OpenCL_helpers library

hk@r4in.tk
mns@r4in.tk

September 15, 2020

1 Introduction

The OpenCL_helpers library is designed to simplify the programming of multithreaded applications using GPGPU (General-purpose computing on graphics processing units). The library does not cover all of needs of the programming of applications using GPGPU, but it had been written to simplify the making of applications using multilevel (hierarchical) parallelism on one computer. In other words, it allows to parallelise the task into threads on a main computer (CPU) and than parallelise the task inside each GPGPU-device, dividing GPU-threads into squads performing different subtasks.

While designing the library it was assumed that each CPU-thread uses its own separate GPGPU-device, but it is not prohibited to use the same GPGPU-device in two or more CPU-threads. Parallelism at the CPU level is provided by POSIX Threads and parallelism at the GPGPU level is provided by the library.

The library consists of four parts, not all of which are directly related to parallelism and which could be used independently of each other.

The first part is OpenCL programs build tools (s.2). Build tools allow to build OpenCL programs from the command line and view the build result without writing and executing another application.

The second part (s.3) is the CPU-functions of the OpenCL_helpers library and syntax which allows to make header-files common for CPU and GPGPU programs.

The third part (s.4) makes up for the lack of memory management tools in OpenCL C and provides instruments for GPGPU memory allocation and deallocation, heap diagnostics and pointer reinterpretation.

The fourth part is the GPGPU-functions which allow to organize parallelism inside GPGPU, dividing whole amount of GPGPU-threads into squads engaged in performing their own tasks.

1.1 Build and install

1.1.1 Prerequisites

It's supposed that you have:

1. A computer with OS Linux installed.
2. An installed and ready to work C compiler.
3. The installed and available standard C language library (libc).
4. An installed OpenCL software of version 1.2 or later from any vendor.

1.1.2 Getting the source code of the OpenCL_helpers library

A copy of the source code of the OpenCL_helpers library can be downloaded from the address https://ggs.void.r4in.tk/hk/OpenCL_helpers/archive/master.tar.gz than unpacked into a suitable directory.

Besides of that, if VCS Git (https://git-scm.com) is installed, a copy of the source code of the library could be obtained with the following command:

```
git clone https://ggs.void.r4in.tk/hk/OpenCL_helpers.git
```

1.1.3 Build

To build with default settings it's necessary to navigate to the `OpenCL_helpers` directory and run the command

```
make
```

It is acceptable to use option `-j` for multithreaded build. If the `make` command completed without errors, the following files would appear in the `OpenCL_helpers/build` directory:

```
liboclh.so.I.J    oclh_br    oclh_cr    oclh_lr
```

and a few `*.o` subdirectories containing object files. In the name of the first file *I* is the major version of the library and *J* is the minor version.

Each file could be built separately with commands:

```
make oclh_library
make oclh_builder
make oclh_compiler
make oclh_linker
```

If it was necessary, the library could be built for debugging with the command:

```
make debug
```

1.1.4 Installation

Installation is performed by the command:

```
make install
```

As the result the `~/opt/oclh` directory is created, where executable files, the library file and header files are copied into the `bin`, `lib`, `include` subdirectories respectively. After that it is advisable to add the `~/opt/oclh/bin` directory to the `PATH` environment variable and the `~/opt/oclh/lib` directory to the `LD_LIBRARY_PATH` environment variable.

The destination path can be changed with the command:

```
make PRFX_PATH=destination_path install
```

1.1.5 Uninstallation

Uninstallation is performed by the command:

```
make uninstall
```

or

```
make PRFX_PATH=destination_path uninstall
```

if the library had been installed in a non-default directory.

1.1.6 Documentation

The documentation of the OpenCL_helpers library is built separately. To build the documentation, it's necessary to have the XeTeX/XeLaTeX typesetting system or another TeX/LaTeX-compatible system. The XeTeX system and related packages are provided within the TeX Live distri-

bution (https://www.tug.org/texlive/). Using a system other than XeTeX may require changes in the source code of the documentation.

In addition to the typesetting system itself, it's necessary to have a number of packages, for example, xindy for composition of the index. All packages used for preparation of the documentation are freely available as part of the TeX Live distribution.

The documentation build itself is performed in the `OpenCL_helpers/documentation` directory by running build script

```
./build_script
```

If no errors occurred during the execution of the current script, the following files would appear in the `OpenCL_helpers/documentation/build` directory:

```
opencl_helpers_documentation-russian.pdf
opencl_helpers_documentation-english.pdf
```

which contain the documentation in Russian and English languages, respectively.

The build uses fonts of the IBM Plex family, but it is possible to return to the basic Computer Modern family by uncommenting the line

```
\input{fonts/font_settings-Computer_Modern.tex}
```

in the preamble of the documentation source code.

1.2 Log file format

The library tools allow maintain a log in log files about events occurring in an application, in addition, the library itself, if necessary, writes to the log file. Description of logging functions is given in s.3.2.

The standard log file entry looks like

```
YYYY-MM-DD hh:mm:ss ws_0xHHHH entry_content
```

where

- *YYYY* – year written in four decimal digits;
- *MM* – month of the year written in two decimal digits from 01 to 12;
- *DD* – day of the month written in two decimal digits from 01 to 31;
- *hh* – hour of the day written in two decimal digits from 00 to 23;
- *mm* – minute of the hour written in two decimal digits from 00 to 59;
- *ss* – second of the minute written in two decimal digits from 00 to 59;
- *HHHH* – the last two bytes of an address of the working configuration of the GPGPU device (workset, for details see s.3.1) written in four hexadecimal digits.

entry_content – can be any text which passed to a logging function, but the library itself obeys, if possible, the next conventions:

1. Information related to OpenCL instances is recorded as *instance_type*_0x*HHHH*, where *HHHH* – the last two bytes of the instance address, written in four hexadecimal digits. So, for example, a GPGPU device could be recorded as `dev_0x2a78`, and a platform as `platform_0xf190`. An exhaustive list of OpenCL instances is given in the OpenCL specifications.

2. As a delimiter of information blocks in the entries and marking the relativity of such blocks, the symbol «|» is used. So, the entry

```
2019-06-03 15:42:47 ws_0x9c00 context_0x9f60 |  dev_0xf260 | ...
```

means that entry describes event related to OpenCL context `0x9f60` using GPGPU device `0xf260`.

3. In case of recording information that is an explicitation, an additional space is put before it, for example:

```
2019-06-03 15:42:47 ws_0x9c00 context_0x9f60 | Reference count: 1
2019-06-03 15:42:47 ws_0x9c00 context_0x9f60 | Number of devices: 1
2019-06-03 15:42:47 ws_0x9c00 context_0x9f60 | Device ID(s): 0x1acf260
2019-06-03 15:42:47 ws_0x9c00 context_0x9f60 |  dev_0xf260 | GPU: 15 units/17...
2019-06-03 15:42:47 ws_0x9c00 context_0x9f60 |  dev_0xf260 | Memory: 8116.43...
2019-06-03 15:42:47 ws_0x9c00 context_0x9f60 |  dev_0xf260 | Vendor: NVIDIA Corp...
2019-06-03 15:42:47 ws_0x9c00 context_0x9f60 |  dev_0xf260 | Model: GeForce GT...
2019-06-03 15:42:47 ws_0x9c00 context_0x9f60 | Context properties:
2019-06-03 15:42:47 ws_0x9c00 context_0x9f60 |  Platform: 0xf190
2019-06-03 15:42:47 ws_0x9c00 context_0x9f60 |   platform_0xf190 | Profile: FULL_PROFILE
2019-06-03 15:42:47 ws_0x9c00 context_0x9f60 |   platform_0xf190 | Version: OpenCL 1...
2019-06-03 15:42:47 ws_0x9c00 context_0x9f60 |   platform_0xf190 | Name: NVIDIA CUDA
2019-06-03 15:42:47 ws_0x9c00 context_0x9f60 |   platform_0xf190 | Vendor: NVIDIA Corp...
2019-06-03 15:42:47 ws_0x9c00 context_0x9f60 |   platform_0xf190 | Extensions: cl_khr...
2019-06-03 15:42:47 ws_0x9c00 context_0x9f60 |  Is user responsible for sync: Undefined (presumable No)
```

4. If an error occurred during the execution of the library function, there would be added to the log an entry starting with `oclerr:` and containing information about all function calls from the library to the OpenCL API. So, the entry

```
YYYY-MM-DD hh:mm:ss ws_0xHHHH oclerr:
_ghf_getBuildStatus/clGetProgramBuildInfo/CL_PROGRAM_BUILD_STATUS
returned error -3 - CL_COMPILER_NOT_AVAILABLE
```

means that the `_ghf_getBuildStatus` function called the OpenCL API function `clGetProgramBuildInfo` with the argument `CL_PROGRAM_BUILD_STATUS` and received as a response the `-3` error code, which stands for `CL_COMPILER_NOT_AVAILABLE`.

Given that OpenCL instance addresses are unique for one application run, it is highly likely that the combination of the name of the instance and the last two bytes of its address is also unique. Therefore, the use of these conventions allows, with substring filtering, obtain the necessary information from the log file for a particular OpenCL instance.

In addition to the standard log entry there is also the header entry, which looks like

```
YYYY-MM-DD hh:mm:ss ws_0xHHHH __________
YYYY-MM-DD hh:mm:ss ws_0xHHHH Title_text
YYYY-MM-DD hh:mm:ss ws_0xHHHH ~~~~~~~~~~
```

and the delimiter entry, which looks like

```
YYYY-MM-DD hh:mm:ss ws_0xHHHH ____________________________________________________
YYYY-MM-DD hh:mm:ss ws_0xHHHH ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

1.3 Naming conventions

The following substitutions were used to describe the conventions:

`*` – any string of characters;
`Action` – semantic name of an action, for example, «Sync» for synchronization or «Fill» for filling;
`Tname` – **t**ype **name**, semantic name of an data type, it does not obligingly corresponds to the technical name of the structure;
`BTA` – **b**ase **t**ype **a**cronym, acronym for a basic data type, so the following acronyms were used in the library already:

`i8`	(`signed char`),	`u8`	(`unsigned char`),
`i16`	(`signed short`),	`u16`	(`unsigned short`),
`i32`	(`signed int`),	`u32`	(`unsigned int`),
`i64`	(`signed long int`),	`u64`	(`unsigned long int`),
`f32`	(`float`),	`f64`	(`double`).

The following naming conventions were used in the library:

`__*`
: the library instances for internal use begin with two underscores. While regular use of the library, using of such instances is not assumed.

Macrodefinitions

`_GHM_*`
: **g**pgpu **h**ost **m**acro, preprocessor macrodefinition for the CPU program compiler.

`_GDM_*`
: **g**pgpu **d**evice **m**acro, preprocessor macrodefinition for the OpenCL program compiler.

`_GHDM_*`
: **g**pgpu **h**ost-**d**evice **m**acro, preprocessor macrodefinition common for the CPU program compiler and the OpenCL program compiler.

Data types

`_GHT_*`
: **g**pgpu **h**ost **t**ype, data type for CPU programs.

`_GDT_*`
: **g**pgpu **d**evice **t**ype, data type for OpenCL programs.

`_GHDT_*`
: **g**pgpu **h**ost-**d**evice **t**ype, data type common for CPU and OpenCL programs.

Enumerations

`_GHE_*`
: **g**pgpu **h**ost **e**numeration, enumeration for CPU programs.

`_GDE_*`
: **g**pgpu **d**evice **e**numeration, enumeration for OpenCL programs.

`_GHDE_*`
: **g**pgpu **h**ost-**d**evice **e**numeration, enumeration common for CPU and OpenCL programs.

Functions

`_ghf_*()`
: **g**pgpu **h**ost **f**unction, library function available for CPU programs only.

`_gdf_*()`
: **g**pgpu **d**evice **f**unction, library function available for OpenCL programs only.

`_ghdf_*()`
: **g**pgpu **h**ost-**d**evice **f**unction, library function common for CPU and OpenCL programs.

`*_wdc*()`
: **w**ith **d**ata **c**leaner, special type of function. If error occurred while the execution of such function, then callback function would be called which deallocates memory and sets to zeroes members of structure from the user data member of the workset, for details see s.3.1.1.

`*_decl`*Tname*`()`
`*_decl`*Tname_BTA*`()`
: **decl**arator, function returns a structure with members initialized to default values. The function does not allocate memory and *Tname*'s members-pointers are set to `NULL`. Assignment the value returned such a function to an existing structure may lead to a memory leak, therefore `*_decl*()` functions are being called for structure declaration only.

`*_genr`*Tname*`()`
`*_genr`*Tname_BTA*`()`
: **gen**e**r**ator, the function allocates memory for all members-pointer of the *Tname* structure, the pointer to which is obtained from the arguments. Then the function assign values in accordance with its arguments. Such a function is analogue of the constructor. If an error occurred, the function would return `int` value other than zero and completely deallocate memory of the *Tname* structure, including the members-structures. *Tname* is a semantic name, that does not obligingly corresponds to the technical name of the structure.

> Important warning: If a pointer to an existing structure was passed as an argument of the `*_genr*()` function, the structure would be correctly recreated with the deletion of all previous data and the deallocation of the corresponding memory, including the members-structures.

`*_is`*Tname*`_Valid()`
`*_is`*Tname_BTA*`_Valid()`
: the function performs minimal integrity check of the structure data and returns an `int` value. If the structure data is integral, the `*_is*_Valid()` function returns 1, otherwise 0 is returned.

`*`*ActionTname*`()`
`*`*ActionTname_BTA*`()`
: the function performs *Action* over a *Tname* type. If an error occurred, an `int` value other than zero would be returned and the memory of the structure would be completely deallocated, including the members-structures. *Tname* is a semantic name, that does not obligingly corresponds to the technical name of the structure.

`*_wipe`*Tname*`()`
`*_wipe`*Tname_BTA*`()`
: the function completely deallocates the memory occupied by the members of the *Tname* structure, including the members-structures, then assigns default values to all of the structure members, and `NULL` to the members-pointers. After applying the `*_wipe*()` function, the state of the *Tname* structure is fully equivalent to the value returned by the `*_decl*()` function and the memory occupied by the structure can be deallocated or the `*_genr*()` function can be applied again. *Tname* is a semantic name, that does not obligingly corresponds to the technical name of the structure.

`*_get`*Tname*`()`
: the indirect data access function, if is called returns the *Tname* value obtained from the function arguments. *Tname* is a semantic name, that does not obligingly corresponds to the technical name of the structure.

Files

`*.clc`
: file with the source code of the OpenCL program in OpenCL C language.

`*.clh`
: header file with the source code of the OpenCL program in OpenCL C language.

`*.clo`
: compiled OpenCL object. It only makes sense for OpenCL devices of the same architecture.

`*.clso`
: linked OpenCL library (shared object/library). It only makes sense for OpenCL devices of the same architecture.

`*.clexe`
: linked OpenCL executable. It only makes sense for OpenCL devices of the same architecture.

`*.clout`
: an OpenCL file, content of which could not be identified using the library.

`*.log`
: log file.

2 OpenCL programs build tools

The library includes three executable files:

- `oclh_cr` – compiles an OpenCL program into an OpenCL object;
- `oclh_lr` – links OpenCL objects;
- `oclh_br` – completly builds an OpenCL program.

During the execution of these programs, a detailed diagnostic log is being maintained in the `oclh_*r.log` file (according to the name of the tool), where excessive information is stored on all available GPGPU devices, used platforms, and contexts created for build. In fact, you can run, for example, `oclh_cr` with any input file, even with itself as `./oclh_cr oclh_cr`. The input file, of course, will not be built into an OpenCL object, but the `oclh_cr.log` log file will contain complete information on GPGPU devices found in the system. The log file format is human-readable, adapted to search for substrings using the `grep` command and analogues. The log file format is described in s.1.2.

Let us take a look at the use cases for each of these tools.

2.1 Compilation

Isolated compilation is performed by `oclh_cr`.

2.1.1 Synopsis

```
oclh_cr [--dev-idxs=#,#,... | --dev-name=mask]
        [--verbatim-output-name] [-o outfile]
        [COMPILER_OPTIONS] infile...
```

2.1.2 Description

When `oclh_cr` is called, source code is compiled from files *infile*`...` for all GPGPU devices available on the system. If the `--dev-idxs=`*#*`,`*#*`,...` option was specified, then compilation would be performed only for the GPGPU devices with the *#*`,`*#*`,...` indices (for details, see s.2.1.3). If the `--dev-name=`*mask* was specified, then compilation would be performed only for the GPGPU devices whose model matches the *mask* (for details, see s.2.1.3).

During the execution of the `oclh_cr` tool, a detailed diagnostic log is being maintained in the `oclh_cr.log` file, where excessive information is stored on all available GPGPU devices, used platforms, and contexts created for compilation. In fact, you can run `oclh_cr` with any input file, even with itself as `./oclh_cr oclh_cr`. The input file, of course, will not be compiled into an OpenCL object, but the `oclh_cr.log` log file will contain complete information on GPGPU devices found in the system. The log file format is human-readable, adapted to search for substrings using the `grep` command and analogues. The log file format is described in s.1.2.

Considering that compilation may be performed for several devices of different vendors, the program compilation log is maintained in different log files *outfile*`-`*GPGPU_device_model*`-trans.log`.

The result of the compiler's work is an unlinked binary object saved in the *outfile*`-`*GPGPU_device_model*`.clo` file. If the option `--verbatim-output-name` was specified, then the result would be saved in the *outfile* file. Sometimes a situation arises when the vendor's OpenCL library generates several binary objects as a result, in which case all binary objects will be saved, but the postfix *.N* will be added to the file names, where *N* is a decimal number denoting the sequence number (starting from zero) of the binary object generated by the compiler of the vendor of the GPGPU device.

If the option `-o` is not specified, then the *outfile* in the file name will be replaced by a substring of the form `program_0x`*HHHH*. In case if the GPGPU device model is not identified by the OpenCL means, then the *GPGPU_device_model* will be replaced by a substring of the form `dev_0x`*HHHH*. In the replacements mentioned above, *HHHH* is hexadecimal representation of the last two bytes of the program and the GPGPU device addresses, respectively. Given that addresses of the program and GPGPU device are unique for one application run, it is highly likely that the combination of the name of the instance and the last two bytes of its address is also unique, so can be used as a substring to search related entries in the main log file `oclh_cr.log`.

The main log file name `oclh_cr.log` and the file saving path can be changed when building the OpenCL_helpers library in the header file

```
src/inc/oclh_settings.h
```

That name and the path to save logs and compilation results are defined in macrodefinitions

```
#define _GHM_LOG_PATH                    "."
#define _GHM_OCLH_COMPILER_LOG_FILENAME "oclh_cr.log"
```

The compiler always receives the `-D_OCLH_OCL_COMPILER_` argument. It is hardcoded in the library code and introduced for ability to use header files both in programs for GPGPU programs and CPU without changing them. For details, see s.3.3.

2.1.3 Arguments

`--dev-idxs=`*#*`,`*#*`,...`

the *#*`,`*#*`,...` numbers specified without spaces separated by commas after the `--dev-idxs=` option are sequence numbers (indices) of GPGPU devices in the system for which the compilation will be performed. Indices start with zero. You can find out the indice of a specific device from the log file, in the first section of which in the device description the first line has the form

YYYY-MM-DD hh`:`*mm*`:ss ws_0x`*HHHH* `dev_0x`*HHHH* `| Device index:` *N*

where *N* is the indice of this device.

`--dev-name=`*mask*

the *mask* string specified after the `--dev-name=` option is a wildcard that defines which device models present in the system the compilation will be performed for. Wildcard characters are:

`?` – matches any single character;

`*` – matches any number of any characters including none.

In the absence of wildcard characters the *mask* is considered to be the exact name of the device model. You can find out the model of a specific device from the log file, in the first section of which in the device description there is the line of the form

YYYY-MM-DD hh`:`*mm*`:`*ss* `ws_0x`*HHHH* `dev_0x`*HHHH* `| Device name:` *model*

where *model* is the string that is checked for matching with the *mask*.

`--verbatim-output-name`

the given option instructs the compiler not to add the GPGPU device model and an extension to the output file name, but to use it exactly as described. But, if the result of the compiler's work was more than one binary object, then the *.N* postfix would be added to the file name, where *N* is decimal number denoting the sequence number (starting from zero) of the binary object formed by the compiler of the GPGPU device vendor.

`-o` *outfile*

the *outfile* string is the name of the output file. If the `--verbatim-output-name` option was not specified, then the *outfile* string would be used as the prefix of the *outfile*`-GPGPU_`*device_model*`.clo` file name, which contains the binary object generated as the result of compilation. If the `--verbatim-output-name` option was specified, then the *outfile* string would be used «as is», unless several binary objects was generated as result of compilation – in this case, all binary objects would be saved with the *outfile*`.`*N* file names, where *N* is the sequence number of the binary object starting from zero. Additionally, the *outfile* string is used as the prefix in the name of the compilation log file.

> Important warning: Space characters at the beginning and the end of the *outfile* string are deleted. Space characters inside of the *outfile* string are replaced with underscores.

`COMPILER_OPTIONS`

compiler arguments. In unchanged form and with the preservation of the sequence passed to the compiler of the vendor. The compiler arguments themselves are described in the OpenCL specifications, in addition, the vendor's compiler can support additional arguments not fixed in the OpenCL specifications.

infile`...`

the list separated by spaces with the names of the files containing the source code of the OpenCL C or OpenCL C++ program. The file name cannot begin with a «`-`» character.

2.2 Linking

Isolated linking is performed by `oclh_lr`.

2.2.1 Synopsis

```
oclh_lr [--dev-idxs=#,#,... | --dev-name=mask]
        [--verbatim-output-name] [-o outfile]
        [LINKER_OPTIONS] infile...
```

2.2.2 Description

When `oclh_lr` is called, OpenCL objects are linked from files *infile*`...` for all GPGPU devices available on the system. It should be understood that only objects compiled for one GPGPU architecture can be linked, and GPGPU architectures can differ even from one vendor's devices. An attempt to link object compiled for an architecture different from the device architecture will result in error `-42` OpenCL API:

```
oclerr: clCreateProgramWithBinary returned error -42 - CL_INVALID_BINARY
```

According to mentioned above, it is recommended to run the linker for only one device or device model using option `--dev-idxs=`*#*`,`*#*`,...` or `--dev-name=`*mask*. If the `--dev-idxs=`*#*`,`*#*`,...` option was specified, then linking would be performed only for the GPGPU devices with the *#*`,`*#*`,...` indices (for details, see s.2.2.3). If the `--dev-name=`*mask* was specified, then linking would be performed only for the GPGPU devices whose model matches the *mask* (for details, see s.2.2.3).

During the execution of the `oclh_lr` tool, a detailed diagnostic log is being maintained in the `oclh_lr.log` file, where excessive information is stored on all available GPGPU devices, used platforms, and contexts created for linking. The log file format is described in s.1.2.

The linking log is maintained in log files *outfile*`-`*GPGPU_device_model*`-link.log`.

The result of the linker's work is an executable binary object or a shared binary object (library) saved in the *outfile*`-`*GPGPU_device_model*`.clexe` or *outfile*`-`*GPGPU_device_model*`.clso` file, respectively. If the option `--verbatim-output-name` was specified, then the result would be saved in the *outfile* file. Sometimes a situation arises when the vendor's OpenCL library generates several binary objects as a result, in which case all binary objects will be saved, but the postfix *.N* will be added to the file names, where *N* is a decimal number denoting the sequence number (starting from zero) of the binary object generated by the linker of the vendor of the GPGPU device.

If the option `-o` is not specified, then the *outfile* in the file name will be replaced by a substring of the form `program_0x`*HHHH*. In case if the GPGPU device model is not identified by the OpenCL means, then the *GPGPU_device_model* will be replaced by a substring of the form `dev_0x`*HHHH*. In the replacements mentioned above, *HHHH* is hexadecimal representation of the last two bytes of the program and the GPGPU device addresses, respectively. Given that addresses of the program and GPGPU device are unique for one application run, it is highly likely that the combination of the name of the instance and the last two bytes of its address is also unique, so can be used as a substring to search related entries in the main log file `oclh_lr.log`.

The main log file name `oclh_lr.log` and the file saving path can be changed when building the OpenCL_helpers library in the header file

```
src/inc/oclh_settings.h
```

That name and the path to save logs and linking results are defined in macrodefinitions

```
#define _GHM_LOG_PATH                 "."
#define _GHM_OCLH_LINKER_LOG_FILENAME "oclh_lr.log"
```

2.2.3 Arguments

`--dev-idxs=`*#*`,`*#*`,...`

the *#*`,`*#*`,...` numbers specified without spaces separated by commas after the `--dev-idxs=` option are sequence numbers (indices) of GPGPU devices in the system for which the linking will be performed. Indices start with zero. You can find out the indice of a specific device from the log file, in the first section of which in the device description the first line has the form

YYYY-MM-DD hh`:`*mm*`:`*ss* `ws_0x`*HHHH* `dev_0x`*HHHH* `| Device index:` *N*,

where *N* is the indice of this device.

`--dev-name=`*mask*

the *mask* string specified after the `--dev-name=` option is a wildcard that defines which device models present in the system the linking will be performed for. Wildcard characters are:

? – matches any single character;

* – matches any number of any characters including none.

In the absence of wildcard characters the *mask* is considered to be the exact name of the device model. You can find out the model of a specific device from the log file, in the first section of which in the device description there is the line of the form

YYYY-MM-DD hh:*mm*:*ss* ws_0x*HHHH* dev_0x*HHHH* | Device name: *model*

where *model* is the string that is checked for matching with the *mask*.

--verbatim-output-name

the given option instructs the linker not to add the GPGPU device model and an extension to the output file name, but to use it exactly as described. But, if the result of the linker's work was more than one binary object, then the .*N* postfix would be added to the file name, where *N* is decimal number denoting the sequence number (starting from zero) of the binary object formed by the linker of the GPGPU device vendor.

-o *outfile*

the *outfile* string is the name of the output file. If the --verbatim-output-name option was not specified, then the *outfile* string would be used as the prefix of the *outfile*-*GPGPU_device_model*.clexe or *outfile*-*GPGPU_device_model*.clso file name, which contains the binary object generated as the result of linking. If the --verbatim-output-name option was specified, then the *outfile* string would be used «as is», unless several binary objects was generated as result of linking – in this case, all binary objects would be saved with the *outfile*.*N* file names, where *N* is the sequence number of the binary object starting from zero. Additionally, the *outfile* string is used as the prefix in the name of the linking log file.

> Important warning: Space characters at the beginning and the end of the *outfile* string are deleted. Space characters inside of the *outfile* string are replaced with underscores.

LINKER_OPTIONS

linker arguments. In unchanged form and with the preservation of the sequence passed to the linker of the vendor. The linker arguments themselves are described in the OpenCL specifications, in addition, the vendor's linker can support additional arguments not fixed in the OpenCL specifications.

infile...

the list separated by spaces with the names of the files containing the OpenCL compiled objects. The file name cannot begin with a «-» character.

2.3 Complete build routine

Complete build routine is performed by oclh_br.

2.3.1 Synopsis

```
oclh_br [--dev-idxs=#,#,... | --dev-name=mask]
        [--verbatim-output-name] [-o outfile]
        [COMPILER_OPTIONS] [LINKER_OPTIONS] infile...
```

2.3.2 Description

When oclh_br is called, a complete build routine (compilation and linking) of the source code from from files *infile*... is performed for all GPGPU devices available on the system. If the --dev-idxs=#,#,... option was specified, then build would be performed only for the GPGPU devices with the #,#,... indices (for details, see s.2.3.3). If the --dev-name=*mask* was specified, then

build would be performed only for the GPGPU devices whose model matches the *mask* (for details, see s.2.3.3).

> Important warning: At first glance, using the builder seems preferable than separate compilation and linking, as it allows to get executable files from the source code for all OpenCL devices at once. However, there are nuances that are related to the fact that the work of the builder is determined by the library of the GPGPU device vendor. So, during testing, the following problems were identified:
>
> (1) not all builders support the generation of libraries, most of them create an executable file, so to create exactly the library, it is necessary to use the linker with the `-create-library` option (according to the current OpenCL specification);
>
> (2) for some unclear reason, some builders ignore the declaration of a function and interpret only its definition, what leads to the necessity of specifying files with the definition of the function in the *infile*`...` list of files with source codes strictly before the first use of the function, despite the declaration of the function in the headers.
>
> Perhaps there are or will appear other pitfalls in using vendors' builders, so it is recommended to use separate compilation and linking.

During the execution of the `oclh_br` tool, a detailed diagnostic log is being maintained in the `oclh_br.log` file, where excessive information is stored on all available GPGPU devices, used platforms, and contexts created for build. The log file format is described in s.1.2.

Considering that build may be performed for several devices of different vendors, the program build log is maintained in different log files *outfile*`-`*GPGPU_device_model*`-build.log`.

The result of the builder's work is an executable binary object saved in the *outfile*`-`*GPGPU_device_model*`.clexe` file. If the option `--verbatim-output-name` was specified, then the result would be saved in the *outfile* file. Sometimes a situation arises when the vendor's OpenCL library generates several binary objects as a result, in which case all binary objects will be saved, but the postfix `.`*N* will be added to the file names, where *N* is a decimal number denoting the sequence number (starting from zero) of the binary object generated by the builder of the vendor of the GPGPU device. An introduction to the file name of this number is due to the fact that, when implementing OpenCL, vendors are free to choose the output format of the built program. So, for example, the implementation from Intel will create and output a binary object, which is an ELF-file; and the NVidia's OpenCL implementation will output a human-readable text block of ptx-code (it's kind of Assembler variation for GPGPU, also named IR/IL – Intermediate Representation/Intermediate Language). Moreover, sometimes several binary objects with different contents can be generated. Before saving, what exactly is contained in the binary object is not possible by OpenCL means, therefore binary objects are simply numbered by *N* in the order in which they are stored in memory by the OpenCL implementation.

If the option `-o` is not specified, then the *outfile* in the file name will be replaced by a substring of the form `program_0x`*HHHH*. In case if the GPGPU device model is not identified by the OpenCL means, then the *GPGPU_device_model* will be replaced by a substring of the form `dev_0x`*HHHH*. In the replacements mentioned above, *HHHH* is hexadecimal representation of the last two bytes of the program and the GPGPU device addresses, respectively. Given that addresses of the program and GPGPU device are unique for one application run, it is highly likely that the combination of the name of the instance and the last two bytes of its address is also unique, so can be used as a substring to search related entries in the main log file `oclh_br.log`.

The main log file name `oclh_br.log` and the file saving path can be changed when building

the OpenCL_helpers library in the header file

```
src/inc/oclh_settings.h
```

That name and the path to save logs and build results are defined in macrodefinitions

```
#define _GHM_LOG_PATH                    "."
#define _GHM_OCLH_BUILDER_LOG_FILENAME "oclh_br.log"
```

The builder always receives the `-D_OCLH_OCL_COMPILER_` argument. It is hardcoded in the library code and introduced for ability to use header files both in programs for GPGPU programs and CPU without changing them. For details, see s.3.3.

2.3.3 Arguments

`--dev-idxs=`*#*`,`*#*`,...`

the *#*,*#*,... numbers specified without spaces separated by commas after the `--dev-idxs=` option are sequence numbers (indices) of GPGPU devices in the system for which the build will be performed. Indices start with zero. You can find out the indice of a specific device from the log file, in the first section of which in the device description the first line has the form

YYYY-MM-DD hh:*mm*:*ss* ws_0x*HHHH* dev_0x*HHHH* | Device index: *N*,

where *N* – is the indice of this device.

`--dev-name=`*mask*

the *mask* string specified after the `--dev-name=` option is a wildcard that defines which device models present in the system the build will be performed for. Wildcard characters are:

? – matches any single character;

* – matches any number of any characters including none.

In the absence of wildcard characters the *mask* is considered to be the exact name of the device model. You can find out the model of a specific device from the log file, in the first section of which in the device description there is the line of the form

YYYY-MM-DD hh:*mm*:*ss* ws_0x*HHHH* dev_0x*HHHH* | Device name: *model*

where *model* is the string that is checked for matching with the *mask*.

`--verbatim-output-name`

the given option instructs the builder not to add the GPGPU device model and an extension to the output file name, but to use it exactly as described. But, if the result of the builder's work was more than one binary object, then the `.`*N* postfix would be added to the file name, where *N* is decimal number denoting the sequence number (starting from zero) of the binary object formed by the builder of the GPGPU device vendor.

`-o` *outfile*

the *outfile* string is the name of the output file. If the `--verbatim-output-name` option was not specified, then the *outfile* string would be used as the prefix of the *outfile*`-GPGPU_`*device_model*`.clexe` file name, which contains the binary object generated as the result of build. If the `--verbatim-output-name` option was specified, then the *outfile* string would be used «as is», unless several binary objects was generated as result of build – in this case, all binary objects would be saved with the *outfile*`.`*N* file names, where *N* is the sequence number of the binary object starting from zero. Additionally, the *outfile* string is used as the prefix in the name of the build log file.

> Important warning: Space characters at the beginning and the end of the *outfile* string are deleted. Space characters inside of the *outfile* string are replaced with underscores.

`COMPILER_OPTIONS`

compiler arguments. In unchanged form and with the preservation of the sequence passed to the builder of the vendor. The compiler arguments themselves are described in the OpenCL specifications, in addition, the vendor's builder can support additional arguments not fixed in the OpenCL specifications.

`LINKER_OPTIONS`
: linker arguments. In unchanged form and with the preservation of the sequence passed to the builder of the vendor. The linker arguments themselves are described in the OpenCL specifications, in addition, the vendor's builder can support additional arguments not fixed in the OpenCL specifications.

`infile``...`
: the list separated by spaces with the names of the files containing the source code of the OpenCL C or OpenCL C++ program. The file name cannot begin with a «-» character.

3 Using the OpenCL_helpers library. Structures, functions and headers

Stub. The section will be completed after sufficient testing of functionality.

3.1 Structures

Stub. The section will be completed after sufficient testing of functionality.

3.1.1 Main structure of the working configuration

Stub. The section will be completed after sufficient testing of functionality.

3.2 Logging functions

Stub. The section will be completed after sufficient testing of functionality.

3.3 Common header files for CPU and GPGPU code

Stub. The section will be completed after sufficient testing of functionality.

4 Memory management and pointer reinterpretation in OpenCL C programs

Stub. The section will be completed after sufficient testing of functionality.

5 Parallelism inside GPU

Stub. The section will be completed after sufficient testing of functionality.

Index

L

log file format 3

N

naming conventions 5

O

`oclh_br` 11

`oclh_cr` 7

`oclh_lr` 9

OpenCL builder 11

OpenCL compiler 7

OpenCL linker 9

Contents

1 Introduction **1**

1.1 Build and install . . . 1

1.1.1 Prerequisites . . . 1

1.1.2 Getting the source code of the OpenCL_helpers library . . . 2

1.1.3 Build . . . 2

1.1.4 Installation . . . 2

1.1.5 Uninstallation . . . 2

1.1.6 Documentation . . . 2

1.2 Log file format . . . 3

1.3 Naming conventions . . . 5

2 OpenCL programs build tools **7**

2.1 Compilation . . . 7

2.1.1 Synopsis . . . 7

2.1.2 Description . . . 8

2.1.3 Arguments . . . 8

2.2 Linking . . . 9

2.2.1 Synopsis . . . 9

2.2.2 Description . . . 10

2.2.3 Arguments . . . 10

2.3 Complete build routine . . . 11

2.3.1 Synopsis . . . 11

2.3.2 Description . . . 11

2.3.3 Arguments . . . 13

3 Using the OpenCL_helpers library. Structures, functions and headers **14**

3.1 Structures . . . 14

3.1.1 Main structure of the working configuration . . . 14

3.2 Logging functions . . . 14

3.3 Common header files for CPU and GPGPU code . . . 14

4 Memory management and pointer reinterpretation in OpenCL C programs **14**

5 Parallelism inside GPU **14**

Index **15**