THE SOLLVE OMPVV OFFLOADING VERIFICATION AND VALIDATION TEST SUITE HANDBOOK

Prepared by: Joshua Hoke Davis

Last Updated: March 2, 2021

ABSTRACT

This handbook contains documentation for the OpenMP Offloading Verification and Validation (OMPVV) test suite. Crucial to the success of the test suite is that the suite is easy to use and contribute to. Procedures for using the suite are covered in detail in this handbook, including setting up the suite, using the Makefile to compile and run tests and generate reports of results. Also covered are procedures for adding new tests to the suite, reviewing tests, submitting issues, and the overall workflow of OMPVV. The intent of this handbook is to equip collaborators with a readable and useful reference for contributing, and to assist in onboarding new students as project developers.

TABLE OF CONTENTS

ABSTRACT **2**

TABLE OF CONTENTS **2**

LIST OF ILLUSTRATIONS **3**

INTRODUCTION **5**
- Purpose, Scope, and Limitations 5
- Methods of Research 5
- Report Organization 5

SETTING UP THE SUITE **6**
- System Preparation 6
- Cloning the Repository 7
- Modifying make.def 7
- Creating a system.def File For Your System 9

RUNNING THE SUITE **11**
- Directory Structure 11
- The Makefile 11

GENERATING REPORTS **15**
- Raw Format 15
- Reports for Post-processing 16
- Reports for Viewing 17

CREATING NEW TESTS **21**
- Project Workflow 21
- The OMPVV Header File 22
- Coding Standards 23

CONTRIBUTING TO THE SUITE **26**
Pull Requests 26
Issues 26
Vendor Bug Reports 27
APPENDIX A: Interview **28**
APPENDIX B: Changes from previous versions **29**
Changes from version 1.0 to version 1.1 29
WORKS CITED **30**

LIST OF ILLUSTRATIONS

Figure 1: Table of Popular OpenMP Offloading Compilers 6
Figure 2: An example of a set of compiler flags specified in the make.def file 7
Figure 3: Table of OMPVV test suite default flags used for supported flags 8
Figure 4: An example of a compiler module definition in summit.def 9
Figure 5: Select examples of Makefile commands for the test suite 11
Figure 6: All OMPVV Makefile options for compiling and running tests 12
Figure 7: All custom rules provided by the OMPVV Makefile 12
Figure 8: Example of output from running the test suite with the Makefile 13
Figure 9: Format of a log file segment header and footer 15
Figure 10: Example of a complete log file segment 16
Figure 11: Partial example of a JSON report file 16
Figure 12: Partial example of an HTML report file 17
Figure 13: Example of a report summary 17
Figure 14: Screenshot of an example HTML results report, viewed in Google Chrome 18
Figure 15: Example usage of the online report Makefile rule 19
Figure 16: OMPVV Makefile options for the online report rule 19
Figure 17: The OMPVV workflow for creating new tests 21
Figure 18: Table of OMPVV header file macros 22
Figure 19: Example of a specification-based test from the OMPVV suite 23
Figure 20: Steps to ensure your branch is up-to-date with master 26
Figure 21: Information requested in each category of Github issue 26

INTRODUCTION

Purpose, Scope, and Limitations

The purpose of this handbook is to make it easier to bring on new contributors to the OpenMP Verification and Validation (OMPVV) project and to engage interested researchers with the suite and enable them to to get the suite working on their machine. The scope of this document includes:

- Cloning the repository, building the test suite, and running the tests
- Processing results and creating a report
- Coding standards and format for contributing new tests
- Submitting issues and pull requests
- Submitting bugs to vendors
- Installing and building compilers for offload support
- Creating a definitions file for a new machine
- Documentation of makefile options with examples
- Documentation of macros provided by the test suite header file

Note: Some machines often used by the project, including Summit at Oak Ridge Leadership Computing Facility and those that are covered under non-disclosure agreements between Oak Ridge National Lab, the University of Delaware, and vendors, are not available for public use, so instructions for accessing these particular machines and submitting jobs to them will not be provided on public-facing documents.

Methods of Research

Sources for this handbook include interviews with the developers, personal experience of the author as a developer, and publically-available documentation on the project website and Github. Interviews with users are planned, and feedback from those interviews will be integrated in the future.

Report Organization

This report is organized generally in sequence from the most basic steps in getting started running the suite to contributing tests and filing bug reports. The first content section starts with preparing your system to run the test suite.

SETTING UP THE SUITE

System Preparation

Running the test suite first requires access to a machine with offloading hardware (i.e., a GPU or other accelerator that is distinct from the main CPU). At least one compiler that supports OpenMP offloading must also be installed on the system. Figure 1 below lists popular OpenMP offloading compilers, along with the hardware supported by that compiler and the latest release version as of May 2020. Compiler packages typically support C, C++ and Fortran.

Compiler Package	Invocation Names (C/C++/Fortran)	Vendor	Offload Hardware Supported	Latest Release (of May 2020)
GCC	`gcc, g++, gfortran`	GNU	Intel MIC, Nvidia PTX/CUDA, AMD HSAIL/GCN	10.1 (30-04-2020)
Clang	`clang, clang++`	LLVM	Nvidia CUDA (partial)	10.0.0 (24-04-2020)
XL	`xlc, xlc++, xlf`	IBM	Nvidia CUDA	16.1.1 (30-11-2019)
ICC	`icc, i++, ifort`	Intel	Intel Integrated Graphics	19.0.8 (6-4-2019)
AOMP	`clang, clang++`	AMD	AMD GCN	11.5-0 (30-04-2020)
CCE	`cc, CC, ftn`	Cray	Nvidia CUDA, AMD GCN	9.1.1 (19-12-2019)

Figure 1: Table of Popular OpenMP Offloading Compilers (Source: self-made)

Additionally, a Linux-based system with a BASH environment, as well as GNU git and Make are expected. For results reporting, Python 3 is required, and the online report feature requires either the “requests” package or Curl (“requests” is preferred). More details on results reporting and the online report feature can be found in the Generating Reports section of this handbook.

Steps for installing each of these compilers can be found on the vendors’ websites, and are outside of this handbook’s scope. More information on this topic can be found at https://crpl.cis.udel.edu/ompvvsollve/project/gettingstarted/.

Cloning the Repository

The test suite uses Git for version control. The following command will download the test suite repository to your system:

```
git clone https://github.com/SOLLVE/sollve_vv.git
```

Modifying make.def

The **`make.def`** file (found in the **`sys/make/`** directory of the repository) tells the OMPVV Makefile what flags to use for the compiler(s) you choose to run.

If you are using a recent version of one of the popular compilers listed in Figure 1 above, then it is possible you will not need to modify the provided make.def, as it already includes the needed flags for most compilers. However, even in this case it is wise to review the provided flags, as in some cases they will need to change based on the particular type of offloading device you are targeting. For example, the Cray and LLVM compilers require hardware-specific information like the “sm” number for Nvidia targets and the “gfx” number for AMD targets. Figure 2 is an example of a simple flag definition.

```
# CRAY compiler
ifeq ($(CC), cc)
  CFLAGS += -fopenmp -fopenmp-targets=nvptx64 -Xopenmp-target -march=sm_70
  CLINK = cc
  CLINKFLAGS += -fopenmp -fopenmp-targets=nvptx64 -Xopenmp-target -march=sm_70
endif
```

Figure 2: An example of a set of compiler flags specified in the make.def file (Source: https://crpl.cis.udel.edu/ompvvsollve)

Figure 2 above shows the basic format for adding flags to the make.def file. First, **`$(CC)`** is the language in question. This example is for a C compiler, so the symbol should be swapped out for **`$(CXX)`** or **`$(FC)`** for C++ or Fortran compilers. **`cc`** is the invocation name for the compiler, which depends on the compiler package being tested and the language, as seen in Figure 1 above.

`CFLAGS` is the list of flags for the compiler, **`CLINK`** is the linker to be used (typically the same as the compiler), and **`CLINKFLAGS`** is the list of flags for the linker (also typically the same as the flags for the compiler). These names vary slightly for different languages. For example, **`CFLAGS`** becomes **`CXXFLAGS`** for C++ and **`FFLAGS`** for Fortran.

More examples of definitions can be found in the provided **`make.def`**, as mentioned above. Figure 3, below, shows the standard flags the **`make.def`** already includes for each pre-supported compiler.

Compiler	**Language**	**Flags**
GNU `gcc`	C	`-O3 -std=c99 -fopenmp -foffload="-lm" -lm`
GNU `g++`	C++	`-O3 -std=c++11 -fopenmp -foffload="-lm" -lm`
GNU `gfortran`	Fortran	`-O3 -fopenmp -foffload="-lm" -lm -ffree-line-length-none -J./ompvv`
LLVM `clang`	C	`-lm -O3 -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda -D__NO_MATH_INLINES -U__SSE2_MATH__ -U__SSE_MATH__`
LLVM `clang++`	C++	`-std=c++11 -lm -O3 -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda -D__NO_MATH_INLINES -U__SSE2_MATH__ -U__SSE_MATH__`
IBM `xlc`	C	`-O3 -qsmp=omp -qoffload`
IBM `xlc++`	C++	`-O3 -qsmp=omp -qoffload`
IBM `xlf`	Fortran	`-O3 -qsmp=omp -qoffload -qmoddir=./ompvv -DEXIT=EXIT_`
Cray `cc`	C	`-homp -O3 -lm`
Cray `CC`	C++	`-homp -O3 -lm`
AMD AOMP `clang`	C	`-lm -O3 -fopenmp -target $(AOMP CPUTARGET) -fopenmptargets=$(AOMP GPUTARGET) -Xopenmp-target=$(AOMP GPUTARGET) -march=$(AOMP GPU) -D__NO_MATH_INLINES -U__SSE2_MATH__ -U__SSE_MATH__`
AMD AOMP `clang++`	C++	`-std=c++11 -lm -O3 -fopenmp -target $(AOMP CPUTARGET) -fopenmptargets=$(AOMP GPUTARGET) -Xopenmp-target=$(AOMP GPUTARGET) -march=$(AOMP GPU) -D__NO_MATH_INLINES -U__SSE2_MATH__ -U__SSE_MATH__`

Figure 3: Table of OMPVV test suite default flags used for supported flags (Source: "A Test Suite Design And Implementation For Openmp 4.5 Offloading Features", Jose Diaz (thesis))

Creating a system.def File For Your System

For systems that use Environment Modules, the test suite is designed to automatically swap out the modules needed to load a new compiler. This helps to guarantee that the correct compiler is being used, and allows the user to avoid having to modify environment variables repeatedly when they wish to test multiple compilers. The test suite infrastructure uses custom system definition files to know which modules to load for a given compiler. These files also contain information needed to interact with the job scheduler, if needed. Each definitions file is named after the system name, followed by a **`.def`**.

If you plan to test multiple compilers on your system, or you are running on a system using a job scheduler (such as Slurm), it is recommended that you create a unique **`.def`** file for your system. System definition files are stored in **`sys/systems/`**. In order to use a **`.def`** file for compiler swapping, you will need to have installed your compilers using Environment Modules (http://modules.sourceforge.net/). A **`.def`** file is already provided in the directory for most systems the suite is regularly tested on (for example, for Summit a file called **`summit.def`** is provided).

To assist with creating your own system definitions file, a template called **`generic.def`** is also provided, which should be copied to a file called **`<your system name>.def`** and modified to make the process easier. Figure 4 is an example of a module specification for a compiler, taken from the Summit definitions file.

```
# GCC compiler
ifeq ($(CC), gcc)
  C_COMPILER_MODULE = gcc/8.1.1
  C_VERSION = gcc -dumpversion
endif
```

Figure 4: An example of a compiler module definition in summit.def (Source: https://crpl.cis.udel.edu/ompvvsollve)

As seen in **`make.def`**, the **`$(CC)`** variable and the prefixes of the variables contained in the definition vary by language, and **`gcc`** is the invocation name for the compiler in question. **`C_COMPILER_MODULE`** should be set to the name of the exact module needed for the compiler to load. To view available modules for a given compiler, run "**`module avail `**" followed by the compiler name in a bash shell. If more than one module is needed in order to load your compiler, then add the modules after the first, separated by "**`; module load `**". **`C_VERSION`** should be set to a one-line command to print the version of the compiler (needed for result reporting).

At the top of the file, the variable **CUDA_MODULE** should also be set to your system's CUDA module, and if needed, the batch scheduler command should be provided to the

BATCH_SCHEDULER variable. Once again, more detailed examples can be found in the provided system definitions files.

Once your system is fully configured, you can run the test and root out any mistakes using the Makefile, as described in the next section of this handbook.

RUNNING THE SUITE

Directory Structure

The Git repository for the project is structured into the following folders:

- **`ompvv`**: Contains the project header files and static library
- **`sys`**: Contains .def files and reporting scripts, grouped into the following subfolders:
    - **make**: Contains the make.def file for compiler flags
    - **results_template**: Contains the html template used for results reporting
    - **scripts**: Contains scripts used for running tests and generating results
    - **sys**: Contains system definition files for module loading
- **`template`**: Contains test templates in C and Fortran
- **`tests`**: Contains tests files, organized by specification version and then by category. Each test starts with **`test_`**, except for those filed under **`application_kernels`**.

After compiling and generating results reports, the following additional folders will be created:

- **`bin`**: Contains the compiled binary files of the tests.
- **`logs`**: Contains the raw format log files if the LOG option was specified.
- **`results_report`**: Created when the make rule **`report_html`** is used, contains an html version of the results

The Makefile

The test suite uses a Makefile to manage building and running all tests, which should be run from the root of the project repository. Figure 5 below lists a few examples of Makefile commands and a full list of available options. In general, it is a good idea to run either '**`make clean`**' (if you want to keep old logs) or '**`make tidy`**' (if you want to start with a completely clean slate) before compiling any tests.

```
make CC=gcc CXX=g++ FC=gfortran all       # compile and run all test cases with GCC
make CC=gcc SOURCES=a.c all               # compile and run a.c with gcc
make CXX=g++ SOURCES=tests/target/* all   # compile and run all tests/target tests
```

Figure 5: Select examples of Makefile commands for the test suite (Source: https://crpl.cis.udel.edu/ompvvsollve)

To both run and compile tests, the '**`all`**' rule is used in Figure 4. If you want only to compile, use the '**`compile`**' rule, and if you want only to run tests, use the '**`run`**' rule (this runs all already-compiled binaries in the **`bin`** directory, so the sources flag should not be used). Figure 6 (below on page 12) lists all options available for compiling.

Option	Description
`SOURCES=[file or path]`	Specify a file to run. Supports wildcard (*) and can be used with or without the path to the test
`CC=[compiler]`	Specify a C compiler (e.g., gcc or clang)
`CXX=[compiler]`	Specify a C++ compiler (e.g., g++ or clang++)
`FC=[compiler]`	Specify a Fortran compiler (e.g., gfortran or xlf)
`VERBOSE=1`	Enable extra output information from make process
`VERBOSE_TESTS=1`	Enable extra output information from tests
`LOG=1`	Enable dump of make process output into logs
`LOG_ALL=1`	Enable dump of errors and test execution output into logs
`OMP_VERSION=[4.5 or 5.0]`	Specify OpenMP specification version, defaults to 4.5
`NUM_THREADS_HOST=[n]`	Specify number of threads to use on the host
`NUM_THREADS_DEVICE=[n]`	Specify number of threads to use on the device
`NUM_TEAMS_HOST=[n]`	Specify number of teams to use on the device
`SYSTEM=[system name]`	Enable inclusion of the specified **`system.def`** file (do not include .def part)
`MODULE_LOAD=1`	Enable loading of modules (must also specify a system using SYSTEM=)
`ADD_BATCH_SCHED=1`	Enable execution using job scheduler command specified in **`make.def`**
`NO_OFFLOADING=1`	Disable offloading (experimental)

Figure 6: All OMPVV Makefile options for compiling and running tests (Source: self-made)

The Makefile also provides a number of additional rules, shown in Figure 7 below (extending to page 13), for cleaning out old logs and generating results reports. Result reporting is covered later in this handbook, but the options are included in the table for completion.

rule	Description
`all`	Compile and run all tests or tests specified with the SOURCES option
`compile`	Just compile all tests or tests specified with the SOURCES option
`run`	Just run the all previously-compiled tests or tests specified with the

	SOURCES option
clean	Remove all binaries, from the **bin** folder
tidy	Remove all binaries (**bin** folder), logs (**log** folder), and results reports
compilers	List currently-available compiler configuration
report_csv	Create a csv-formatted report from the logs in the **log** folder
report_json	Create a json-formatted report from the logs in the **log** folder
report_summary	Create a short summary report in the console listing tests that failed, from the logs in the **log** folder
report_html	Create a prettified report using the HTML template, from the logs in the **log** folder
report_online	Create an HTML report and upload to it to the OMPVV website for easier sharing (experimental)

Figure 7: All custom rules provided by the OMPVV Makefile (Source: self-made)

If your make command is successful, you should receive output something like that displayed in Figure 8 below.

Note: if many tests are specified the runs could take a few minutes to generate, and if ***VERBOSE_TESTS=1*** *is set, the output will be considerably longer than shown in Figure 8.*

```
====    SOLLVE PROJECT MAKEFILE   ====
Running make with the following compilers
CC = gcc 9.0.0
CXX = g++ 9.0.0


 compile: tests/offloading_success.c


 running: bin/offloading_success.c.run
/home/josem/Documents/Sunita/Projects/SOLLVE/sollve_vv/sys/scripts/run_test.sh
bin/offloading_success.c.o
offloading_success.c.o: PASS. exit code: 0
====COMPILE AND RUN DONE====
```

Figure 8: Example of output from running the test suite with the Makefile (Source: https://crpl.cis.udel.edu/ompvvsollve)

If you ran the make command with the **LOG=1** and **LOG_ALL=1** options, a **logs** folder will have been created, containing the logs of all tests run. In the next section, this handbook covers creating reports from those logs.

GENERATING REPORTS

Raw Format

The **`logs`** folder contains individual log files for each test run using the **`LOG=1`** and **`LOG_ALL=1`** Makefile options. Each file is divided into segments, one segment per run or compile operation. For example, if **`test_target_teams_distribute.c`** is run and compiled once with **`gcc`** and once with **`clang`**, there will be four segments in the file: **`gcc`** compile, **`gcc`** run, **`clang`** compile, and **`clang`** run.

Each segment has three parts, a header, the output, and a footer. The header and footer contain formatted information regarding that operation, as shown in Figure 9.

```
HEADER:
*-*-*BEGIN*-*-*COMPILE (command)/RUN*-*-*DATE (long format) *-*-*SYSTEM
NAME*-*-*SOURCE FILE TESTS*-*-*COMPILER VERSION/RUNTIME COMMENTS*-*-*GIT COMMIT*-*-*

FOOTER:
*-*-*END*-*-*COMPILE (command)/RUN*-*-*DATE (long format)*-*-*SYSTEM
NAME*-*-*PASS/FAIL*-*-*COMMENTS*-*-*GIT COMMIT*-*-*
```

Figure 9: Format of a log file segment header and footer (Source: https://crpl.cis.udel.edu/ompvvsollve)

The output segment, which is between the header and the footer, will contain all output printed to the console by the test. The format for this output is standardized by the OMPVV header file. The header file is covered in detail in the next section. Figure 10 (below on page 16) shows an example of a complete log file segment.

```
*-*-*BEGIN*-*-*COMPILE CC=gcc -I./ompvv -O3 -std=c99 -fopenmp -foffload=-lm -lm
*-*-*Wed May  9 19:15:47 EDT
2018*-*-**-*-*/home/josem/Documents/Sunita/Projects/SOLLVE/sollve_vv/tests/applicati
on_kernels/mmm_target.c*-*-*gcc 9.0.0*-*-*04e92c8*-*-*
*-*-*END*-*-*COMPILE CC=gcc -I./ompvv -O3 -std=c99 -fopenmp -foffload=-lm -lm
*-*-*Wed May  9 19:15:47 EDT 2018*-*-**-*-*PASS*-*-*none*-*-*04e92c8*-*-*

*-*-*BEGIN*-*-*RUN*-*-*Wed May  9 19:16:06 EDT
2018*-*-**-*-*bin/mmm_target.c*-*-*none*-*-**-*-*04e92c8*-*-*

 running: bin/mmm_target.c.run
mmm_target.c.o: PASS. exit code: 0
mmm_target.c.o:
Total time for A[500][500] X B[500][500] on device using target directive only:37
Test PASSED.
*-*-*END*-*-*RUN*-*-*Wed May  9 19:16:43 EDT
2018*-*-**-*-*PASS*-*-*none*-*-*04e92c8*-*-*
```

Figure 10: Example of a complete log file segment (Source: https://crpl.cis.udel.edu/ompvvsollve)

Reports for Post-processing

As shown in Figure 7 in the previous section, a number of Makefile rules are available for generating various types of reports. The first category of reports available are those that are useful for post-processing. These report rules condense all log files into a single intermediate format, either JSON and CSV.

Note: using these rules requires Python 3 or greater to be installed on your system.

To create a JSON report, use the rule '**`make report_json`**'. A JSON report is formatted as shown in Figure 11 (below, continued on page 17).

*Note: the file contains terminal coloring codes like '**`\u001b[0;32m`**' which result from the test suite Makefile's coloring features.*

```
[
    ...,
  {
    "Binary path": "bin/mmm_target.c",
    "Compiler command": "gcc -I./ompvv -O3 -std=c99 -fopenmp -foffload=-lm -lm ",
    "Compiler ending date": "Wed May  9 19:16:43 EDT 2018",
    "Compiler name": "gcc 9.0.0",
    "Compiler output": "",
    "Compiler result": "PASS",
    "Compiler starting date": "Wed May  9 19:15:47 EDT 2018",
    "Runtime ending date": "Wed May  9 19:16:43 EDT 2018",
    "Runtime only": false,
    "Runtime output": "\u001b[0;32m \n\n running: bin/mmm_target.c.run
```

```
\u001b[0m\nmmm_target.c.o: PASS. exit code: 0\n\u001b[0;31mmmm_target.c.o:\nTotal
time for A[500][500] X B[500][500] on device using target directive only:37 \nTest
PASSED.\u001b[0m\n",
    "Runtime result": "PASS",
    "Runtime starting date": "Wed May  9 19:16:06 EDT 2018",
    "Test comments": "none\n",
    "Test name": "mmm_target.c",
    "Test path":
"/home/josem/Documents/Sunita/Projects/SOLLVE/sollve_vv/tests/application_kernels/mm
m_target.c",
    "Test system": "",
    "Test gitCommit": "04e92c8"
  },...
]
```

Figure 11: Partial example of a JSON report file (Source: https://crpl.cis.udel.edu/ompvvsollve)

To create a CSV report, use the rule '**`make report_csv`**'. A CSV report is formatted as shown in Figure 12 below, and can be imported into applications like Microsoft Excel for viewing and manipulation.

```
testSystem, testName, testPath, compilerName,compilerCommand,
startingCompilerDate,endingCompilerDate, compilerPass, compilerOutput,runtimeOnly,
binaryPath, startingRuntimeDate,endingRuntimeDate, runtimePass, runtimeOutput,
gitCommit, testComments
"fatnode", "offloading_success.c", "tests/4.5/offloading_success.c", "gcc 9.0.0",
"gcc -I./ompvv -O3 -std=c99 -fopenmp -foffload=-lm -lm", "Mon Dec  2 19:09:35 EST
2019", "Mon Dec  2 19:09:36 EST 2019", "PASS", "", "False",
"bin/offloading_success.c", "Mon Dec  2 19:09:36 EST 2019", "Mon Dec  2 19:09:37 EST
2019", "PASS", "^[[0;32m  running: bin/offloading_success.c.run
^[[0moffloading_success.c.o: PASS. exit code: 0^[[0;31moffloading_success.c.o:Target
region executed on the device^[[0m", "04e92c8", "none"
```

Figure 12: Partial example of an HTML report file (Source: https://crpl.cis.udel.edu/ompvvsollve)

Reports for Viewing

While intermediate report formats are useful for feeding into visualization scripts, more often than not you will want to see the results immediately after running a batch of tests.

Note: Once again, using these rules requires Python 3 or greater to be installed on your system.

The quickest way to see the outcomes of your tests is with the rule '**`make report_summary`**', which provides a short in-console overview of the test results. It will look something like the example shown in Figure 13 (below, continues on page 18).

```
make report_summary
```

```
"Including generic.def file"
FAILED
Checked 97 runs
Reported errors(8):
  test_target_data_use_device_ptr.c on gcc 9.0.0 (compiler)
  test_target_enter_exit_data_classes.cpp on g++ 9.0.0 (compiler)
  test_target_map_classes_default.cpp on g++ 9.0.0 (runtime)
  test_target_teams_distribute_collapse.c on gcc 9.0.0 (runtime)
  test_target_teams_distribute_nowait.c on gcc 9.0.0 (runtime)
  test_target_teams_distribute_parallel_for_firstprivate.c on gcc 9.0.0 (runtime)
  test_target_teams_distribute_reduction_and.c on gcc 9.0.0 (compiler)
  test_target_teams_distribute_reduction_or.c on gcc 9.0.0 (compiler)
```

Figure 13: Example of a report summary (Source: https://crpl.cis.udel.edu/ompvvsollve)

For a more detailed and visually appealing report, the '**`make report_html`**' rule will generate a HTML webpage from the log files, using the template provided in the repository. This page will be stored in a new folder called **`results_report`**, and can be uploaded to a web server if desired. Figure 14 below shows an example screenshot of the generated web page. CSS and Javascript must be supported by your web browser in order to view the page.

#	Test name	Test system	Compiler name	Compiler result	Runtime result
1	gemv_target.cpp	cori	g++ 8.1.1	PASS	PASS
2	gemv_target_many_matrices.cpp	cori	g++ 8.1.1	PASS	PASS
3	gemv_target_reduction.cpp	cori	g++ 8.1.1	PASS	PASS
4	gemv_target_teams_dist_par_for.cpp	cori	g++ 8.1.1	PASS	PASS
5	linked_list.c	cori	gcc 8.1.1	PASS	PASS
6	mmm_target.c	cori	gcc 8.1.1	PASS	PASS
7	mmm_target_parallel_for_simd.c	cori	gcc 8.1.1	PASS	PASS

Figure 14: Screenshot of an example HTML results report, viewed in Google Chrome (Source: self-made)

Finally, to make it easier to share the results, the new '**`make report_online`**' feature has been recently added. This rule creates an HTML report and uploads it to a OMPVV project web server, printing to the console a link to the page. This allows for quick and easy sharing of the results, as this link is accessible from any web browser connected to the Internet. Your published results will be retained for a period of one month. Figure 15 below shows an example of using the online report feature. In addition to Python 3 or greater, this rule requires either the 'requests' package or Curl to be installed. 'requests' is preferred, as it has superior error-handling. It can be installed via pip using the command '**`pip install requests`**'.

```
> make report_online
"Including generic.def file"
Creating results.json file
Currently we only support run logs that contain compilation and run outputs. Use the
'make all' rule to obtain these
 === SUBMITTING ONLINE REPORT ===
We are using CURL because we could not find the `requests` package
Error handling is limited. Please consider installing `requests` through
    pip install requests
 Your report tag is 402796c6e. Do not lose this number
 Visit your report at:

https://crpl.cis.udel.edu/ompvvsollve/result_report/results.html?result_report=40279
6c6e
 This tool is for visualization purposes.
 Our data retention policy is 1 month.
 After this time, we do not guarantee this link will work anymore
 === SUBMISSION DONE ===
```

Figure 15: Example usage of the online report Makefile rule (Source: https://crpl.cis.udel.edu/ompvvsollve)

As shown in Figure 15 above, the output of the online report rule provides, in addition to the link to the webpage, a unique tag. This tag should be kept for future use, as it can be used to append results to the same report using the two special Makefile options provided for the online report feature. Figure 16 lists these two options and describes their use.

Option	Description
`REPORT_ONLINE_TAG=[tag]`	Specify an online report tag. If a report with this tag exists, the generating report will replace the existing report, unless the **`REPORT_ONLINE_APPEND`** option is used.
`REPORT_ONLINE_APPEND=1`	Enables appending of the generating report to an existing report. Requires the **`REPORT_ONLINE_TAG`** option to be specified.

Figure 16: OMPVV Makefile options for the online report rule (Source: self-made)

Now that the full process of using the test suite, from setup to reporting, has been covered, the remainder of this test suite will focus on procedures for creating new tests and contributing to the suite.

CREATING NEW TESTS

Project Workflow

New tests for the repository are categorized into two types, application kernels and construct tests. Application kernels are taken from real-world applications, while construct tests are created by the developers from a close reading of the relevant sections of the OpenMP specifications. Tests that are taken from the specification are divided into folders by which OpenMP construct or combined construct they apply to. Figure 16 shows the general procedure followed to ensure new tests comply with the specification before adding them to the suite.

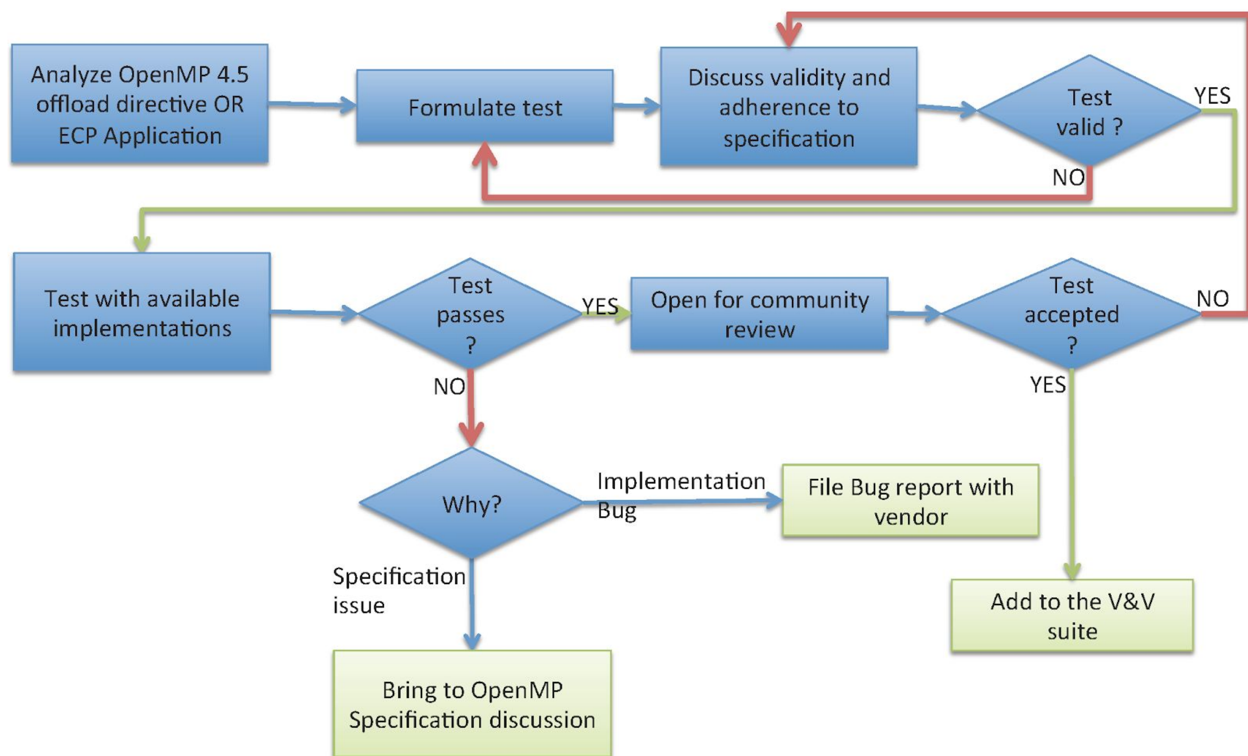

Figure 17: The OMPVV workflow for creating new tests (Source: https://crpl.cis.udel.edu/ompvvsollve)

As shown in Figure 17 above, tests start from either the specification or an application. Typically, a specification-based test is aimed at checking one construct or one clause on one construct, and ensures that the behavior required by the specification is upheld for that construct or clause.

After a test is first formulated, it is submitted for discussion, tested, and reviewed by the community. If it passes through each of these steps, it is added to the test suite.

A test, when compiled and run (i.e., when it is tested), will either pass or fail. When a test that is thought to be valid fails, it is due to one of three problems. First, the compiler implementation could be incorrect, in which case a bug report must be filed with that compiler's vendor. Second, there could be an ambiguity or contradiction in the specification, in which case the issue must be brought to the OpenMP community. Finally, the test itself could be invalid, in which case the developer of the test must revise it.

The OMPVV Header File

In order to ensure output for each test is standardized and formatted consistently, the test suite includes header files in C and Fortran that all tests must use for checking assertions and printing output. Figure 18 (below, continues on page 23) details the available macros. See Figure 19 (pages 23-24) for some usage examples in context.

Macro	Description
`OMPVV_INFOMSG(message, ...)`	If VERBOSE_MODE is defined, print an info message with the header [OMPVV_INFO filename.c:]
`OMPVV_INFOMSG_IF(condition, message, ...)`	If VERBOSE_MODE is defined and (condition == true), print an info message with the header [OMPVV_INFO filename.c:]
`OMPVV_WARNING(message, ...)`	If VERBOSE_MODE is defined, print a warning message with the header [OMPVV_WARNING filename.c:]
`OMPVV_WARNING_IF(condition, message, ...)`	If VERBOSE_MODE is defined and (condition == true), print a warning message with the header [OMPVV_WARNING filename.c:]
`OMPVV_ERROR(message, ...)`	If VERBOSE_MODE is define, print an error message in stderr with the header [OMPVV_ERROR filename.c:]
`OMPVV_ERROR_IF(condition, message, ...)`	If VERBOSE_MODE is defined and (condition == true), print an error message in stderr with the header [OMPVV_ERROR filename.c:]

OMPVV_TEST_OFFLOADING	check if offloading is enabled, fill this variable with such information. If VERBOSE_MODE is defined, print an info messages saying where the test is running (e.g. [OMPVV_INFO test.c:20] Test is running on device.).
OMPVV_TEST_AND_SET_OFFLOADING(var2set)	Same as OMPVV_TEST_OFFLOADING, but sets the variable var2set with the result to be used outside.
OMPVV_TEST_AND_SET(err, errorCondition)	If the errorCondition is true, error will be set to true.
OMPVV_TEST_AND_SET_VERBOSE(err, errorCondition)	If the errorCondition is true, error will be set to true. If VERBOSE_MODE, an error message will be generated when the errorCondition is true. The errorCondition is the condition that is required to generate an error
OMPVV_REPORT(err)	Based on the *err* variable, print if the test passes or fails (e.g. [OMPVV_RESULT] Test passed on the device).
OMPVV_RETURN(err)	Based on the *err* variable, return *EXIT_SUCCESS* or *EXIT_FAILURE*. *err* should be 0 if no error was encountered during the test
OMPVV_REPORT_AND_RETURN(err)	Same as calling OMPVV_REPORT and OMPVV_RETURN one after the other

Figure 18: Table of OMPVV header file macros (Source: https://crpl.cis.udel.edu/ompvvsollve)

Coding Standards

Similarly, the test suite has a number of coding standards that should be followed when creating new tests. Figure 19 (continues on page 24) shows an example of a test which follows the coding standards.

```
//===-- test_target_data_map_from.c ----------------------------------------===//
//
// OpenMP API Version 4.5 Nov 2015
//
```

```
// This file is a test for the target data construct when used with the map
// clause. This clause should create the mapping of variables into the device
// and do the data movement or allocation depending on the map type modifier
// from. This test uses arrays of size N which values are modified in the
// device and tested in the host.
//
//===----------------------------------------------------------------------===//

#include <omp.h>
#include <stdio.h>
#include “ompvv.h”

// Test for OpenMP 4.5 target data map(from: )
int main() {
  OMPVV_TEST_OFFLOADING;
  int sum = 0, sum2 = 0, errors = 0, isHost = 0;

  // host arrays: heap and stack
  int *h_array_h = (int *)malloc(N*sizeof(int));
  int h_array_s[N];

#pragma omp target data map(from: h_array_h[0:N])  \
    map(from: h_array_s[0:N])                      \
    map(from: isHost)
  {
#pragma omp target
    {
      isHost = omp_is_initial_device();
      for (int i = 0; i < N; ++i) {
        h_array_h[i] = 1;
        h_array_s[i] = 2;
      }
    } // end target
  } // end target data

  // checking results
  for (int i = 0; i < N; ++i) {
    sum += h_array_h[i];
    sum2 += h_array_s[i];
  }

  free(h_array_h);
  OMPVV_TEST_AND_SET_VERBOSE(errors, N != sum) || (2*N != sum2));
  OMPVV_INFOMSG_IF(errors, "Test failed on %s: sum=%d, sum2=%d, N=%d\n", (isHost ?
"host" : "device"), sum, sum2, N);

  OMPVV_REPORT_AND_RETURN(errors);
}
```

Figure 19: Example of a specification-based test from the OMPVV suite (Source: https://crpl.cis.udel.edu/ompvvsollve)

Figure 19 also exemplifies the use of header file macros, as documented in Figure 18 (pages 22-23). This example demonstrates the use of the following key rules from the coding standards:

1. All tests should start with a top-of-file header comment fully describing the test, and formatted as shown in Figure 19 (above on page 24)
2. Indentation should be performed with two spaces, and never with tabs.
3. Spaces should appear in the following situations:
    a. Before opening parenthesis after any control flow statement (e.g., if, for and while).
    b. After closing parenthesis before opening brace for a code block.
    c. Around arithmetic, logical, and assignment operators, except * and /.
    d. After commas in function parameter and argument lists, and in lists within OpenMP clauses
    e. After the colon in an OpenMP clause in which the colon precedes a list
4. No spaces should appear between function name and opening parenthesis in function calls, or between the increment/decrement operator and the variable name.
5. In OpenMP clauses, no spaces should appear around colons in array sections, or between the variable name and opening bracket for array sections
6. C-style comments using // are preferred
7. OpenMP pragmas should be unindented

CONTRIBUTING TO THE SUITE

Pull Requests

When working on a new test, commit your test and any changes to that test to a new git branch from master called **`new_test/<name_of_your_test>`**. Similarly, if working on a bug fix for a test, prefix the branch name with **`fix_test/`**, or if working on the infrastructure, prefix with **`infrastructure/`**. Before creating any pull request for your branch, make sure it is synchronized with master by running the git commands shown in Figure 20.

```
git fetch upstream
git checkout <your branch name here>
git merge upstream/master
```

Figure 20: Steps to ensure your branch is up-to-date with master (Source: https://crpl.cis.udel.edu/ompvvsollve)

Once a branch is ready to be merged, you can start the review process (see Figure 16) by creating a pull request (PR) on the project Github (https://github.com/SOLLVE/sollve_vv), to merge your branch with master. Include in the request a description of your changes and the current testing results for the changes on a particular system. Be sure also to request reviews from the developers, and add the appropriate tags for the pull request (for example, if your PR is a new test, add the "new_test" flag, and if your PR is a bug fix, use the "fix_bug" tag). If your PR addresses an issue, indicate the issue number.

After submitting a pull request, you will be notified of any comments or reviews on the request. It is your responsibility as the developer to respond to those comments. When a PR has received at least two approving reviews from the developers, it will be merged to the master branch.

Issues

If you encounter a problem with the test suite, such as a bug in the infrastructure or in a test, or you want to propose a new feature or idea for a test, create an issue on the project Github. There are template forms on the project Github for each of these three categories of issue (bug report, feature request, and test request), which will request the information shown in Figure 21 (below, continues on page 27) to help the developers tackle your issue.

Category	Requested Information
Bug Fix	● Clear and concise description of the bug

	• Test or a list of tests it applies to • Steps needed to reproduce the bug • Description of the expected behavior • Which compilers were used in finding the bugs • What accelerator hardware was used in finding the bugs
Feature Request	• Description of any problem with the suite related to the feature being requested (e.g., "I'm always frustrated when…") • Description of the requested solution to the problem • Description of any alternative solutions or features considered • Additional context for understanding the request
Test Request	• OpenMP directive or clause the test applies to • Details from the OpenMP specification that are being tested, especially those that are not obvious • Pseudocode for an ideal code structure

Figure 21: Information requested in each category of Github issue (Source: self-made)

After submitting an issue, you will receive notification of any comments on the issue. It is your responsibility to respond to comments to help the developers address your issue. Be sure also to add any additional tags needed to categorize your bug report, such as "5.0" if it applies to specification version 5.0, or "infrastructure" if it applies to testing infrastructure.

Vendor Bug Reports

In some cases, during the development of the suite it is necessary to file a bug report with a compiler vendor when one of the tests indicates there is a problem with their implementation of OpenMP. Before filing any bug with a vendor, be sure that the test had been determined to be valid by the OMPVV developers and that you fully understand the behavior of your code, as well as the expected behavior. A good bug report should also include a simple working example of a code that demonstrates the bug (usually this is just a stripped-down version of one of the OMPVV tests).

For some compilers, filing a bug report is simply a matter of emailing the appropriate contact, and email addresses of those contacts can be found on the Github wiki for the project. However, for GCC and Clang, you must file the report through the Bugzilla system. GCC has a form for creating Bugzilla accounts (https://gcc.gnu.org/bugzilla), while for Clang you must email bugs-admin@lists.llvm.org in order to request an account. Follow the particular instructions on the Bugzilla to complete your bug report.

APPENDIX A: Interview

This handbook was prepared with the assistance of an interview with Jose Monsalve Diaz, a PhD student at UD and the primary developer of the OMPVV test suite. During our interview, I asked Jose what parts of the documentation needed the most attention, and where I could expand into new subjects not already covered by the documentation. Unfortunately, while his suggestions were excellent, I could not include many of them as they would extend the page count too far for this assignment. However, one suggestion in particular I did spend some time working on: the documentation for creating a new system.def file and editing the make.def file. At Jose's suggestion I spent additional time documenting the procedure for editing these files and the meaning of the variables in them. These steps are crucial to getting a new system set up with the test suite, but they are not very intuitive, so I believe it will be of great benefit to the OMPVV test suite that this handbook details those procedures.

Some of Jose's other suggestions included:

- Instructions on how to extend the infrastructure (add new macros for the header file, adjust the log file scripts)
- Steps to build the GCC and Clang compilers with offload support
- Improving the Makefile examples to be more comprehensive
- Adding a table of compiler flags for offloading

I also interviewed four AMD employees who regularly use the test suite to get their general feedback on the documentation and other aspects of the test suite. Their names are Damon McDougall, Greg Rodgers, Ron Lieberman, and Ethan Stewart. Our discussion was wide-ranging, and included speculation and feedback about some topics outside the scope of this report, but here are some relevant points they raised:

- Adding instructions on how to build offloading compilers is likely not worth my time, since it is a very involved process and the compiler vendors will already have their own documentation for those tasks
- On the other hand, it would be useful to have a table of flags needed for each compiler to use offloading (see Jose's suggestion
- In the future, a convenient "dashboard" of results on the web page would be useful\
- The report_online feature can't be used by some vendors because results cannot be uploaded to unsecure web servers

APPENDIX B: Changes from previous versions

Changes from version 1.0 to version 1.1

- Corrected Figure 7 descriptions for the run, compile, all, clean, and tidy rules.

WORKS CITED

"-Qoffload IBM Knowledge Center.", https://www.ibm.com/support/knowledgecenter/SSXVZZ_16.1.0/com.ibm.xlcpp161.lelinux.doc/compiler_ref/opt_offload.html.

"Cray Documentation Portal.", https://pubs.cray.com/content/S-5212/9.1/cray-compiling-environment-cce-release-overview/cce-91-software-enhancements.

"Getting Started: SOLLVE OMPVV.", https://crpl.cis.udel.edu/ompvvsollve/project/gettingstarted/.

"Intel® C++ Compiler 19.0 for Linux* Release Notes for Intel® Parallel Studio XE 2019.", https://software.intel.com/content/www/us/en/develop/articles/intel-c-compiler-190-for-linux-release-notes-for-intel-parallel-studio-xe-2019.html.

"OpenMP 5.0 TARGET with Intel Compilers .", https://software.intel.com/content/www/us/en/develop/articles/openmp-50-target-with-intel-compilers.html.

"OpenMP Support - Clang 11 Documentation.", https://clang.llvm.org/docs/OpenMPSupport.html#basic-support-for-cuda-devices.

"Releases - ROCm Developer Tools.", https://github.com/ROCm-Developer-Tools/aomp/releases.