Assignment 3: GPU Tuning

(1) Write a program to print device details like serial number, UUID, Board number and PCI address etc.

Solution:

```
/*
 * Copyright (C) 2020-2021 Intel Corporation
 *
 * SPDX-License-Identifier: MIT
 *
 */

#include <level_zero/zes_api.h>

#include <algorithm>
#include <fstream>
#include <getopt.h>
#include <iostream>
#include <map>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>
#include <vector>

bool verbose = true;

std::string getErrorString(ze_result_t error)
{
    static const std::map<ze_result_t, std::string> mgetErrorString{
        {ZE_RESULT_NOT_READY, "ZE_RESULT_NOT_READY"},
        {ZE_RESULT_ERROR_DEVICE_LOST, "ZE_RESULT_ERROR_DEVICE_LOST"},
        {ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY, "
   ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY"},
        {ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY, "
   ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY"},
        {ZE_RESULT_ERROR_MODULE_BUILD_FAILURE, "
   ZE_RESULT_ERROR_MODULE_BUILD_FAILURE"},
        {ZE_RESULT_ERROR_MODULE_LINK_FAILURE, "
   ZE_RESULT_ERROR_MODULE_LINK_FAILURE"},
        {ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS, "
   ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS"},
        {ZE_RESULT_ERROR_NOT_AVAILABLE, "ZE_RESULT_ERROR_NOT_AVAILABLE"},
        {ZE_RESULT_ERROR_DEPENDENCY_UNAVAILABLE, "
   ZE_RESULT_ERROR_DEPENDENCY_UNAVAILABLE"},
        {ZE_RESULT_ERROR_UNINITIALIZED, "ZE_RESULT_ERROR_UNINITIALIZED"},
        {ZE_RESULT_ERROR_UNSUPPORTED_VERSION, "
   ZE_RESULT_ERROR_UNSUPPORTED_VERSION"},
        {ZE_RESULT_ERROR_UNSUPPORTED_FEATURE, "
```

```
    ZE_RESULT_ERROR_UNSUPPORTED_FEATURE"},
37      {ZE_RESULT_ERROR_INVALID_ARGUMENT, "
    ZE_RESULT_ERROR_INVALID_ARGUMENT"},
38      {ZE_RESULT_ERROR_INVALID_NULL_HANDLE, "
    ZE_RESULT_ERROR_INVALID_NULL_HANDLE"},
39      {ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE, "
    ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE"},
40      {ZE_RESULT_ERROR_INVALID_NULL_POINTER, "
    ZE_RESULT_ERROR_INVALID_NULL_POINTER"},
41      {ZE_RESULT_ERROR_INVALID_SIZE, "ZE_RESULT_ERROR_INVALID_SIZE"},
42      {ZE_RESULT_ERROR_UNSUPPORTED_SIZE, "
    ZE_RESULT_ERROR_UNSUPPORTED_SIZE"},
43      {ZE_RESULT_ERROR_UNSUPPORTED_ALIGNMENT, "
    ZE_RESULT_ERROR_UNSUPPORTED_ALIGNMENT"},
44      {ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT, "
    ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT"},
45      {ZE_RESULT_ERROR_INVALID_ENUMERATION, "
    ZE_RESULT_ERROR_INVALID_ENUMERATION"},
46      {ZE_RESULT_ERROR_UNSUPPORTED_ENUMERATION, "
    ZE_RESULT_ERROR_UNSUPPORTED_ENUMERATION"},
47      {ZE_RESULT_ERROR_UNSUPPORTED_IMAGE_FORMAT, "
    ZE_RESULT_ERROR_UNSUPPORTED_IMAGE_FORMAT"},
48      {ZE_RESULT_ERROR_INVALID_NATIVE_BINARY, "
    ZE_RESULT_ERROR_INVALID_NATIVE_BINARY"},
49      {ZE_RESULT_ERROR_INVALID_GLOBAL_NAME, "
    ZE_RESULT_ERROR_INVALID_GLOBAL_NAME"},
50      {ZE_RESULT_ERROR_INVALID_KERNEL_NAME, "
    ZE_RESULT_ERROR_INVALID_KERNEL_NAME"},
51      {ZE_RESULT_ERROR_INVALID_FUNCTION_NAME, "
    ZE_RESULT_ERROR_INVALID_FUNCTION_NAME"},
52      {ZE_RESULT_ERROR_INVALID_GROUP_SIZE_DIMENSION, "
    ZE_RESULT_ERROR_INVALID_GROUP_SIZE_DIMENSION"},
53      {ZE_RESULT_ERROR_INVALID_GLOBAL_WIDTH_DIMENSION, "
    ZE_RESULT_ERROR_INVALID_GLOBAL_WIDTH_DIMENSION"},
54      {ZE_RESULT_ERROR_INVALID_KERNEL_ARGUMENT_INDEX, "
    ZE_RESULT_ERROR_INVALID_KERNEL_ARGUMENT_INDEX"},
55      {ZE_RESULT_ERROR_INVALID_KERNEL_ARGUMENT_SIZE, "
    ZE_RESULT_ERROR_INVALID_KERNEL_ARGUMENT_SIZE"},
56      {ZE_RESULT_ERROR_INVALID_KERNEL_ATTRIBUTE_VALUE, "
    ZE_RESULT_ERROR_INVALID_KERNEL_ATTRIBUTE_VALUE"},
57      {ZE_RESULT_ERROR_INVALID_MODULE_UNLINKED, "
    ZE_RESULT_ERROR_INVALID_MODULE_UNLINKED"},
58      {ZE_RESULT_ERROR_INVALID_COMMAND_LIST_TYPE, "
    ZE_RESULT_ERROR_INVALID_COMMAND_LIST_TYPE"},
59      {ZE_RESULT_ERROR_OVERLAPPING_REGIONS, "
    ZE_RESULT_ERROR_OVERLAPPING_REGIONS"},
60      {ZE_RESULT_ERROR_UNKNOWN, "ZE_RESULT_ERROR_UNKNOWN"}};
61   auto i = mgetErrorString.find(error);
62   if (i == mgetErrorString.end())
63      return "ZE_RESULT_ERROR_UNKNOWN";
64   else
65      return mgetErrorString.at(error);
```

```
}

#define VALIDATECALL(myZeCall)                \
    do                                        \
    {                                         \
        ze_result_t r = myZeCall;             \
        if (r != ZE_RESULT_SUCCESS)           \
        {                                     \
            std::cout << getErrorString(r)    \
                      << " returned by "      \
                      << #myZeCall << ": "    \
                      << __FUNCTION__ << ": " \
                      << __LINE__ << "\n";    \
        }                                     \
    } while (0);

void getDeviceHandles(ze_driver_handle_t &driverHandle, std::vector<
    ze_device_handle_t> &devices, int argc, char *argv[])
{

    VALIDATECALL(zeInit(ZE_INIT_FLAG_GPU_ONLY));

    uint32_t driverCount = 0;
    VALIDATECALL(zeDriverGet(&driverCount, nullptr));
    if (driverCount == 0)
    {
        std::cout << "Error could not retrieve driver" << std::endl;
        std::terminate();
    }
    VALIDATECALL(zeDriverGet(&driverCount, &driverHandle));

    uint32_t deviceCount = 0;
    VALIDATECALL(zeDeviceGet(driverHandle, &deviceCount, nullptr));
    if (deviceCount == 0)
    {
        std::cout << "Error could not retrieve device" << std::endl;
        std::terminate();
    }
    devices.resize(deviceCount);
    VALIDATECALL(zeDeviceGet(driverHandle, &deviceCount, devices.data()));

    ze_device_properties_t deviceProperties = {
    ZE_STRUCTURE_TYPE_DEVICE_PROPERTIES};
    for (const auto &device : devices)
    {
        VALIDATECALL(zeDeviceGetProperties(device, &deviceProperties));

        if (verbose)
        {
            std::cout << "Device Name = " << deviceProperties.name << std
    ::endl;
            std::cout << "deviceProperties.flags =  " << deviceProperties.
```

```
    flags << "on device" << device << std::endl;
        }
    }

}

bool validateGetenv(const char *name)
{
    const char *env = getenv(name);
    if ((nullptr == env) || (0 == strcmp("0", env)))
        return false;
    return (0 == strcmp("1", env));
}
int main(int argc, char *argv[])
{
    std::vector<ze_device_handle_t> devices;
    std::vector<ze_device_handle_t> devices;
    std::vector<ze_device_handle_t> devices;
    ze_driver_handle_t driver;

    if (!validateGetenv("ZES_ENABLE_SYSMAN"))
    {
        std::cout << "Must set environment variable ZES_ENABLE_SYSMAN=1"
    << std::endl;
        exit(0);
    }
    getDeviceHandles(driver, devices, argc, argv);

    /*Using the structures zes_device_properties_t and
    zes_pci_properties_t
       to get device properties and pci addresses respectively */
    zes_device_properties_t devProps;
        if (zesDeviceGetProperties(devices[0], &devProps) ==
    ZE_RESULT_SUCCESS)
        {
            printf("    UUID:           %s \n", devProps.core.uuid.id)
            printf("    #subdevices:    %u \n", devProps.numSubdevices)
            printf("    brand:          %s \n", devProps.brandName)
            printf("    model:          %s \n", devProps.modelName)
        }
    zes_pci_properties_t pciProps;
        if (zesDevicePciGetProperties(devices[0], &pciProps) ==
    ZE_RESULT_SUCCESS)
            printf("    PCI address:        %04u:%02u:%02u.%u",
            pciProps.address.domain,
            pciProps.address.bus,
            pciProps.address.device,
            pciProps.address.function);
}
```

(2) Write a program to Write a daemon/application who keep monitoring temperature and if any temperature changes just print to report change in temperature

Solution:

```
*
 * Copyright (C) 2020-2021 Intel Corporation
 *
 * SPDX-License-Identifier: MIT
 *
 */

#include <level_zero/zes_api.h>

#include <algorithm>
#include <fstream>
#include <getopt.h>
#include <iostream>
#include <map>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>
#include <vector>

bool verbose = true;

std::string getErrorString(ze_result_t error)
{
    static const std::map<ze_result_t, std::string> mgetErrorString{
        {ZE_RESULT_NOT_READY, "ZE_RESULT_NOT_READY"},
        {ZE_RESULT_ERROR_DEVICE_LOST, "ZE_RESULT_ERROR_DEVICE_LOST"},
        {ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY, "
   ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY"},
        {ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY, "
   ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY"},
        {ZE_RESULT_ERROR_MODULE_BUILD_FAILURE, "
   ZE_RESULT_ERROR_MODULE_BUILD_FAILURE"},
        {ZE_RESULT_ERROR_MODULE_LINK_FAILURE, "
   ZE_RESULT_ERROR_MODULE_LINK_FAILURE"},
        {ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS, "
   ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS"},
        {ZE_RESULT_ERROR_NOT_AVAILABLE, "ZE_RESULT_ERROR_NOT_AVAILABLE"},
        {ZE_RESULT_ERROR_DEPENDENCY_UNAVAILABLE, "
   ZE_RESULT_ERROR_DEPENDENCY_UNAVAILABLE"},
        {ZE_RESULT_ERROR_UNINITIALIZED, "ZE_RESULT_ERROR_UNINITIALIZED"},
        {ZE_RESULT_ERROR_UNSUPPORTED_VERSION, "
   ZE_RESULT_ERROR_UNSUPPORTED_VERSION"},
        {ZE_RESULT_ERROR_UNSUPPORTED_FEATURE, "
   ZE_RESULT_ERROR_UNSUPPORTED_FEATURE"},
        {ZE_RESULT_ERROR_INVALID_ARGUMENT, "
   ZE_RESULT_ERROR_INVALID_ARGUMENT"},
        {ZE_RESULT_ERROR_INVALID_NULL_HANDLE, "
   ZE_RESULT_ERROR_INVALID_NULL_HANDLE"},
```

```
39          {ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE, "
    ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE"},
40          {ZE_RESULT_ERROR_INVALID_NULL_POINTER, "
    ZE_RESULT_ERROR_INVALID_NULL_POINTER"},
41          {ZE_RESULT_ERROR_INVALID_SIZE, "ZE_RESULT_ERROR_INVALID_SIZE"},
42          {ZE_RESULT_ERROR_UNSUPPORTED_SIZE, "
    ZE_RESULT_ERROR_UNSUPPORTED_SIZE"},
43          {ZE_RESULT_ERROR_UNSUPPORTED_ALIGNMENT, "
    ZE_RESULT_ERROR_UNSUPPORTED_ALIGNMENT"},
44          {ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT, "
    ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT"},
45          {ZE_RESULT_ERROR_INVALID_ENUMERATION, "
    ZE_RESULT_ERROR_INVALID_ENUMERATION"},
46          {ZE_RESULT_ERROR_UNSUPPORTED_ENUMERATION, "
    ZE_RESULT_ERROR_UNSUPPORTED_ENUMERATION"},
47          {ZE_RESULT_ERROR_UNSUPPORTED_IMAGE_FORMAT, "
    ZE_RESULT_ERROR_UNSUPPORTED_IMAGE_FORMAT"},
48          {ZE_RESULT_ERROR_INVALID_NATIVE_BINARY, "
    ZE_RESULT_ERROR_INVALID_NATIVE_BINARY"},
49          {ZE_RESULT_ERROR_INVALID_GLOBAL_NAME, "
    ZE_RESULT_ERROR_INVALID_GLOBAL_NAME"},
50          {ZE_RESULT_ERROR_INVALID_KERNEL_NAME, "
    ZE_RESULT_ERROR_INVALID_KERNEL_NAME"},
51          {ZE_RESULT_ERROR_INVALID_FUNCTION_NAME, "
    ZE_RESULT_ERROR_INVALID_FUNCTION_NAME"},
52          {ZE_RESULT_ERROR_INVALID_GROUP_SIZE_DIMENSION, "
    ZE_RESULT_ERROR_INVALID_GROUP_SIZE_DIMENSION"},
53          {ZE_RESULT_ERROR_INVALID_GLOBAL_WIDTH_DIMENSION, "
    ZE_RESULT_ERROR_INVALID_GLOBAL_WIDTH_DIMENSION"},
54          {ZE_RESULT_ERROR_INVALID_KERNEL_ARGUMENT_INDEX, "
    ZE_RESULT_ERROR_INVALID_KERNEL_ARGUMENT_INDEX"},
55          {ZE_RESULT_ERROR_INVALID_KERNEL_ARGUMENT_SIZE, "
    ZE_RESULT_ERROR_INVALID_KERNEL_ARGUMENT_SIZE"},
56          {ZE_RESULT_ERROR_INVALID_KERNEL_ATTRIBUTE_VALUE, "
    ZE_RESULT_ERROR_INVALID_KERNEL_ATTRIBUTE_VALUE"},
57          {ZE_RESULT_ERROR_INVALID_MODULE_UNLINKED, "
    ZE_RESULT_ERROR_INVALID_MODULE_UNLINKED"},
58          {ZE_RESULT_ERROR_INVALID_COMMAND_LIST_TYPE, "
    ZE_RESULT_ERROR_INVALID_COMMAND_LIST_TYPE"},
59          {ZE_RESULT_ERROR_OVERLAPPING_REGIONS, "
    ZE_RESULT_ERROR_OVERLAPPING_REGIONS"},
60          {ZE_RESULT_ERROR_UNKNOWN, "ZE_RESULT_ERROR_UNKNOWN"}};
61      auto i = mgetErrorString.find(error);
62      if (i == mgetErrorString.end())
63          return "ZE_RESULT_ERROR_UNKNOWN";
64      else
65          return mgetErrorString.at(error);
66  }
67
68  #define VALIDATECALL(myZeCall)                \
69      do                                        \
70      {                                         \
```

```
        ze_result_t r = myZeCall;                \
        if (r != ZE_RESULT_SUCCESS)              \
        {                                        \
            std::cout << getErrorString(r)       \
                      << " returned by "         \
                      << #myZeCall << ": "       \
                      << __FUNCTION__ << ": "    \
                      << __LINE__ << "\n";       \
        }                                        \
    } while (0);

void getDeviceHandles(ze_driver_handle_t &driverHandle, std::vector<
    ze_device_handle_t> &devices, int argc, char *argv[])
{

    VALIDATECALL(zeInit(ZE_INIT_FLAG_GPU_ONLY));

    uint32_t driverCount = 0;
    VALIDATECALL(zeDriverGet(&driverCount, nullptr));
    if (driverCount == 0)
    {
        std::cout << "Error could not retrieve driver" << std::endl;
        std::terminate();
    }
    VALIDATECALL(zeDriverGet(&driverCount, &driverHandle));

    uint32_t deviceCount = 0;
    VALIDATECALL(zeDeviceGet(driverHandle, &deviceCount, nullptr));
    if (deviceCount == 0)
    {
        std::cout << "Error could not retrieve device" << std::endl;
        std::terminate();
    }
    devices.resize(deviceCount);
    VALIDATECALL(zeDeviceGet(driverHandle, &deviceCount, devices.data()));

    ze_device_properties_t deviceProperties = {
    ZE_STRUCTURE_TYPE_DEVICE_PROPERTIES};
    for (const auto &device : devices)
    {
        VALIDATECALL(zeDeviceGetProperties(device, &deviceProperties));

        if (verbose)
        {
            std::cout << "Device Name = " << deviceProperties.name << std
    ::endl;
            std::cout << "deviceProperties.flags =  " << deviceProperties.
    flags << "on device" << device << std::endl;
        }
    }
}
//After getting the device handles
```

```
119 //Let us use this Temperature_Monitor function using the mentioned 3 APIs
    to complete the task
120 void Temperature_Monitor(std::vector<ze_device_handle_t> &devices)
121 {
122     std::cout << std::endl;
123 
124     uint32_t numSens = 0;
125 //We use the zesDeviceEnumTemperatureSensors API to get handle of the
    temperature sensors
126     VALIDATECALL(zesDeviceEnumTemperatureSensors(devices, &numSens, NULL))
    ;
127     if (numSens == 0)
128     {
129         std::cout << "Error: Could not get handle of temp. sensors" << std
    ::endl;
130     }
131 //We use the Temp. properties and state api structures here where we check
     the properties and monitor the state
132     zes_temp_properties_t temp;
133     for (const auto &temp_properties)
134     {
135         float t_i = 0;
136         float t_p = 0;
137 
138         VALIDATECALL(zesTemperatureGetProperties(temp, &temp_properties));
139         if (verbose)
140         {
141            std::cout << "maxTemp "<< temp_properties.maxTemperature;
142            //maxTemperature is the only double value supported, gives
    output in deg. Celcius
143            std::cout<<std::endl;
144         }
145 
146         VALIDATECALL(zesTemperatureGetState(temp, &t_i));
147         if (verbose)
148         {
149             std::cout << "instantaneous_temp" << t_i << std::endl;
150         }
151       float t_d = t_p - t_i;
152         if (verbose)
153         {
154             std::cout << 10 << std::endl;  //Temp Data freq.
155         }
156         t_d = 0;
157         while(1)
158         {
159         VALIDATECALL(zesTemperatureGetState(temp, &t_p));     //Ongoing
    loop for continuous checking and monitoring
160         }
161     }
162 }
163 bool validateGetenv(const char *name)
```

```
{
    const char *env = getenv(name);
    if ((nullptr == env) || (0 == strcmp("0", env)))
        return false;
    return (0 == strcmp("1", env));
}
int main(int argc, char *argv[])
{
    std::vector<ze_device_handle_t> devices;
    ze_driver_handle_t driver;

    if (!validateGetenv("ZES_ENABLE_SYSMAN"))
    {
        std::cout << "Must set environment variable ZES_ENABLE_SYSMAN=1"
   << std::endl;
        exit(0);
    }
    getDeviceHandles(driver, devices, argc, argv);
    Temperature_Monitor(devices[0]);
}
```

(3) Write a program to Read the available frequency clocks and current frequency. Try to set maximum frequency range through API and Now read back what is current frequency set

Solution:

```
/*
 * Copyright (C) 2020-2021 Intel Corporation
 *
 * SPDX-License-Identifier: MIT
 *
 */

#include <level_zero/zes_api.h>

#include <algorithm>
#include <fstream>
#include <getopt.h>
#include <iostream>
#include <map>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>
#include <vector>

bool verbose = true;

std::string getErrorString(ze_result_t error)
{
    static const std::map<ze_result_t, std::string> mgetErrorString{
        {ZE_RESULT_NOT_READY, "ZE_RESULT_NOT_READY"},
        {ZE_RESULT_ERROR_DEVICE_LOST, "ZE_RESULT_ERROR_DEVICE_LOST"},
        {ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY, "
   ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY"},
        {ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY, "
   ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY"},
        {ZE_RESULT_ERROR_MODULE_BUILD_FAILURE, "
   ZE_RESULT_ERROR_MODULE_BUILD_FAILURE"},
        {ZE_RESULT_ERROR_MODULE_LINK_FAILURE, "
   ZE_RESULT_ERROR_MODULE_LINK_FAILURE"},
        {ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS, "
   ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS"},
        {ZE_RESULT_ERROR_NOT_AVAILABLE, "ZE_RESULT_ERROR_NOT_AVAILABLE"},
        {ZE_RESULT_ERROR_DEPENDENCY_UNAVAILABLE, "
   ZE_RESULT_ERROR_DEPENDENCY_UNAVAILABLE"},
        {ZE_RESULT_ERROR_UNINITIALIZED, "ZE_RESULT_ERROR_UNINITIALIZED"},
        {ZE_RESULT_ERROR_UNSUPPORTED_VERSION, "
   ZE_RESULT_ERROR_UNSUPPORTED_VERSION"},
        {ZE_RESULT_ERROR_UNSUPPORTED_FEATURE, "
   ZE_RESULT_ERROR_UNSUPPORTED_FEATURE"},
        {ZE_RESULT_ERROR_INVALID_ARGUMENT, "
   ZE_RESULT_ERROR_INVALID_ARGUMENT"},
```

```
38          {ZE_RESULT_ERROR_INVALID_NULL_HANDLE, "
    ZE_RESULT_ERROR_INVALID_NULL_HANDLE"},
39          {ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE, "
    ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE"},
40          {ZE_RESULT_ERROR_INVALID_NULL_POINTER, "
    ZE_RESULT_ERROR_INVALID_NULL_POINTER"},
41          {ZE_RESULT_ERROR_INVALID_SIZE, "ZE_RESULT_ERROR_INVALID_SIZE"},
42          {ZE_RESULT_ERROR_UNSUPPORTED_SIZE, "
    ZE_RESULT_ERROR_UNSUPPORTED_SIZE"},
43          {ZE_RESULT_ERROR_UNSUPPORTED_ALIGNMENT, "
    ZE_RESULT_ERROR_UNSUPPORTED_ALIGNMENT"},
44          {ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT, "
    ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT"},
45          {ZE_RESULT_ERROR_INVALID_ENUMERATION, "
    ZE_RESULT_ERROR_INVALID_ENUMERATION"},
46          {ZE_RESULT_ERROR_UNSUPPORTED_ENUMERATION, "
    ZE_RESULT_ERROR_UNSUPPORTED_ENUMERATION"},
47          {ZE_RESULT_ERROR_UNSUPPORTED_IMAGE_FORMAT, "
    ZE_RESULT_ERROR_UNSUPPORTED_IMAGE_FORMAT"},
48          {ZE_RESULT_ERROR_INVALID_NATIVE_BINARY, "
    ZE_RESULT_ERROR_INVALID_NATIVE_BINARY"},
49          {ZE_RESULT_ERROR_INVALID_GLOBAL_NAME, "
    ZE_RESULT_ERROR_INVALID_GLOBAL_NAME"},
50          {ZE_RESULT_ERROR_INVALID_KERNEL_NAME, "
    ZE_RESULT_ERROR_INVALID_KERNEL_NAME"},
51          {ZE_RESULT_ERROR_INVALID_FUNCTION_NAME, "
    ZE_RESULT_ERROR_INVALID_FUNCTION_NAME"},
52          {ZE_RESULT_ERROR_INVALID_GROUP_SIZE_DIMENSION, "
    ZE_RESULT_ERROR_INVALID_GROUP_SIZE_DIMENSION"},
53          {ZE_RESULT_ERROR_INVALID_GLOBAL_WIDTH_DIMENSION, "
    ZE_RESULT_ERROR_INVALID_GLOBAL_WIDTH_DIMENSION"},
54          {ZE_RESULT_ERROR_INVALID_KERNEL_ARGUMENT_INDEX, "
    ZE_RESULT_ERROR_INVALID_KERNEL_ARGUMENT_INDEX"},
55          {ZE_RESULT_ERROR_INVALID_KERNEL_ARGUMENT_SIZE, "
    ZE_RESULT_ERROR_INVALID_KERNEL_ARGUMENT_SIZE"},
56          {ZE_RESULT_ERROR_INVALID_KERNEL_ATTRIBUTE_VALUE, "
    ZE_RESULT_ERROR_INVALID_KERNEL_ATTRIBUTE_VALUE"},
57          {ZE_RESULT_ERROR_INVALID_MODULE_UNLINKED, "
    ZE_RESULT_ERROR_INVALID_MODULE_UNLINKED"},
58          {ZE_RESULT_ERROR_INVALID_COMMAND_LIST_TYPE, "
    ZE_RESULT_ERROR_INVALID_COMMAND_LIST_TYPE"},
59          {ZE_RESULT_ERROR_OVERLAPPING_REGIONS, "
    ZE_RESULT_ERROR_OVERLAPPING_REGIONS"},
60          {ZE_RESULT_ERROR_UNKNOWN, "ZE_RESULT_ERROR_UNKNOWN"}};
61      auto i = mgetErrorString.find(error);
62      if (i == mgetErrorString.end())
63          return "ZE_RESULT_ERROR_UNKNOWN";
64      else
65          return mgetErrorString.at(error);
66  }
67
68  #define VALIDATECALL(myZeCall)                \
```

```
69      do                                              \
70      {                                               \
71          ze_result_t r = myZeCall;                   \
72          if (r != ZE_RESULT_SUCCESS)                 \
73          {                                           \
74              std::cout << getErrorString(r)          \
75                        << " returned by "            \
76                        << #myZeCall << ": "          \
77                        << __FUNCTION__ << ": "       \
78                        << __LINE__ << "\n";          \
79          }                                           \
80      } while (0);
81
82  void getDeviceHandles(ze_driver_handle_t &driverHandle, std::vector<
        ze_device_handle_t> &devices, int argc, char *argv[])
83  {
84
85      VALIDATECALL(zeInit(ZE_INIT_FLAG_GPU_ONLY));
86
87      uint32_t driverCount = 0;
88      VALIDATECALL(zeDriverGet(&driverCount, nullptr));
89      if (driverCount == 0)
90      {
91          std::cout << "Error could not retrieve driver" << std::endl;
92          std::terminate();
93      }
94      VALIDATECALL(zeDriverGet(&driverCount, &driverHandle));
95
96      uint32_t deviceCount = 0;
97      VALIDATECALL(zeDeviceGet(driverHandle, &deviceCount, nullptr));
98      if (deviceCount == 0)
99      {
100         std::cout << "Error could not retrieve device" << std::endl;
101         std::terminate();
102     }
103     devices.resize(deviceCount);
104     VALIDATECALL(zeDeviceGet(driverHandle, &deviceCount, devices.data()));
105
106     ze_device_properties_t deviceProperties = {
        ZE_STRUCTURE_TYPE_DEVICE_PROPERTIES};
107     for (const auto &device : devices)
108     {
109         VALIDATECALL(zeDeviceGetProperties(device, &deviceProperties));
110
111         if (verbose)
112         {
113             std::cout << "Device Name = " << deviceProperties.name << std
        ::endl;
114             std::cout << "deviceProperties.flags =  " << deviceProperties.
        flags << "on device" << device << std::endl;
115         }
116     }
```

```
}

void testSysmanFrequency(ze_device_handle_t &device) {
    std::cout << std::endl
              << " ----  Frequency tests ---- " << std::endl;
    bool iamroot = (geteuid() == 0);

    uint32_t count = 0;
    VALIDATECALL(zesDeviceEnumFrequencyDomains(device, &count, nullptr));
    if (count == 0) {
        std::cout << "Could not retrieve frequency domains" << std::endl;
        return;
    }
    std::vector<zes_freq_handle_t> handles(count, nullptr);
    VALIDATECALL(zesDeviceEnumFrequencyDomains(device, &count, handles.
   data()));

    for (const auto &handle : handles) {
        zes_freq_properties_t freqProperties = {};
        zes_freq_range_t freqRange = {};
        zes_freq_range_t testFreqRange = {};
        zes_freq_state_t freqState = {};

        VALIDATECALL(zesFrequencyGetProperties(handle, &freqProperties));
        if (verbose) {
            std::cout << "freqProperties.type = " << freqProperties.type
   << std::endl;
            std::cout << "freqProperties.canControl = " << freqProperties.
   canControl << std::endl;
            std::cout << "freqProperties.isThrottleEventSupported = " <<
   freqProperties.isThrottleEventSupported << std::endl;
            std::cout << "freqProperties.min = " << freqProperties.min <<
   std::endl;
            std::cout << "freqProperties.max = " << freqProperties.max <<
   std::endl;
            if (freqProperties.onSubdevice) {
                std::cout << "freqProperties.subdeviceId = " <<
   freqProperties.subdeviceId << std::endl;
            }
        }

        VALIDATECALL(zesFrequencyGetState(handle, &freqState));
        if (verbose) {
            std::cout << "freqState.currentVoltage = " << freqState.
   currentVoltage << std::endl;
            std::cout << "freqState.request = " << freqState.request <<
   std::endl;
            std::cout << "freqState.tdp = " << freqState.tdp << std::endl;
            std::cout << "freqState.efficient = " << freqState.efficient
   << std::endl;
            std::cout << "freqState.actual = " << freqState.actual << std
   ::endl;
```

```
158             std::cout << "freqState.throttleReasons = " << freqState.
    throttleReasons << std::endl;
159         }
160 
161         VALIDATECALL(zesFrequencyGetRange(handle, &freqRange));
162         if (verbose) {
163             std::cout << "freqRange.min = " << freqRange.min << std::endl;
164             std::cout << "freqRange.max = " << freqRange.max << std::endl;
165         }
166         count = 0;
167         VALIDATECALL(zesFrequencyGetAvailableClocks(handle, &count,
    nullptr));
168         std::vector<double> frequency(count);
169         VALIDATECALL(zesFrequencyGetAvailableClocks(handle, &count,
    frequency.data()));
170         if (verbose) {
171             for (auto freq : frequency) {
172                 std::cout << " frequency = " << freq << std::endl;
173             }
174         }
175         if (iamroot) {
176             // Test setting min and max frequency the same, then restore
    originals
177             testFreqRange.min = freqRange.min;
178             testFreqRange.max = freqRange.min;
179             if (verbose) {
180                 std::cout << "Setting Frequency Range . min " <<
    testFreqRange.min << std::endl;
181                 std::cout << "Setting Frequency Range . max " <<
    testFreqRange.max << std::endl;
182             }
183             VALIDATECALL(zesFrequencySetRange(handle, &testFreqRange));
184             VALIDATECALL(zesFrequencyGetRange(handle, &testFreqRange));
185             if (verbose) {
186                 std::cout << "After Setting Getting Frequency Range . min
    " << testFreqRange.min << std::endl;
187                 std::cout << "After Setting Getting Frequency Range . max
    " << testFreqRange.max << std::endl;
188             }
189             testFreqRange.min = freqRange.min;
190             testFreqRange.max = freqRange.max;
191             if (verbose) {
192                 std::cout << "Setting Frequency Range . min " <<
    testFreqRange.min << std::endl;
193                 std::cout << "Setting Frequency Range . max " <<
    testFreqRange.max << std::endl;
194             }
195             VALIDATECALL(zesFrequencySetRange(handle, &testFreqRange));
196             VALIDATECALL(zesFrequencyGetRange(handle, &testFreqRange));
197             if (verbose) {
198                 std::cout << "After Setting Getting Frequency Range . min
    " << testFreqRange.min << std::endl;
```

```
199                 std::cout << "After Setting Getting Frequency Range . max
    " << testFreqRange.max << std::endl;
200             }
201         } else {
202             std::cout << "Not running as Root. Skipping
    zetSysmanFrequencySetRange test." << std::endl;
203         }
204     }
205 }
206
207 bool validateGetenv(const char *name)
208 {
209     const char *env = getenv(name);
210     if ((nullptr == env) || (0 == strcmp("0", env)))
211         return false;
212     return (0 == strcmp("1", env));
213 }
214 int main(int argc, char *argv[])
215 {
216     std::vector<ze_device_handle_t> devices;
217     ze_driver_handle_t driver;
218
219     if (!validateGetenv("ZES_ENABLE_SYSMAN"))
220     {
221         std::cout << "Must set environment variable ZES_ENABLE_SYSMAN=1"
    << std::endl;
222         exit(0);
223     }
224     getDeviceHandles(driver, devices, argc, argv);
225     testSysmanFrequency(devices[0]);
226 }
```

****

Soham Kulkarni, EE19BTECH11053, IIT Hyderabad