CDSC Mapping Toolchain

The Center for Domain-Specific Computing

March 2014

Disclaimer

Center for Domain-Specific Computing The CDSC Mapper has been developed by the Center for Domain-Specific Computing, funded by the US National Science Fundation Expeditions in Computing award 0926127. The center is led by Prof. Jason Cong (UCLA, director) and Prof. Vivek Sarkar (Rice University, co-director).
Contributors to the CDSC Mapper include:

- Tom Henretty, Justin Holewinski, Atanas Rountev, P. Sadayappan (Ohio State University)
- Zoran Budimlic, Vincent Cave, Alina Sbirlea, Dragos Sbirlea (Rice University)
- Michael Gill, Hui Huang, Louis-Noel Pouchet, Peng Zhang (UCLA)

Status The CDSC Mapper is currently in its first Beta release. Please provide feedback and possible improvements at `cdsc-mapper-users@cdsc.ucla.edu`.

License The CDSC Mapper is released under a standard 3-clause BSD license. The license can be found in the `LICENSE.txt` file.

Contents

Chapter 1

Overview

1.1 Introduction

The CDSC Mapper is a compiler package for heterogeneous mapping on various targets such as multi-core CPUs, GPUs and FPGAs. The objective is to provide the user with a complete compilation platform to ease the programming of complex heterogeneous devices, such as a Convey HC1-ex machine.

The architecture of the compiler is based on a collection of production-quality compilers such as GNU GCC, Nvidia GCC and LLVM; two open-source compilation infrastructures on top of which development has been performed: the LLNL ROSE compiler and the LLVM project; and a collection of research compilers and runtime such as CnC-HC, PolyOpt and SDSLc.

1.1.1 Programming Flow

The objective of the CDSC Mapper is to simplify the development of applications on heterogeneous devices. To achieve this goal, we use a multi-layer programming principle:

1. The application is decomposed into steps, which can be executed atomically on a given device. These steps can be implemented in a variety of sequential and parallel languages, such as C, C++, CUDA, or Habanero-C.

2. The coordination (control and data flow) between these steps is described through a CnC file, which is automatically translated into the Habanero-C parallel language, thereby exploiting a dynamic run-time system for parallel step execution.

3. Furthermore inside each step implementation, a domain-specific language (Stencil-DSL) is used to represent some compute-intensive part of the application. Target-specific compilers for this DSL are used to create highly-efficient execution of these program parts.

1.1.2 Compilation Modules

The main compiler modules developped for the CDSC Mapper are:

- CnC to HC translation
- CDSC-GR to HC translation
- HC compiler
- HC runtime

- SDSL to C translation
- SDSL-GPU domain-specific-target-specific compiler
- PolyOpt/C and PolyOpt/HLS

These compilers are orchestrated through the CDSC Mapper driver, as shown in Figure 1.1. It summarizes the various components included in the CDSC Mapper, as well as the various input languages supported.

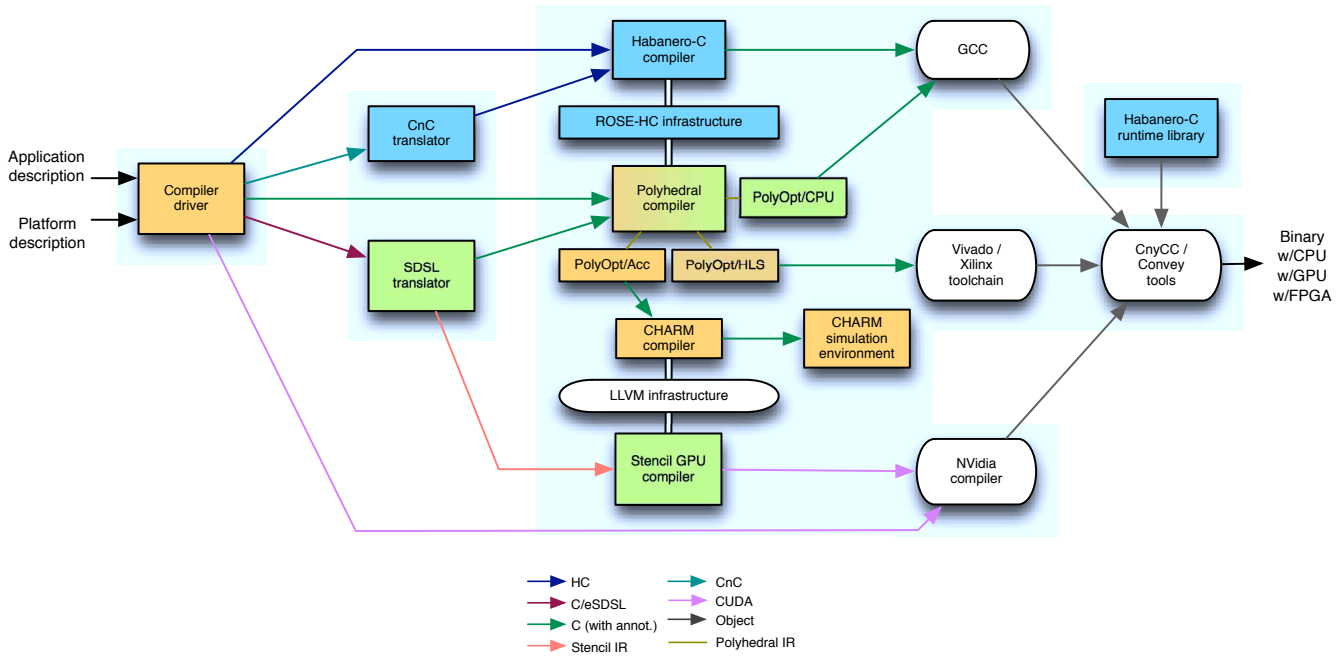

Figure 1.1: CDSC Mapper flow

1.2 Communication

Three mailing lists are available for communications.

- `cdsc-mapper-announces@cdsc.ucla.edu` for the major announcements about the CDSC Mapper, such as new releases, etc.
- `cdsc-mapper-users@cdsc.ucla.edu` for all questions regarding how to use the CDSC Mapper. Users are encouraged to post their questions on this list.
- `cdsc-mapper-dev@cdsc.ucla.edu` for all questions relating to developping inside/for the CDSC Mapper.

In addition, you can contact directly Louis-Noël Pouchet `pouchet@cs.ucla.edu`, the maintainer of the CDSC Mapper, in case of problems.

1.3 Running Example

All along this manual we illustrate the various components of the CDSC Mapper using the CDSC imaging pipeline application. This application was developed for the CDSC project and is aimed at performing the automatic analysis of CT scans for tumor detection. The application has four phases:

- First, a 3D image is reconstructed from the scan raw data. This is done using an EM+TV compressive sensing algorithm. The corresponding step is called `emtv`.
- Then, a Rician denoising and deblurring of the image is computed. Denoising reduces the noise that is introduced by the medical scanner, and may involve ambient/environmental factors. The type of noise is dependent on the type of scan (e.g., noise from an magnetic resonance (MR) imaging study is given by a Rician model; for a computed tomography (CT) study, noise usually follows a Poisson distribution). Deblurring sharpens the image, thereby handling imaging artifacts due to motion (e.g., the patient moving in the scanner; or due to normal physiologic function, such as breathing or cardiac motion). The corresponding step is called `denoise`.
- Then, registration is applied. Registration aligns the input image with respect to a reference image. This process further helps to compensate for patient breathing/movement during the scan and overall alignment to compare two different studies together. The corresponding step is called `registration`.
- Finally, segmentation is applied. Segmentation finds areas or regions of interest (ROIs). This step groups together pixels (voxels) that are related (e.g., an anatomical region; a tumor; etc.). The corresponding step is called `segmentation`.

This flow is the driver application for the CDSC project, which illustrates domain-specific computing by focusing on medical imaging. It fits perfectly well our proposed programming model, as the flow of data between these various steps can be exactly captured using a CnC representation, as illustrated in this manual.

The CDSC imaging pipeline can be downloaded from:

```
http://cadlab.cs.ucla.edu/mapper
```

Chapter 2

Installation

2.1 Machine Requirements

To build the CDSC Mapper toolchain, you need:

1. A x86-64 Linux machine. That is,
    - A 64-bit processor implementing a x86 instruction set
    - A standard Linux operating system, also in 64-bit mode (i.e., Debian, Fedora, etc.)
2. GNU GCC and G++, any version between 4.1 and 4.4 included, as the system default `gcc` and `g++` binaries.
3. A collection of software, including Java JDK, automake, autoconf, libtool, libxml2, libxml2-dev, ghostscript, texinfo, subversion, git. See Sec. 2.3.1 for more details.

In addition, to compile programs using the CDSC Mapper, we recommend:

- One or more GPU card(s)
- A GPU compilation toolchain supporting CUDA
- One or more multi-core CPU(s) with SIMD support
- A working optimizing C compiler for the CPUs (i.e., GNU GCC, Intel ICC, etc).

2.2 Download

The CDSC Mapper core installation files can be downloaded from:

`http://cadlab.cs.ucla.edu/mapper`

Please note that during the installation process numerous other software packages will also be downloaded, so you need a persistent internet connection during the install process. Once built, **the CDSC Mapper takes about 6GB of space on your hard drive**.

2.3 Install

2.3.1 Prerequesites

The CDSC Mapper requires numerous software packages to be installed on your Linux box before you start the installation.

GNU GCC and G++ version 4.1 to 4.4 It is required that the default GNU compiler collection installed on the system is one of GCC 4.1, GCC 4.2, GCC 4.3 or GCC 4.4. At this stage of development our customized version of ROSE requires precisely one of these versions to be installed.

For instance, on Ubuntu 12.04, the following commands will install GCC 4.4 and make it the default compiler, in place of GCC 4.6 (the default version on current Ubuntu). Note you can always restore the default GCC to a version of your choice once the mapper has been installed.

```
$> sudo apt-get install g++ gcc-4.4 g++-4.4
$> sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.6 100
$> sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.4 50
$> sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.6 100
$> sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.4 50
$> sudo update-alternatives --install /usr/bin/cpp cpp-bin /usr/bin/cpp-4.6 100
$> sudo update-alternatives --install /usr/bin/cpp cpp-bin /usr/bin/cpp-4.4 50
$> sudo update-alternatives --set g++ /usr/bin/g++-4.4
$> sudo update-alternatives --set gcc /usr/bin/gcc-4.4
$> sudo update-alternatives --set cpp-bin /usr/bin/cpp-4.4
```

Java JDK version 1.6 or above It is also required to have a working version of the Java JDK, version 1.6 or above. You must also correctly set the environment variable `JAVA_HOME` to where this SDK is installed.

The Sun Java SDK version 1.7 is preferred (but not required). To install it on your machine on your local account, you can for instance go to the Oracle website `http://www.oracle.com/technetwork/java/javase/downloads/index.html` and download the JDK for Linux-64 (for instance `jdk-7u17-linux-x64.tar.gz`). To install, simply do:

```
$> tar xzf jdk-7u17-linux-x64.tar.gz
$> cd jdk-7u17-linux-x64
$> export JAVA_HOME=`pwd`
$> export PATH=`pwd`/bin:$PATH
```

You need to export those two variables systematically in your shell environment before you use the CDSC Mapper.

Alternatively, you can use the OpenJDK. For instance, on Ubuntu 12.04, the following commands can achieve this setup:

```
$> sudo apt-get install openjdk-6-jdk
$> export JAVA_HOME=/usr/lib/jvm/java-6-openjdk-amd64
```

Other software packages Finally, numerous standard software for open-source project development are required. These are:

- Automake, Autoconf and Libtool (required)
- Make (required)
- LibXML2 and LibXML2-dev (for the development headers) (required)
- Ant (should be optional)
- Flex and Bison (should be optional)
- Ghostscript, Texinfo, Doxygen (required)
- Subversion, Git (should be optional)

For instance, on Ubuntu 12.04, the following command install these packages:

```
$> sudo apt-get install flex bison libxml2 libxml2-dev autoconf automake libtool subversion git \
 doxygen texinfo
```

You also need to have the directory `/usr/include/sys` existing on your machine. It is the case with most Linux distribution already, but if this directory does not exist on your machine you need to create it as follows.

```
$> sudo ln -s /usr/include/x86_64-linux-gnu/sys /usr/include/sys
```

2.3.2 Building the CDSC Mapper

To build the CDSC Mapper, during the whole installation process (it takes several hours), you need:

- At least 6GB of free space on your hard drive.
- A permanent internet connection during the install process.

Indeed, numerous packages are downloaded during the course of the installation process, therefore an internet connection is required during the install. However this is not needed anymore once the mapper has been built.

To build the mapper, simply do the following:

```
$> tar xzf cdsc-mapper-1.1-pre.tar.gz
$> cd cdsc-mapper-1.1-pre
$> ./install.sh
$> export PATH=`pwd`/driver/scripts:$PATH
```

It should complete by displaying `[CDSCmp-make] Done` followed by a message about setting up the `CUDA_HOME` environment variable. If this is not the case, then go to Section 2.4 for troubleshooting your installation process.

2.3.3 Post-Installation Tasks

If the mapper built correctly, you can perform the following post-install tasks:

1. Restore your system default GCC/G++ version
2. Remove the link to `/usr/include/sys` you created
3. Export permanently the `CUDA_HOME`, `JAVA_HOME` and `PATH` variables.

Restoring the system It is now safe to restore the default GCC version on your system to a version of your choice. Assumung you have used the `update-alternative` commands shown above, to restore your system, do:

```
$> sudo update-alternatives --auto g++
$> sudo update-alternatives --auto gcc
$> sudo update-alternatives --auto cpp-bin
$> sudo rm /usr/include/sys
```

Exporting environment variables Finally, as mentioned a few environment variables must be set by the user for the mapper to correctly operate on the machine.

1. `PATH` must be extended with `/path-to-mapper-dir/driver/scripts`
2. `JAVA_HOME` must be set to the install path of the Java JDK
3. `PATH` must be extended with `$JAVA_HOME/bin`
4. Optionally, if you have GPU card(s) installed, `CUDA_HOME` should be set to point to the root directory where the CUDA SDK is installed on your machine. And you should extend `PATH` with `$CUDA_HOME/bin` so that `nvcc` can be run.

2.4 Troubleshoot

Compiling the CDSC Mapper is a long and tedious task. In case of a failure, restarting the entire process can result in hours of compilation time being wasted. We provide developers tools to re-build only parts of the mapper, as well as a log file to monitor which modules have been successfully installed so far.

To inspect which modules have been successfully built, use the following command. We show here a log file resulting in a successful build of the CDSC Mapper.

```
$> cat driver/config/install_report.log
boost-1.44
libtool-2.2.6b
apache-ant-1.8.4
rose-hc
habanero-c-cdsc
cnc-c-cdsc
habanero-c-cdsc
rose-hc
hc-cdsc
cdscgr
polyopt-hls
```

If you encountered an error when building a module, the whole compilation process stops at this module, and you must fix the problem and resume the install. A wild approach is to fix the problem (i.e., a missing software dependence) and re-run `install.sh`, but a faster approach is to simply (re-)build the failing module and the missing ones. For instance, say the module `llvm-3.1` failed to compile. You have found and fixed the problem, and you now want to resume the CDSC Mapper build process. You would do:

```
$> ./driver/scripts/cdscmp-util --make llvm-3.1
$> ./driver/scripts/cdscmp-util --make sdslc
$> ./driver/scripts/cdscmp-util --make autoconf-2.60
$> ./driver/scripts/cdscmp-util --make polyopt-hls
```

In general, if you have a problem with the compilation of a module, such as because some software dependence is missing at start, to fix the problem you shall:

1. Install the missing software on your system.
2. Erase the source of the problematic module (say, `mod`) to be on the safe side (modules are located in the `higherlevel`, `highlevel` and `external` directories),
3. Do `cdscmp-util --checkout mod`
4. Do `cdscmp-util --make mod`
5. For all remaining modules, to the corresponding `cdscmp-util --make` command.

Chapter 3

Input Files

3.1 Overview

The CDSC Mapper allows the user to compile a full project in a single command line. To achieve this, a collection of files are required:

1. A project description file (text file), listing the various source files and other information, required to capture the compilation flow. See Section 3.2.

2. A machine description file (text/XML file), listing the various features of the target machine, required to capture the compilation targets. See Section 3.3.

3. A CnC file (text file), describing the dataflow between computation steps. See Section 3.4.

4. A list of source files, implementing each step in the CnC file above. Numerous input languages are allowed for these implementation files (C, C++, CUDA, Habanero-C, etc). See Section 3.5.

Technically, those files are required only if using the automatic compilation mode of the CDSC Mapper, which takes care of determining the targets and invoking all compiler modules with the correct parameters. It is also possible to use each compiler pass individually instead, in that case only the file(s) required for this pass is(are) required.

3.2 The Project Description File

In order to instruct the compiler about the location and type of the various files composing a full project, it is required to create a *project description file*. In essence, it is a text file which lists some key features such as:

- The CnC file which describes the dataflow between steps.

- The location of the source file(s) for each of the steps mentioned in the CnC file.

- Some basic information about the output binary name, and optionally the compilers to be used for compiling C files and linking the project object files.

- The machine description file, which is used to determine for which target(s) (such as multi-core CPU, GPU, or FPGA) the steps will be compiled.

The file format to be used is plain text. Lines starting by # are ignored, and sections (starting by a tag `[somename]`) can be provided in any order. Fields in a section (starting by `fieldname:`) can also be provided in any order. Inside a field, name/attributes/values are separated using the `:` character.

Figure 3.1 provides an example of such a file, named `project.desc`. It corresponds to the project description file for the CDSC imaging pipeline. This pipeline is described in Chapter 1. It is composed of four steps: *emtv*, *denoise*, *segmentation* and *registration*. The dataflow between those four steps is described in the file `pipeline.cnc`.

```
1  ## Describe here the CnC main file information.
2  [cnc-description]
3  project: cnc: pipeline.cnc
4
5
6  ## Describe here the step file(s). Supported types: c, sdsl.c, cu, hc.
7  [steps]
8  emtv: c: emtv_cpu/*.c
9  denoise: sdsl.c: denoise_step.c
10 segmentation: sdsl.c: segmentation_step.c
11 registration: sdsl.c: registration_step.c
12
13
14 ## Describe here the output binary info
15 [build-info]
16 output-bin: mi_pipeline
17 c-compiler: gcc
18
19 ## Describe here the machine description file.
20 [machine-desc]
21 desc: xml: cpugpu_system_description.xml
```

Figure 3.1: Sample `project.desc` file for the CDSC imaging pipeline

CnC file The section `[cnc-description]` is mandatory, and shall contain exactly one field, named `project`. This field contains a mandatory attribute which is the file type. The type of the file associated must be `cnc`. On line 2, we see that the CnC file used for this project is called `pipeline.cnc`.

Steps The section `[steps]` is mandatory, and shall contain exactly as many fields as there are steps declared in the CnC filed referenced on line 2. In addition, the name of each field (e.g., `emtv`, `denoise`, etc.) must match exactly the step name used in the CnC file. For each step, one must specify (1) the step name; (2) the step file format; and (3) the source file(s) for this step. The file format can be:

- `c`, for plain C files;
- `sdsl.c`, for C files embedding at least one Stencil-DSL region in them;
- `cu`, for plain CUDA files;
- `hc`, for plain Habanero-C files.

For the source file(s) implementing the step, there are two options:

- a single, stand-alone source file, such as in lines 9, 10 and 11. It may include an arbitrary number of header files. Note that this is the only option available for `sdsl.c` and `cu` step files.
- A set of files, described using a wildcard, such as in line 8. Note that at this stage, this option is supported only for `c` and `hc` files.

Build information The section [build-info] is mandatory, and must contain at least one field: output-bin. This field set the name of the binary produced as a result of the entire compilation process. The other fields are optional, and are used to overrride the default compilers used at various stages. These optional fields are:

- c-compiler to set the binary called when compiling C files, and linking object files (default: gcc);
- cpp-compiler to set the binary called when compiling C++ files (default: g++);
- cuda-compiler to set the binary called when compiling CUDA files (default: nvcc).

Machine description The section [machine-desc] is mandatory, and must contain at least one field: desc. This field contains a mandatory attribute which is the file type. The type of the file associated must be xml. On line 21, we see that the XML machine description file used for this project is called cpugpu_system_description.xml. The details of this file are described below in Section 3.3.

3.3 The Machine Description File

The machine description file plays a double role for the CDSC Mapper. First, it is used to determine which compilation passes have to be invoked. Second, it is used at run-time, when executing the produced binary, to let the Habanero-C runtime adequately use all hardware resources listed in this file when executing the various steps.

Figure 3.2 shows an example file which describes a machine with CPU and GPU as available computing devices. The file is made of two parts, both being mandatory, as described below. The whole description must be enclosed in a unique<CHP> XML tag.

```
1  <?xml version="1.0"?>
2  <CHP>
3
4  <!-- Mandatory part 1: basic machine description -->
5  <component name="my Testing Machine!">
6
7  <!-- The instance sections describe the HW components and their numbers -->
8  <instance name="CPU" component="CPU_system" count="2"></instance>
9  <instance name="GPU" component="GPU_system" count="1"></instance>
10
11 </component>
12
13
14 <!-- Mandatory part 2: The HPT description of the full system -->
15 <HPT version="0.1" info="an HPT for fornax, including the GPU">
16 <place num="1" type="mem">
17  <place num="1" type="nvgpu" size="4G"> </place>
18  <place num="1" type="fpga" size="4G"> </place>
19  <worker num="2"/>
20 </place>
21 </HPT>
22 </CHP>
```

Figure 3.2: Sample machine description file cpugpu_system_description.xml

System description The first part of this XML file describes, at a high level, the system topology. The format is free, and the user can provide any information in this file. For correct support of the CDSC Mapper capability to determine which compilation pass(es) to invoke, the <component> section is required. Three targets are possible, and for each of them a specific XML tag must be put somewhere in the file.

- To enable CPU support (mandatory for all projects), add the tag:

  ```
  <instance name="CPU" component="CPU_system" count="2"></instance>
  ```

  The instance name must be exactly `CPU`. Other attributes can have any value.

- To enable GPU support, add the tag:

  ```
  <instance name="GPU" component="GPU_system" count="1"></instance>
  ```

  The instance name must be exactly `GPU`. Other attributes can have any value.

- To enable FPGA support, add the tag:

  ```
  <instance name="FPGA" component="FPGA_system" count="4"></instance>
  ```

  The instance name must be exactly `FPGA`. Other attributes can have any value.

HPT description The second part of this file repeats the description of the machine, specifically using Hierarchical Place Trees (HPT). This segment must be enclosed in a `<HPT>` XML tag, and is used to instruct the Habanero-C runtime about possible places available for computing. Tuning this file for best performance can be needed. However, a basic and working approach is to use a generic HPT file as the one described in Figure 3.2 (lines 15 to 21), with all possible targets (here CPU, GPU and FPGA). Note that it is possible to use a HPT file which describes computing resources not available on the machine where the binary is run, provided the CnC file does not reference those non-existing resources. We show below in Section 3.4 how to specify the hardware targets on which steps can be mapped.

3.4 The CnC Description File

The CDSC Mapper extensively rely on a CnC description of the application. Figure 3.3 shows a CnC file describing the CDSC imaging pipeline. A CnC file is typically made of five parts: (1) the *control collections* declaration; (2) the *item collections* declaration; (3) the *step prescriptions*; (4) the *input/output relations* between steps; and (5) the *environment* setup.

Control collections Let us first remember the CnC constructs. CnC graphs have three components: item collections, control collections and step collections. Item collections store data in the form of $< key, value >$ pairs, and these items are dynamic single assignment. Control collections are sets of tags and are each associated with one or more step collections. This means that adding a tag to a control collection will spawn one or more steps with that tag. Step collections group together computational kernels with the same functionality and parametrized with a tag.

In the pipeline example in Figure 3.3, lines 1-4 contain control collection declarations. For example, line 1 declares a control collection named `emtv_tag`, which contains tags which are single dimensional tuples with type int. Thus, any int value is a valid tag for control collection `emtv_tag`. In general, tags may be multidimensional, such as having a pair or a triplet, but each of their components has the type int.

Item collections The next five declarations (lines 6-10) are for item collections. Each line declares a collection of type `float*`, which means that each value in this collection has the type `float*`. For the CDSC imaging pipeline, this corresponds to a pointer to the image data, that is passed between filters.

```
1  < int [1] emtv_tag>;
2  < int [1] denoise_tag >;
3  < int [1] reg_tag >;
4  < int [1] seg_tag >;
5
6  [ float* emtv_input ];
7  [ float* emtv_output ];
8  [ float* denoise_output ];
9  [ float* registration_output ];
10 [ float* final_output ];
11
12 <emtv_tag>::(emtv@CPU=1);
13 <denoise_tag>::(denoise@CPU=1,GPU=2);
14 <reg_tag> :: (registration@CPU=2,GPU=1);
15 <seg_tag> ::(segmentation@CPU=2,GPU=1);
16
17 [emtv_input : p] -> (emtv : p) -> [emtv_output : p];
18 [emtv_output : s]-> (denoise : s) -> [ denoise_output : s];
19 [denoise_output : k]-> ( registration : k ) -> [ registration_output : k ];
20 [registration_output : k] -> (segmentation : k)->[ final_output : k ];
21
22 env -> [emtv_input : {1 .. 3} ], <emtv_tag : {1 .. 3}>, \
23        <denoise_tag : {1 .. 3} >, <reg_tag : {1 .. 3}>, <seg_tag : {1 .. 3}>;
24 [final_output : {1 .. 3}] -> env;
```

Figure 3.3: Sample `pipeline.cnc` file for the CDSC imaging pipeline

Step prescriptions Lines 12-15 are step prescriptions, which means they define which control collection will start which step(s). In this example, line 12 defines a control collection (`emtv_tag`) which prescribes the `emtv` step. The next lines (13-15) define three other prescription relation, each for a different step. The CnC semantics impose that a step be prescribed by only one control collection. On each prescription relation, a step may specify its affinity with one or more devices. For example, on line 12, step `emtv` is defined to have affinity equal to 1 with the CPU. This implies that step `emtv` cannot run anywhere else except on the CPU, no matter what has been defined in the machine description file. The user will thus provide the appropriate code for running the `emtv` computation on a CPU, while the CnC runtime will handle the assignment of the step to the appropriate device. On line 13, step `denoise` is defined to have affinities with the CPU and the GPU, with a higher affinity with the CPU. In this situation, the CnC runtime will favor running `denoise` on the GPU if the device is found to be available and there are no requests with higher affinities to use the device. Apart from the affinity information, there must be a version of the `denoise` step for the CPU and for the GPU, either automatically generated by the CDSC Mapper or provided directly by the user. Then, we rely on the CnC runtime to provide the best overall application performance based on the machine load and the step affinity values. If no affinities are defined for a step, it is assumed the step can only run on the CPU.

Input/output relations Lines 17-20 describe input-output relations for each step. For example, on line 18, step `denoise` with some tag s is defined to read from item collection `emtv_output` an item with the key=s, and to write into item collection `denoise_output` an item with key=s.

Environment setup Lines 22-24 describe input-ouput relations for the environment. The writes from the environment are performed in order to provide the initial data and start the graph execution, while the reads from the environment are performed in order to get the results of the graph computation once this has finished. For example, line 22 specifies that the environment writes an item with keys 1 to 3 into item collection `emtv_input`, a range of tags with values from 1 to 3 into the control collection `emtv_tag` (thus starting 3 instances of the `emtv` step). On line 24, the environment is defined to read from item collection `final_output`, 2 items with tags ranging from 1 to 3 (exclusive), once the execution of the graph has terminated.

3.4.1 Data Allocation for Item Collections

The CnC-HC toolchain automatically handles the allocation and deallocation of the collections. However, the user is responsible for providing functions to allocate the item data produced by a step. These user-defined functions are automatically called in the HC codes generated from the CnC specification; and the user must provide an implementation of these functions, typically in the corresponding step source code. The prototype of those function is systematically as follows:

```
void stepname_outputXX_userDefined(item_collection_type*);
```

Where `XX` goes from 1 to the number of item collections written by the step. Example of such functions are described below.

In addition, the function:

```
void useGraphOutputs(item_collection_type*);
```

Needs to be implemented. This function performs any action on the final data written to the environment. This function may contain no useful code, but needs to be implemented as the automated CnC-HC compilation chain will link against this function.

3.5 Source Code for Steps

With the CDSC Mapper, in order to achieve automation several rules need to be carefully followed. These relate to:

- the prototype of the entry point function of each step;
- the various user-defined functions to allocate items in a collection.

We detail below what are those requirements.

3.5.1 Common Requirements for All Steps

Step entry point Each step must define an entry function, which is called when the step is being run on a particular target. The function prototype must be as follows.

```
void stepname_target(int tag, item_collection_type* input1, ...,
                              item_collection_type* output1, ...);
```

For instance, in the CDSC imaging pipeline, the prototype of the `emtv` step is:

```
void emtv_cpu(int , float* , float* );
```

The various elements of this prototype are:

- `stepname`: the name of the step, precisely matching the name used in the CnC file for the step declaration. For instance it can be `denoise`, `segmentation`, etc. for the CDSC imaging pipeline.
- `target`: it must be one of `cpu`, `gpu`, `fpga` or `sdsl`. That is, for instance the user can provide an implementation of the step `denoise` for CPUs and another implementation for GPUs (for instance in CUDA). The CPU implementation must be named `denoise_cpu` and the GPU implementation must be named `denoise_gpu`. The `sdsl` target is described in Section 3.5.4.
- The arguments of the function are as follows. First, an integer corresponding to the tag value will be passed to this function. The user may decide to ignore this value without if not needed. Then, the set of all input data to the step are passed. Finally, the set of all output data is passed. Output data is allocated through a special function as described below.

Data allocation functions As mentioned above, each step is responsible for providing one item allocation function for each of its input collection. This function has the following prototype:

```
void stepname_outputXX_userDefined(item_collection_type*);
```

For instance, for the CDSC imaging pipeline, the following function is defined for `emtv` output handling:

```
void
emtv_output1_userDefined(float** output)
{
  *output = (float*) calloc (128 * 128 * 128, sizeof(float));
}
```

The various elements of this prototype are:

- `stepname`: the name of the step, precisely matching the name used in the CnC file for the step declaration. For instance it can be `denoise`, `segmentation`, etc. for the CDSC imaging pipeline.
- `outputXX`: `XX` corresponds to the output collection count in order of appearance in the CnC program. If there is a single output produced by a step, then it is `output1`.
- `item_collection_type`: the type of the data elements in the corresponding collection, as declared in the CnC file.

We remark that the user is free to implement any code in those functions. In particular, debugging code or file I/O can take place in those functions. We also note that those functions will be executed on the same target as the Habanero-C runtime, that is x86/CPU code. Therefore the implementation must use the CPU's address space.

3.5.2 Step source: regular C

Once the entry point of a step has been implemented according to the above rules for its prototype, and once the various data allocation functions have been prepared, the user is free to provide any implementation for the step. It can use an arbitrary number of functions, header files, etc. The only requirement for regular C steps is of course for the file to be compilable with a standard C compiler, such as GCC.

3.5.3 Step source: regular C++

Steps can also be in C++, and strictly the same requirements as for regular C steps apply, except that the step code must be compilable with G++. For best interoperability, we encourage the user to declare the step entry point and other data allocation functions using the `extern "C"` qualifier. For instance, for `denoise` in the CDSC imaging pipeline, the full implementation of the output data allocation function is:

```
extern "C"
void
denoise_output1_userDefined(float** img_output)
{
  *img_output = (float*)malloc(sizeof(float)*(128*128*128));
}
```

3.5.4 Step source: Stencil-DSL embedded in C

The CDSC Mapper offers the possibility to embed fragments of a domain-specific language for stencils (SDSL) inside C/C++ step implementations. In that case, the fragments will be automatically compiled for various targets (CPU, GPU, FPGA) by using domain-specific and target-specific compilers.

More information about SDSL can be found in the package documentation: `highlevel/sdsl-compiler/doc`. In the CDSC imaging pipeline, three steps are implemented using SDSL: `denoise`, `registration` and `segmentation`.

The same requirements as above regarding the various functions to implement applies, except regarding the name of the entry point function. As using SDSL enable automatic mapping from a single file to various target architectures, the CDSC Mapper takes care of producing the various target-specific implementations (CPU, GPU, FPGA) of the step code. Therefore, instead of being called `stepname_target`, the step entry point must be named `stepname_sdsl`. The `sdsl` part will be automatically substituted by the appropriate target name (e.g., `cpu`, `gpu` or `fpga`).

For instance, in the CDSC imaging pipeline the prototype of the `denoise` step, written in SDSL, is:

```
extern "C"
void
denoise_sdsl(int ImageID, float *F, float* U);
```

Where, as usual, the first argument is the tag, the second one is the input data element, the third one the output data element.

3.5.5 Step source: Habanero-C

The CDSC Mapper offers the possibility to use Habanero-C to implement a step. The same requirements as above regarding the various functions to implement applies.

3.5.6 Step source: CUDA

The CDSC Mapper offers the possibility to use CUDA to implement a step. The same requirements as above regarding the various functions to implement applies. Using `extern "C"` is a requirement for each of these functions.

3.5.7 Step source: Multiple Files

The CDSC Mapper offers the possibility to use multiple files to implement a step. The same requirements as above regarding the various functions to implement applies. In addition, the compilation of all these files must work when they are all passed at the same time as argument of a compiler. For instance, if the step `foo` is implemented through the files `foo1.c` and `foo2.c`, then the following compilation line should be valid to compile your project:

```
$> gcc -c foo1.c foo2.c
```

3.6 Troubleshoot

Users are recommended to use the group `cdsc-mapper-users@cdsc.ucla.edu` for any question related to creating a project for the CDSC Mapper.

Typical issues to be investigated first are:

- Incorrect/incomplete CnC file.
- Incorrect prototype for the step function.

- Missing/incorrectly named data allocation functions.

These can be sorted out by looking at the various files generated by the CnC-to-HC translator, and in particular at the bottom of `Dispatch.h` which contains the prototype of the various functions that will be automatically called by the CnC-HC toolchain. All the functions placed after the `place_t` declarations will be called at some point by the run-time.

Chapter 4

Building A Project

In this chapter we review the basis of compiling a full project to a single binary that runs on an heterogeneous machine.

4.1 Assembling the Files into a Project

The previous chapter describes each of the various files that are required to create a project, using the CDSC imaging pipeline as a driving example. We recall here the various steps to be taken to create a project from scratch.

1. Create a machine description file. See Sec. 3.3. This file must reflect the machine on which the binary will be ran. Do not forget to fill in both the description and the HPT sections of the file.
2. Decompose the full application into steps, such that (1) each step can be executed atomically on a given target device (i.e., CPU, GPU, ...); and (2) the control and data flow between steps can be modeled throuh an item collection.
3. Create a CnC file that describes the control and data flow between steps. See Sec. 3.4.
4. Adjust the implementation of the various steps such that the entry point of each step is a function which conforms the specific prototype described in Sec. 3.5. Similarly, implement the allocation functions for items in the item collections used in the CnC file for the project.
5. Create a project description file, listing the location of the various files needed to build the project. See Sec. 3.2.
6. Compile the project using the `cdscmp-util --compile-proj` command, as described below.

Users are advised to pay particular attention to the various file names listed in the project description file. Indeed, currently there is no check implemented in the CDSC Mapper to ensure that all files are present, and a typo in a filename may result in an error only when the corresponding compiler is called much later in the process.

We also highly recommend to *put all files in a single directory*, which will be the project directory.

4.2 Running the Full Compiler

Once all the required files have been prepared, the CDSC Mapper has enough information to produce a Linux x86/64 binary program for the application. This binary will automatically use the HC runtime for effective execution on heterogeneous devices.

Assuming the project description file was named `project.desc` by the user, to create the binary (whose name is specified in `project.desc`) one needs to type the following command:

```
$> cdscmp-util --compile-proj project.desc
```

This command takes care of calling each pass of the CDSC Mapper, for each of the target architectures. This command is decomposed into three parts:

1. `cdscmp-util` is the name of the CDSC Mapper driver binary. This command is located in the `driver/scripts` sub-directory of the CDSC Mapper.

2. `--compile-proj` is the name of the particular mode of the CDSC Mapper where a project description file is given and an entire compilation process is to be run.

3. `project.desc` is the argument of the command, in this case the project description filename.

Of course, it is also possible to call each commands invoked one by one, instead of all at once. This is extremely convenient for debugging purposes, so as to avoid re-building the entire project each time a compilation error is encountered. Taking as an example the CDSC imaging pipeline, the command `cdscmp-util --compile-proj project.desc` is equivalent to the following sequence of commands.

```
$> cdscmp-util --pass clean emtv denoise registration segmentation
$> cdscmp-util --pass cncc pipeline.cnc
$> cdscmp-util --pass hcc cnc emtv.hc denoise.hc registration.hc segmentation.hc
$> cdscmp-util --pass cc gcc emtv_cpu/*.c
$> cdscmp-util --pass sdsl-gpu nvcc denoise_step.c
$> cdscmp-util --pass sdsl-gpu nvcc registration_step.c
$> cdscmp-util --pass sdsl-gpu nvcc segmentation_step.c
$> cdscmp-util --pass sdsl-cpu-poly gcc denoise_step.c
$> cdscmp-util --pass sdsl-c-poly gcc registration_step.c
$> cdscmp-util --pass sdsl-cpu-poly gcc segmentation_step.c
$> cdscmp-util --pass link gcc mi_pipeline
```

Each of the CDSC Mapper available passes are described in Chapter 5. But in a nutshell, the set of commands above create a binary named `mi_pipeline`, with support for CPU and GPU, from (1) a CnC file named `pipeline.cnc`; (2) steps `denoise`, `registration` and `segmentation` implemented in a single file using C embedding SDSL regions; and (3) a step `emtv` which is implemented for CPUs only. One may note that the machine description file is not used here: its role during the compilation process is currently limited to determining which compiler passes to invoke. In the present case, the SDSL steps are compiled to both GPU and optimized CPU code, using the `sdsl-gpu` and `sdsl-cpu-poly` passes as described in the next chapter.

Finally, the generated binary shall be run also through the `cdscmp-util` command, since a collection of environment variables must have their value correctly set. At this stage, the machine description file is required otherwise the binary will default to an all-CPU execution. For instance, for the CDSC imaging pipeline the binary is run with the following command.

```
$> cdscmp-util --pass run ./mi_pipeline -hpt machine_description.xml
```

4.3 Troubleshoot

Users are recommended to use the group `cdsc-mapper-users@cdsc.ucla.edu` for any question related to creating a project for the CDSC Mapper.

Typical issues to be investigated first are:

- Missing file and/or incorrect project description file.

- Incorrect/incomplete CnC file.
- Incorrect prototype for the step function.
- Missing/incorrectly named data allocation functions.

Chapter 5

List of Passes and Options

This chapter covers the main passes of the CDSC Mapper. We use the following convention for the syntax:

- `<some stuff>` means that `some stuff` is a mandatory argument to the command.
- `[some stuff]` means that `some stuff` is an optional argument to the command.

5.1 The `clean` Pass: Cleaning Old Files

Syntax `cdscmp-util --pass clean [step names]`

Example for the CDSC imaging pipeline:

```
$> cdscmp-util --pass clean emtv denoise registration segmentation
```

Description This pass cleans all previously automatically generated files. It does not remove any user-provided file. The step names are optionals, when invoked without any step name then only common files are cleaned. The step names must conform exactly the names used in the CnC file for the project.

Cleaning all steps is required before any call to the CnC translator, otherwise errors may appear.

Output None.

Known limitations / problems None.

5.2 The `custom` Pass: Customized Command

Syntax `cdscmp-util --pass custom [commands]`

Example to list the environment variables:

```
$> cdscmp-util --pass custom env
```

Description This pass lets the user execute any command but using the environment of the CDSC Mapper. That is, all environment variables are set first before `commands` is invoked. This command is useful for debugging.

Output None.

Known limitations / problems None.

5.3 The `cncc` Pass: CnC to HC Compilation

Syntax `cdscmp-util --pass cncc <cnc file>`

Example for the CDSC imaging pipeline:

```
$> cdscmp-util --pass cncc pipeline.cnc
```

Description This command runs the CnC-to-HC translator. See `higherlevel/cncc-cdsc/README` for more information. The input file must be a valid CnC file.

Output This command generates multiple files in an automatic fashion. For each of the step `thestep` declared in the CnC file passed as argument, a corresponding `thestep.hc` file will be generated. In addition, the following files are automatically generated:

- `Main.hc`: This is the main file for the project.
- `Common.hc` and `Common.h`: These contain HC code to manage the steps.
- `Dispatch.hc` and `Dispatch.h`: These contain also HC code to manage the steps. `Dispatch.h` lists the C functions that are called when a step is being invoked by the run-time.
- `Context.h`: This file declares the item collections.

Known limitations / problems Before invoking the CnC translator, all previously generated files must be deleted, for instance using the `clean` pass described above. This imply that all user modifications of those files, if any, will be lost.

This is due to the `-full-auto` mode of `cncc_t` being used. To overcome this limitation and call the CnC translator in standard mode (therefore compatible with user modifications), do:

```
$> cdscmp-util --pass custom cncc_t pipeline.cnc
```

5.4 The `cdscgrhc` Pass: CDSC-GR to HC Compilation

Syntax `cdscmp-util --pass cdscgrhc <cdscgl file>`

Example for the CDSC imaging pipeline:

```
$> cdscmp-util --pass cdscgrhc pipeline.cnc
```

Description This command runs the CDSCGR-to-HC translator. See `higherlevel/cdscgr/README` for more information. The input file must be a valid CDSC-GR file.

Output This command generates multiple files in an automatic fashion. For each of the step `thestep` declared in the CnC file passed as argument, a corresponding `thestep.hc` file will be generated. In addition, the following files are automatically generated:

- `Main.hc`: This is the main file for the project.
- `Common.hc` and `Common.h`: These contain HC code to manage the steps.
- `Dispatch.hc` and `Dispatch.h`: These contain also HC code to manage the steps. `Dispatch.h` lists the C functions that are called when a step is being invoked by the run-time.
- `Context.h`: This file declares the item collections.

Known limitations / problems **Note: this pass is currently under active development. No guarantee is provided about usability and potential bugs.**

Before invoking the CDSCGR translator, all previously generated files must be deleted, for instance using the `clean` pass described above. This imply that all user modifications of those files, if any, will be lost.

This is due to the `-full-auto` mode of `cdscgltohc_t` being used. To overcome this limitation and call the CDSCGR translator in standard mode (therefore compatible with user modifications), do:

```
$> cdscmp-util --pass custom cdscgltohc_t pipeline.cdscgl
```

5.5 The hcc Pass: HC to C compilation

Syntax `cdscmp-util --pass hcc <mode> <somefile1.hc> [somefile2.hc]`

Example for the CDSC imaging pipeline:

```
$> cdscmp-util --pass hcc cnc segmentation.hc
```

Description This command compiles the HC file(s) passed as argument into compilable C file(s) that use the HC runtime. The `mode` argument is either `cnc` or `cdscgl`, depending on which software was used to produce the HC files. In doubt, use `cnc`. It also systematically compile the other HC files generated by the `cncc` pass (in `cnc` mode) or the `cdscgrhc` pass (in `cdscgl` mode), that is `Main.hc`, `Dispatch.hc` and `Common.hc`. Those files must be located under the current working directory. The default CPU compiler for the project is used to compile those files (GCC).

Output For each HC file `somefile.hc` passed as argument, and for `Main.hc`, `Dispatch.hc` and `Common.hc`, a file `somefile.o` is generated. The C file generated from the HC file is named `rose_somefile.c`.

Known limitations / problems None.

5.6 The `sdsl-cpu` Pass: SDSL to C compilation

Syntax `cdscmp-util --pass sdsl-cpu <cpu compiler> <somefile.c>`

Example for the CDSC imaging pipeline:

```
$> cdscmp-util --pass sdsl-cpu gcc segmentation_step.c
```

Description This command uses the Stencil-DSL (SDSL) translator to generate a clean, affine C code for each of the embedded SDSL segments in the source C file. Then, a CPU compiler is invoked to compile the generated file.

More information about the syntax of SDSL can be found in `highlevel/sdsl-compiler-0.2.2/docs`.

Output For a given input `somefile.c` file, which contain SDSL regions embedded in it, the following two files are generated:

- `somefile.c.cpu.cpp`: the translated source file (C or CPP) without any SDSL code in it.
- `somefile.c.cpu.o`: the object file after compilation with a C compiler.

Known limitations / problems Only Jacobi-like stencils (that is, using a temporary array to store the updated field after one time step) are supported at the moment. Seidel-like (directly updating a unique field at each time step) stencils support is unstable.

Also, the CPU compiler is assumed to have a GCC-like syntax. That is, flags such as `-O3` and `-fopenmp` will be used.

Other limitations are discussed in the documentation of this compiler.

5.7 The `sdsl-cpu-poly` Pass: SDSL to Optimized C compilation

Syntax `cdscmp-util --pass sdsl-cpu-poly <cpu compiler> <somefile.c>`

Example for the CDSC imaging pipeline:

```
$> cdscmp-util --pass sdsl-cpu-poly gcc segmentation_step.c
```

Description This command uses the Stencil-DSL (SDSL) translator to generate a clean, affine C code for each of the embedded SDSL segments in the source C file. Then, a polyhedral compilation engine (PolyOpt/C) is invoked on the file produced, to optimize for locality and parallelism the SDSL region that was translated to C. Finally, a CPU compiler is invoked to compile the generated file.

More information about the syntax of SDSL can be found in `highlevel/sdsl-compiler-0.2.2/docs`.

More information about PolyOpt/C can be found in `highlevel/polyopt-cdsc-0.1.0/doc`.

Output For a given input `somefile.c` file, which contain SDSL regions embedded in it, the following two files are generated:

- `somefile.cpu.c`: the translated source file (C) without any SDSL code in it.

- somefile.cpu.opt.c: the optimized source file (C) after polyhedral transformations.
- somefile.cpu.opt.o: the object file after compilation with a C compiler.

Known limitations / problems The same limitations as for the sdsl-cpu pass apply.

5.8 The sdsl-gpu Pass: SDSL to CUDA compilation

Syntax cdscmp-util --pass sdsl-gpu <cuda compiler> <somefile.c>

Example for the CDSC imaging pipeline:

```
$> cdscmp-util --pass sdsl-gpu nvcc segmentation_step.c
```

Description This command uses the Stencil-DSL (SDSL) translator to generate a CUDA implementation of the SDSL region(s), using overlapped tiling. Then, a CUDA compiler is invoked to compile the generated file.

More information about the syntax of SDSL can be found in highlevel/sdsl-compiler-0.2.2/docs.

Output For a given input somefile.c file, which contain SDSL regions embedded in it, the following two files are generated:

- somefile.c.cu: the generated CPP/CUDA source file, where SDSL regions have been translated to CUDA.
- somefile.c.gpu.o: the object file after compilation with a C compiler.

Known limitations / problems The same limitations as for the sdsl-cpu pass apply.

5.9 The sdsl-fpga Pass: SDSL to HLS-friendly C compilation

Syntax cdscmp-util --pass sdsl-fpga <somefile.c>

Example for the CDSC imaging pipeline:

```
$> cdscmp-util --pass sdsl-fpga segmentation_step.c
```

Description This command uses the Stencil-DSL (SDSL) translator to generate a clean, affine C code for each of the embedded SDSL segments in the source C file. Then, a polyhedral compilation engine (PolyOpt/HLS) is invoked on the file produced, to optimize for locality and parallelism the SDSL region that was translated to C.

This command only produces a C file suited for high-level synthesis purpose. The user is responsible for generating RTL, synthesizing the file, and creating a CPU wrapper to call the code.

More information about the syntax of SDSL can be found in highlevel/sdsl-compiler-0.2.2/docs.

More information about PolyOpt/C can be found in highlevel/polyopt-cdsc-0.1.0/doc.

Output This pass is an interactive command to assist the user in creating a template file suited for HLS on the Convey HC1-ex machine. Then, calling again the same pass (choosing different options) trigger polyhedral optimizations and update the file passed as argument.

Known limitations / problems This pass works exclusively with Vivado HLS, for the Convey HC1-ex machine. In addition, the same limitations as for the `sdsl-cpu` pass apply.

5.10 The `gcc` Pass: Compiling Regular C files

Syntax `cdscmp-util --pass ccc <CPU compiler> <somefile1.c> [somefile2.c]`

Example for the CDSC imaging pipeline:

```
$> cdscmp-util --pass cc gcc emtv_cpu/*.c
```

Description This command compiles a (set of) C files with a CPU compiler.

Output For each C file passed as argument, a `.o` file is generated.

Known limitations / problems The CPU compiler is assumed to have a GCC-like syntax. That is, flags such as `-O3` and `-fopenmp` will be used.

5.11 The `link` Pass: Linking all Objects

Syntax `cdscmp-util --pass link <linker command> <output_binary_name>`

Example for the CDSC imaging pipeline:

```
$> cdscmp-util --pass link gcc mi_pipeline
```

Description This command links all `.o` files in the current directory into a binary `output_binary_name`. In addition, it links against the Habanero-C runtime.

Output A Linux x86/64 binary `output_binary_name` is created.

Known limitations / problems None.

5.12 The `run` Pass: Running the Generated Binary

Syntax `cdscmp-util --pass run <somebinary> [arguments]`

Example for the CDSC imaging pipeline:

```
$> cdscmp-util --pass run ./mi_pipeline -hpt machine_description.xml
```

Description This command runs the binary `somebinary` with optional arguments. It is required to use this command to run the generated binary as the environment has to be set for the binary to dynamically load the Habanero-C runtime.

The arguments are specific to the HC runtime, and are optional. However, passing the HPT file is highly recommend for best performance. The user can query the various options by running the binary with the argument `-help`.

Output None.

Known limitations / problems None.