <unistd.h>
3 #include "lnh64.h"
4 #include "gpc_io_swk.h"
5 #include "gpc_handlers.h"
6
7 #define SW_KERNEL_VERSION 26
8 #define DEFINE_LNH_DRIVER

```

```

9  #define DEFINE_MQ_R2L
10 #define DEFINE_MQ_L2R
11 #define __fast_recall__
12
13 #define TEST_STRUCTURE 1
14
15 extern lnh lnh_core;
16 extern global_memory_io gmio;
17 volatile unsigned int event_source;
18
19 int main(void) {
20     //////////////////////////////////////
21     // Main Event Loop
22     //////////////////////////////////////
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             //////////////////////////////////////
36             // Measure GPN operation frequency
37             //////////////////////////////////////
38             case __event__(insert_burst) : insert_burst(); break;
39             case __event__(search_burst) : search_burst(); break;
40         }
41         //Disable RW operations
42         set_gpc_state(IDLE);
43         while (gpc_start());
44     }
45 }
46
47
48
49 void insert_burst() {
50     lnh_del_str_sync(TEST_STRUCTURE);
51
52     unsigned int count = mq_receive();
53     unsigned int size_in_bytes = 2*count*sizeof(uint64_t);
54
55     uint64_t *buffer = (uint64_t*)malloc(size_in_bytes);
56     //Чтение пакета в RAM

```

```

57     buf_read(size_in_bytes, (char*)buffer);
58
59     for (int i=0; i<count; i++) {
60         lnh_ins_sync(TEST_STRUCTURE, buffer[2*i], buffer[2*i+1]);
61     }
62     lnh_sync();
63     free(buffer);
64 }
65
66 void search_burst() {
67     lnh_sync();
68
69     unsigned int count = lnh_get_num(TEST_STRUCTURE);
70
71     uint64_t *buffer = (uint64_t*)malloc(sizeof(uint64_t));
72     buf_read(sizeof(uint64_t), (char*)buffer);
73     uint64_t key = buffer[0];
74
75     auto f = lnh_search(TEST_STRUCTURE, key);
76     if (!f)
77         f = lnh_ngr(TEST_STRUCTURE, key);
78     mq_send(f);
79
80     buffer[0] = lnh_core.result.value;
81     buf_write(sizeof(uint64_t), (char*)buffer);
82     lnh_sync();
83     free(buffer);
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`.