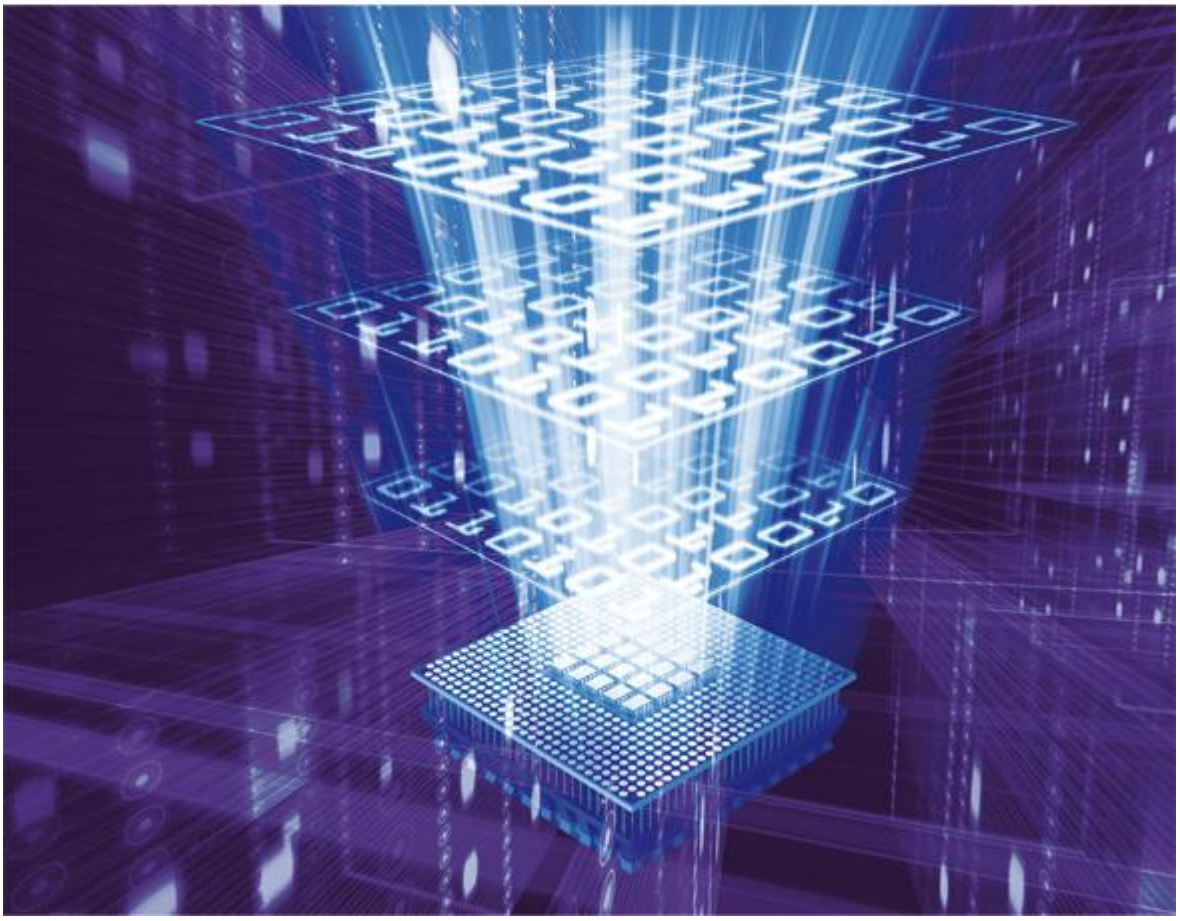
CAPS

hmpp

Hybrid Compiler for Manycore Applications



HMPP Wizard & Performance Analyzer User Manual

Version 2.0

November 2011

The information contained in this document is subject to change without notice.

REVISION HISTORY

Rev.	Date	Author	Chapters and pages modified	Changes
V1.0.0	15/12/2010	CAPS entreprise	All	Creation
V2.0.0	22/11/2011	CAPS enterprise	All	GUI Update

CONTENTS

1 Introduction

Warning:

In this document we assume that the reader knows HMPP concepts introduced in the HMPP Workbench reference manual ([R1]).

The Hybrid Multicore Parallel Programming workbench (HMPP) provides developers with a set of tools dedicated to build parallel hybrid applications running on many-core systems. These architectures combine general-purpose cores with hardware accelerators (HWAs) such as GPUs or SIMD computing units. HMPP allows the programmer to write hardware independent applications where hardware specific codes are dissociated from the legacy code as additional software plug-ins (refer to [R1] for more details concerning HMPP).

Ten years of experience in code porting have enabled CAPS to define a proven GPU focused methodology based on HMPP programming model and completed by a set of tools. HMPP Wizard & Performance Analyzer is one of these tools that help developers to incrementally migrate their code onto many-core architectures (cf. CAPS Methodology Code Porting [R5]). HMPP Wizard & Performance Analyzer answers main GPU kernel performance issues. It provides developers code transformation advice to make kernels GPU friendly. These modifications are either direct code transformation or through HMPP code generation directives (see [R2] for further details).

The objective of the HMPP Wizard & Performance Analyzer is to propose multiple performance issues and advice inside a single report. Performance factors are analyzed during both the compilation and the execution of the application and are then associated to the user code. The static and dynamic analyses of the application exposes different aspects of the application performance the user can investigate. With this approach, HMPP Wizard & Performance Analyzer improves the productivity of the programmer during all the steps of the many-core porting methodology [R5].

1.1 Approach

1.1.1 Integration in the Development Process

HMPP Wizard & Performance Analyzer works in two steps: information gathering and performance analysis.

The *information gathering* step is performed in two manners:

- Statically, on the application source code;
- Dynamically, using profiling information.

HMPP Wizard & Performance Analyzer are invoked as a compiler wrapper like HMPP or a profiler like Valgrind and can be seamless integrated in a Makefile. HMPP wizard gathers all the information and builds a final HTML report. The HTML report contains different views of the application performance with for instance one dedicated to performance issues and another one dedicated to the execution profile.

HMPP Wizard & Performance Analyzer proposes different analyses or modules controlled by the command line. The current version comes in three modules:

- The CPU profiler;
- The advice generator,
- The GPU profiler.

1.1.2 Focus on Hotspots: GNU Profiling

The CPU profiler module of HMPP Wizard integrates the use of the GNU profiler in the compilation process and analyzes the profiling information generated during the execution. The execution profile result is available under a specific view focusing on most important functions.

1.1.3 Highlighting Manycore Porting Issues: Advice Generator

The advice generator of HMPP Wizard analyzes the application source code to detect potential porting or performance issues and to advise the appropriate correction to the user. The suggestion focuses on properties of the code that are essential to get performances when porting on GPU-based systems.

HMPP Wizard proceeds as follows:

- A function is checked as a potential HMPP codelet.
- All loop nests of the function are analyzed as potential many-core kernels. A grid of threads (called kernel in the following) is computed and will be the basis of the HMPP Wizard analyses.
- A set of analyses is applied to each kernel and whenever possible, performance improvement suggestions are emitted along with statistics.
- If a computation matches a pattern supported by HMPP Wizard, specific recommendations are suggested.

HMPP Wizard can offer generic many-core programming advice or specific NVIDIA architecture tuned suggestions for the CUDA. The pattern matching technology of HMPP Wizard is based on the memory access patterns of kernels. Related suggestions mainly deal with CUDA memory coalescing improvements.

1.1.4 Analyze the GPU performance

This last module, called Performance Analyzer, provides information on the GPU behavior of the CUDA kernel generated by HMPP. Based on the use of the CUDA profiler, the collected information allows to detect for each codelets the most consuming time kernel. As for the CPU hotspot detection, this view aims to guide users through the incremental optimization phases.

1.2 Basic concept

We remind in this chapter some concepts handled with HMPP and useful in the context of HMPP Wizard. Please, refer to [R1] for further details.

1.2.1 Codelets

A codelet ([R1]) is a pure function that can be remotely executed on a massively parallel accelerator. It defines the granularity of the computation sections and contains one or more kernels. A codelet is supposed to be a heavy computational function representing a significant execution time of the application (also called "`hotspot`").

Due to the remote execution of codelets, data are uploaded on the accelerator. So, in practice, the notion of codelets should be extended to other functions performing some computations with the same remote data.

1.2.2 Kernels

A kernel represents a loop or a group of nested loops defining both an iteration space and a grid of GPU threads. Kernels extracted from the codelet are the basis of how a computation is distributed on massively parallel architectures.

The shape of the kernel defines an iteration space as follows:

- The dimension of the kernel iteration space is the number of nested loops in one loop, including that outer loop. Two nested loops define a 2D kernel.
- Each loop defines one dimension of the iteration space as given by the loop induction variable. The induction variable range is computed from the loop construction.
- Each dimension of the iteration space must be static and independent from the other ones. The dimension range has to be calculable and invariant.
- A variable defining a linear relation with an induction variable is an indirect induction variable and can be considered as one.

1.2.3 Grid of Threads

A kernel has to define a grid of threads that is a subset of the iteration space of the kernel or, in other words, one or more perfectly nested and parallel loops. These parallel loops define the distribution of the computation on the many-core accelerator.

This process, called *gridification*, is the most important parameter in the performance analysis because it has a direct impact on how memory is accessed. From this gridification, the kernel memory access footprint is computed. The sequence of memory reads and writes done in parallel by the computation is called the memory access pattern.

2 Supported Platforms and Compilers

2.1 Input Supported Languages

HMPP Wizard is valid for C and FORTRAN languages:

- Usual C89 language constructions with main gcc extensions
- Usual Fortran 90 constructions
- Input files with the .c, .f, .f90, .F90 extensions.

2.2 Supported Compilers

HMPP Wizard can be used with:

- GNU gcc 4.1 and above
- Intel icc version 11.x
- GNU gfortran 4.3 and above
- HMPP 2.5.x
- NVIDIA SDK 3.2 and 4.0

2.3 Supported Platforms

HMPP Wizard works with any kernel 2.6 Linux distribution containing the libc library that comes with g++ 4.x and above,

Below is the list of operating systems that have been validated with HMPP Wizard.

- Debian 5.0 and above;
- RedHat Entreprise Linux 5.5 and above;
- OpenSuse 11.2;
- Suse Linux Entreprise Server 11.0;
- Fedora 13

HMPP Wizard is run with the `hmppReport` script that uses standard unix commands:

- Basic file management: `mkdir, mv, touch, basename, dirname, cat, test, wc, tar, find`
- String manipulation: `cut, sed, sort, grep`
- Extra: `date, uudecode, gprof`

The `uudecode` Unix command is often not installed by default but is widely available with the "`sharutils`" package.

2.4 Supported Web Browser

HTML reports generated by HMPP Wizard & Performance Analyzer are officially supported with Firefox, version 8.0.

3 HMPP Wizard installation

3.1 Installation

To install the HMPP Wizard package, launch the following command and follow the instructions:

```
$ HMPPWizard-2.0.x_linux64.bin
```

Figure 1 - HMPP Wizard installation popup windows

A command line mode is also available:

```
$ ./HMPPWizard-2.0.x_linux64.bin --mode text
```

3.2 HMPP Wizard License

A license is required to be able to use the HMPP Wizard tool. This license takes the form of a "`xxxxx.lic`" file provided on demand by CAPS entreprise.

A single license server can be used for HMPP Workbench and HMPP Wizard. Please refer to the License Installation Guide ([R6]) for further details concerning the management of the license and the installation of the HMPP Workbench license server.

3.3 Initialization of the environment

The HMPP Wizard environment needs to be setup before launching it. This is done by sourcing the provided script located in the installation's `bin/` directory:

```
$ source "HMPP_WIZARD_INSTALLATION_PATH"/bin/hmppwizard-env.sh
HMPP Wizard environment set
```

4 Running HMPP Wizard & Performance Analyzer

The use of the HMPP Wizard & Performance Analyzer comprises three stages:

- First stage: analysis of the application source code and generation of data
- Second stage: generation of an HMTL report
- Third stage: launch of the HTML browser that displays the results.

Depending on the demanded analyses, the generation of the report can be done at different levels: either by a static analysis of application source files or by retrieving information when running the application (profiling information for example).

Available commands and options are described below.

Remark:

Note that the current implementation does not support a concurrent execution of the analysis stage for the same output report. For example, the command "make – j" is not supported.

4.1 HMPP Wizard & Performance Analyzer Commands

4.1.1 Generic Command Format

The usual form of the HMPP Wizard commands relies on a typical compilation command line:

```
$ hmppReport [--help] [-h]
             [--help-options] [-H]
             [--version] [-v]
             [--build] [-b]
             [--force] [-f]
             [--verbosity]  [-d]
             [--licenses]
             -tools=[List of supported tools]
             -o [OutputDirectory]
             [--] <Usual command line>[1]
```

With:

- `--help`: display the list of options and quit
- `--help-options`: display more advanced options
- `--version`: display the version number.
- `--build`: end the analysis process, produce the synthesis of the analysis and generate the HTML report (see chapter 4.1.4). This command is mandatory to produce the HTML report.
- `--force`: overwrite existing project and files if any
- `--verbosity`: increase the amount of information displayed during the execution of internal operations
- `--licenses`: get information regarding the license management and display required information for a license request
- `--tools=[List of supported tools]`: specify the tools activated to analyze the application. Several tools can be indicated in the same command separated by a comma (","). Currently HMPP Wizard supports the following tools:

[1] To improve the readability, we present here the "`Usual command line`" on a separate line in the command format. However, be aware that the command line must follow the `hmppReport` command.

- `wizard`: apply source code analysis and give advice (see chapter 4.1.2);
- `gprof`: launch a profiling analysis based on the GNU profiler (see chapter 4.1.3);
- `perfanalyzer`: launch a profiling analysis based on the CUDA profiler (see chapter XXX)

- `-o [Output directory]`: indicate the output directory of the HTML report and of all intermediate files generated during the analysis. The default directory is "`__hmpp_report`". The report is composed of a file named "`index.html`", main entry point, and of two directories: "`data/`" containing the information computed during the analysis, and "`assets/`" containing the read-only and static part of the report.
- `<Usual command line>` refers to the command line used to compile or run the application. When the command line is not a compilation command (and prefixed by supported compilers), the separator "`--`" must be inserted before the command.

When using the static analysis, a simplified compilation line is possible. In that case, the following compiler options might also be needed to correctly preprocess the source files:

- `-D<macro>[=value]`: defines a macro
- `-U<macro>[=value]`: un-defines a macro
- `-I<path>`: add a search path for the include files

and other specific options such as `-include, -isystem, -macro`, etc.

For the internal static analysis, HMPP Wizard does not recognize options that are not related to the preprocessing stage. They are silently ignored and are not passed to the preprocessor.

4.1.2 Wizard options

4.1.2.1 Command line

The "`wizard`" tool provides some analysis applied on the application source files. It is activated by default. The syntax of the command is[2]:

```
$ hmppReport [--help]
             [--v] [--V]
             -tools=wizard
             -o [OutputDirectory]
             <Usual compilation line>
```

The explicit activation is:

- `-tools=wizard`

This command must be followed by a "`--build`" command to generate the HTML report.

In this context, an example of HMPP Wizard command line could be:

[2] In bold, mandatory options.

```
## Launch of the static analysis
hmppReport -o ./report --tools=wizard gcc -O3 -lm -std=c99 Sourcecode_File_x.c

# Implicit call of the static analysis
hmppReport -o ./report gcc -O3 -lm -std=c99 Sourcecode_File_y.c

…


## Generate the HTML the report
$ hmppReport --build -o ./report

## Visualization of the results
$ firefox report/index.html
```

The "`build`" command is called at the end of the compilation process. When the application is made of a single file, the two commands can be combined into one: "`--build --tools=wizard`".

To force the generation of the report at all stages of the compilation, both the options "`--build`" and "`--force`" must be used at the same time. This method will overwrite previous report generations. In that case, the resulting report will be a combination of all analysis performed using the output directory.

4.1.2.2 Diagnoses and Advice

This section describes diagnoses and advice provided by the HMPP Wizard. They are provided in two separate sections:

- One dedicated to the diagnoses preventing a generation of efficient GPU code by HMPP.
- One dedicated to the performance improvement of HMPP generated code

For further details on these diagnoses please refer to Appendix 6.1.

4.1.2.2.1 Programming Diagnoses and Advice

Undefined Control Structure (switch case, goto/labels)

Diagnosis	Unstructured flow operations disrupt the compiler analysis and make the code generation difficult or impossible.
Advice	Use structured flow operations ('if' statements, ...) instead.

Large memory access stride

Diagnosis	The computation of a value uses large array access strides from the innermost loop induction variable. Large access strides may degrade the performance
Advice	Try to reduce the stride of memory accesses.

Induction Variable not found

Diagnosis	Induction variable not found

Advice	Re-write the loop: the current form hides the induction variable
Advice	Replace the loop operation by a proper 'for' loop

Loop may not be parallel

Diagnosis	The loop is not detected nor specified parallel
Advice	Check the parallelism of the loop

Conditional Operation in a loop

Diagnosis	Multiple `if` conditional found. Complex conditional expressions may impact the performance
Advice	Use masks instead of guards in computations
Advice	Split the kernel in several loop nests with the same execution flow (taking the same path)

2D Gridification not performed due to non-nested loop

Diagnosis	Computations between loops prevent the gridification
Advice	Move the computation inside the inner loop with a conditional
Advice	Move the computation before or after the loop nest in a new loop nest

4.1.2.2.2 Optimization Diagnoses and Advice

Reduction Inside a Kernel

Diagnosis	The symbol <symbol name> should be reduced inside the loop nest
Advice	Use the `hmppcg parallel reduce` directive

Bad Coalescing

This advice relies on the gridification and on the CUDA memory coalescing access patterns.

Diagnosis	The computation using <symbol name> is globally not well coalesced
Advice	Insert a loop interchange HMPP directive in the loop nest
Statistic	Number of subscript accesses over X

Statistic	Number of subscript accesses over Y
Statistic	Number of constant subscript accesses over the grid of threads
Transformation	<Code output with the appropriate hmppcg permute directive>

Inefficient Memory Coalescing

Diagnosis	A stride greater than one in the X dimension of the gridification prevents a good memory coalescing
Advice	Reduce the stride of the loop over the induction variable

Potential parallel kernel with a grid

Diagnosis	A kernel has been detected parallel and a 2D grid can be generated
Advice	Consider porting the execution of the function on a many-core architecture.

Call to a standard library function

Diagnosis	A function call belonging to a standard library has been detected and may require a specific action. Two types of library function are detected: • Standard functions of the language (libc); • Call to a GPU-accelerated library function
Advice	According to the type of the function detected the advice recommend to use GPU-accelerated library function

IO function call(s) from <stdio.h>

Diagnosis	Detection of functions requiring the Operating System into the source code. These functions cannot be used into an offloaded GPU code.
Advice	This part of the code must be removed from the function.

Parallel 2D/3D kernel unroll & jam Optimization

Diagnosis	The nest is made of more than 2 loops and will generate a parallel 2D grid. In this grid, there is no array access having more than 2 dimension(s). The computation may reuse data between iterations in the innermost loop.

Advice	You should consider an "`unroll and jam`" transformation that can increase the memory locality of the computation inside the body.

Too Low Compute Density

This advice is emitted when the computation density is too low compared to the memory accesses.

Diagnosis	The computation density is low
Advice	Input data should already be on the accelerator to get performance
Statistic	Number of array accesses
Statistic	Number of operations
Statistic	Number of intrinsic operations

The advice given in this context indicates that if the data are not already available on the HWA (means that they will need to be transferred), the cost of their transfer will likely cost more than the gain in performance.

2D Convolution Patterns

Diagnosis	A 2D convolution pattern was found writing <symbol name> over <induction variable Y> and <induction variable X>
Advice	You should consider using an optimized kernel for this access, more information in the HMPP Cookbook
Statistic	Minimum stencil offset over each induction variable
Statistic	Maximum stencil offset over each induction variable

4.1.3 Gprof for Profiling Information

The CPU profiling information provided by HMPP Wizard is based on the use of the GNU profiler. We assume in the following that readers are already familiar with gprof. For further information please refer to [R8]..

Profiling gives application execution time per function and the functions call graph. This information enable to identify where most of the execution time is spent if some functions are slower than expected, they might need to be rewritten.

When using `gprof`, we distinguish three stages:

- Compilation and link of the application with profiling enabled.
- Execution of the application that generates a profile data file.
- Running `gprof` to analyze the profile data.

These stages are still valid in the context of the HMPP Wizard except that:

- Compilation as well as the execution are directly done by the HMPP Wizard.
- The result information is synthesized in the generated HTML report.

4.1.3.1 Command line

Syntax of the command is[3]:

```
### Instrumentation of the application
$ hmppReport [--help]
             [--v] [--V]
             --tools=gprof
             -o [OutputDirectory]
             <Usual compilation line>

### Generation of the profiling data
$ hmppReport –tools=gprof -- <execution command>
```

Mandatory parameters are:

- `-tools=gprof`

This command must be followed by a "`--build`" command.

In this context, an example of HMPP Wizard command line could be:

```
## Compilation stage
hmppReport --tools=gprof -o ./report gcc -O3 -lm -std=c99 Sourcecode_File_x.c
hmppReport --tools=gprof -o ./report gcc -O3 -lm -std=c99 Sourcecode_File_y.c
…
hmppReport --tools=gprof -o ./report gcc -O3 -lm -std=c99 -o myBinaryFile

## Generation of the profiling data
hmppReport --tools=gprof -o ./report -- myBinaryFile arguments

## Build the report
$ hmppReport --build -o ./report
```

The "`--build`" command must be called only once at the end of the execution process.

This tool can be combined with any other tool.

4.1.4 Performance Analyzer of the GPU code execution

The Performance Analyzer module is provided to let end-user an easily way to analyze the behavior of CUDA kernel on a GPU. This tool is based on the CUDA profiling system. Due to hardware limitations, several execution of the application can be required to get all the necessary profiling information. Depending on the amount of details requested on the hardware type, from one to seven executions are possible.

[3] In bold, mandatory options.

4.1.4.1 Command line

Syntax of the command is[4]:

```
### Instrumentation of the application
$ hmppReport [--help]
             [--v] [--V]
             --tools=perfanalyser
             [--perfanalyzer_runs=<integer>]
             [--perfanalyzer_cuda_arch=<integer>]
             -o [OutputDirectory]
             <Usual compilation line>

### Generation of the profiling data
$ hmppReport -tools=perfanalyser -- <execution command>
```

Where:

- `--perfanalyzer_runs:` specify the number of profiling run. Each run activates different counters. Default value is 3. Each run required by the analysis is automatically launched by the Performance Analyzer Module.
- `--perfanalyzer_cuda_arch:` specify the CUDA architecture. Value can be 13 or 20 regarding the type of GPU used. Default value is 20.

Appendix 6.3 details the different counters validated for each run.

Mandatory parameters are:

- `-tools=perfanalyser`

This command must be followed by a "`--build`" command.

[4] In bold, mandatory options.

```
##Compilation of the application
hmppReport --tools=perfanalyzer -o reportMatrixFilter hmpp -p  gfortran  -O3  -g
filter_matrix-3.o time.o -o filter_matrix-3.exe
…
## Execution of the application managed by the Performance Analyzer
hmppReport --tools=perfanalyzer -o reportMatrixFilter -- ./filter_matrix-3.exe

 Matrix sizes =          203

 GPU Performance =    3.5286738094233772      GFlop/s -   23.707000000000001      ms
 Matrix sizes =          203

 GPU Performance =    3.7762050286642888      GFlop/s -   22.152999999999999      ms
 Matrix sizes =          203

 GPU Performance =    6.8715516674880899      GFlop/s -   12.173999999999999      ms
 Matrix sizes =          203

 GPU Performance =    7.2971275296580593      GFlop/s -   11.464000000000000      ms
 Matrix sizes =          203

 GPU Performance =    6.6355413659078302      GFlop/s -   12.606999999999999      ms
 Matrix sizes =          203

 GPU Performance =    6.7392467574317241      GFlop/s -   12.413000000000000      ms
 Matrix sizes =          203

 GPU Performance =    6.9538046550290931      GFlop/s -   12.029999999999999      ms

## Launch of the report generation
hmppReport --tools=perfanalyzer  -o reportMatrixFilter --build
```

4.1.5 HMPP Wizard Command, Build Option

The last "`--build`" command allows to:

- Aggregate in a single interface all the analysis performed (advice, profiling);
- Build the HTML report.

Syntax of the command is:

```
$ hmppReport --build -o [OutputDirectory]
```

Then to visualize the report enter the command below:

```
$ firefox  [OutputDirectory]/index.html
```

4.2 HMPP Wizard: Visualize the Results

All the work performed by HMPP Wizard is gathered in a single HTML report. The interface is organized in the form of plugins dedicated to some themes. A system of filters allows the user to select relevant information.

4.2.1 Home page

The `Home` page summarizes the number of files, functions, etc. that have been analyzed.

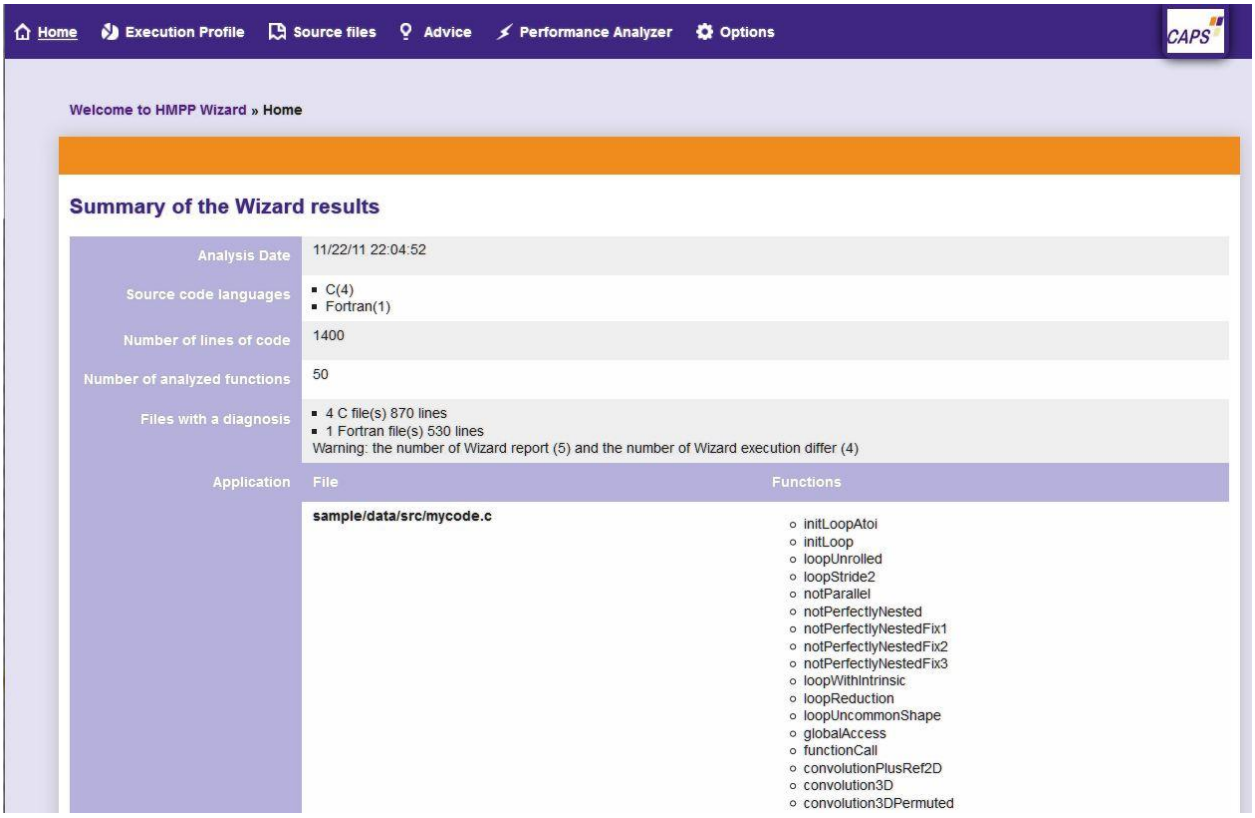


Figure 2- HMPP Wizard & Performance Analyzer - Welcome page

4.2.2 General approach

The HMPP Wizard HTML interface objective is to provide, from different point of views, a feedback of the application performance on a many-core accelerator. Each point of view is called a "`view`" and can be accessed directly from the list at the top of the web page. They all work on the same principles.

To operate, the view works on one particular type of element in the application: can be a file, a function, or a profiling run. Focused on that element, the view opens a page with at least two tabs: the filter tab and the result tab. The filter tab is used to select the subset of element in the application displayed in the result tab. Further tabs can be opened depending on the view.

The current version supports the following views:

- The execution profile: filter on CPU profiling runs; displays hotspots of the application.
- The source view: filter on directories; provides a global overview of the application.

- The advice view: filter on advice categories; provides an overview of performance issues.
- The performance analyzer view: filter on directories; provides details on the performance achieved on the accelerator.

4.2.3 Source View

The source view provides an overview of all files analyzed in the application.

In the Filter Tab of the Source Files menu, select the files for which you want to see the possible advices.

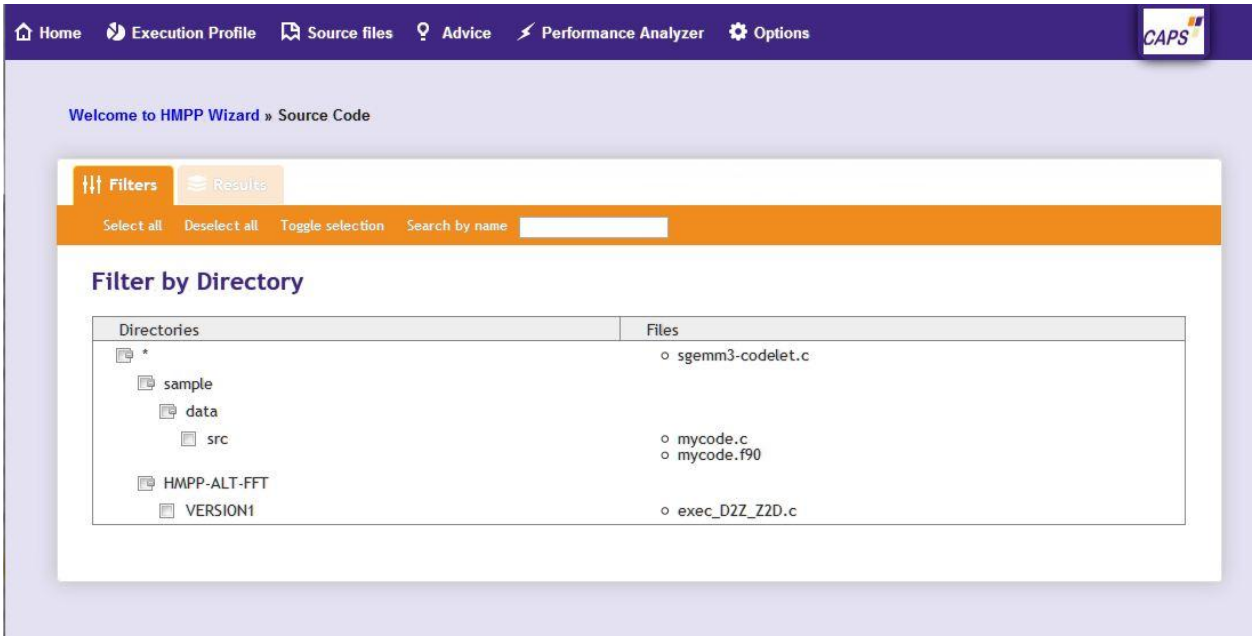


Figure 3 - Source Files tab

Then for each file, you get:

- The list of functions defined in the file
- For each function the number of loop nests present
- For each function the number of advices proposed.

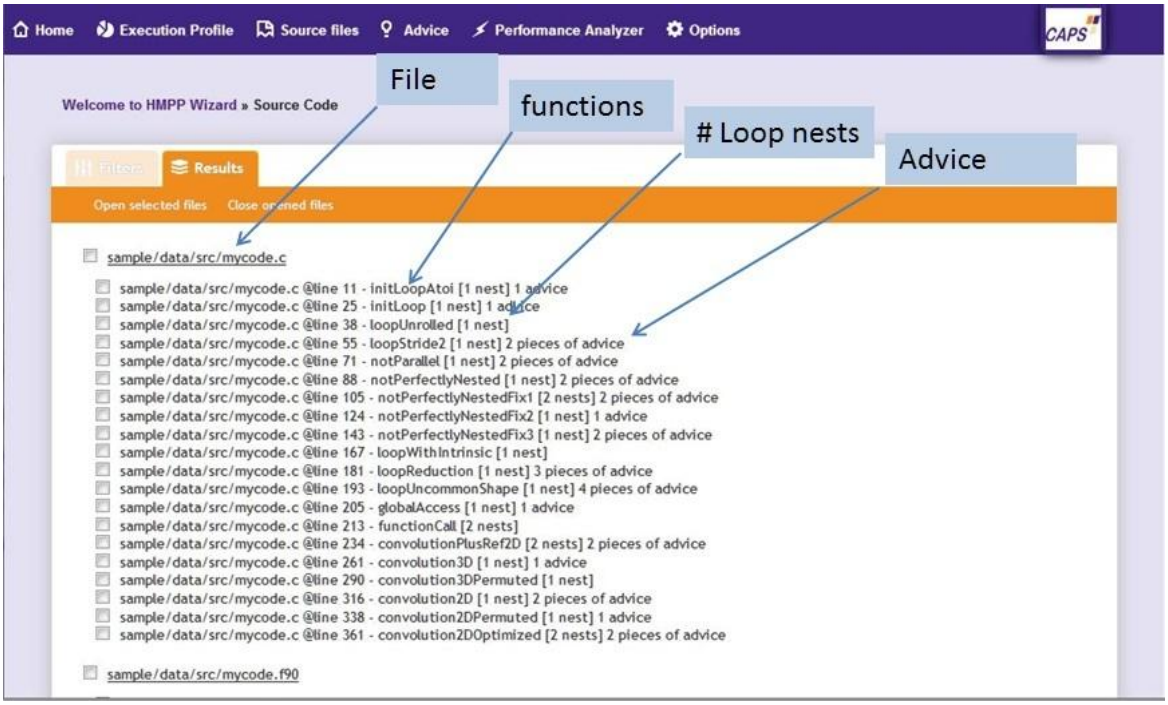


Figure 4 - File-by-File advice view

4.2.3.1 Filter Advice by Type of Diagnosis

In the Filter tab, select all types of diagnoses you want to display the results.

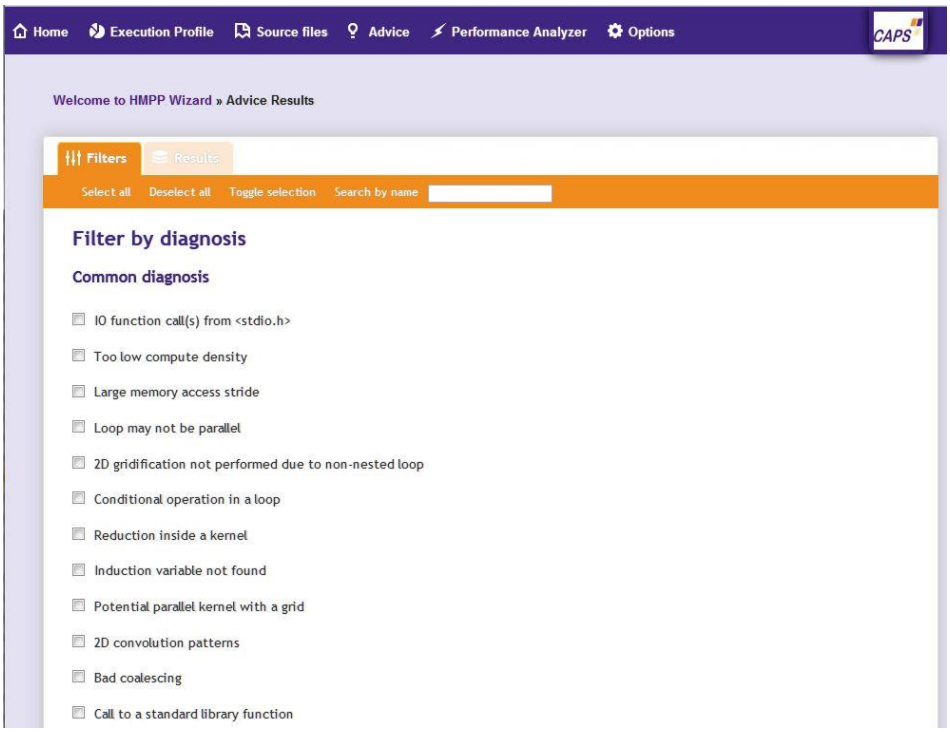


Figure 5 - Advice filter view

4.2.3.2 Overview of Performance Issues per Type of Diagnosis

Clicking in the Result tab displays all file locations where the selected analyses detect potential issues. A click on an advice opens a new tab with a detail description of the problem.

A color legend is used to present the results:

- In red, advices emitted when the code can not be runned on GPU;
- In orange, advices dedicated to potential GPU execution issue;
- In green, advices dedicated to optimization improvement.

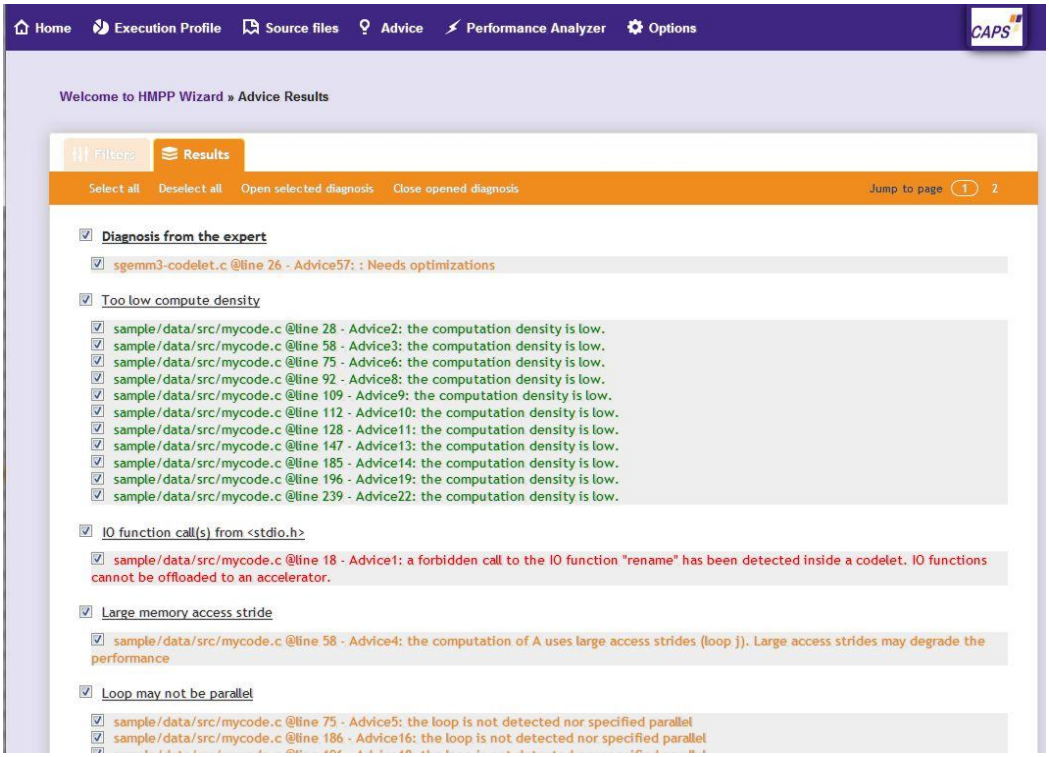


Figure 6 - Advice view

4.2.3.3 Detailed View of an Advice

As shown in the snapshot below (Figure 7), the result of the analysis displays the source code and details about the potential issue..

Furthermore, in some situations, additional information can be provided by clicking on the link "`Connect to MyDevDeck`". MyDevDeck is a website maintained by CAPS entreprise's team whose the purpose is to provide additional information about a situation or a concept (see Figure 8).

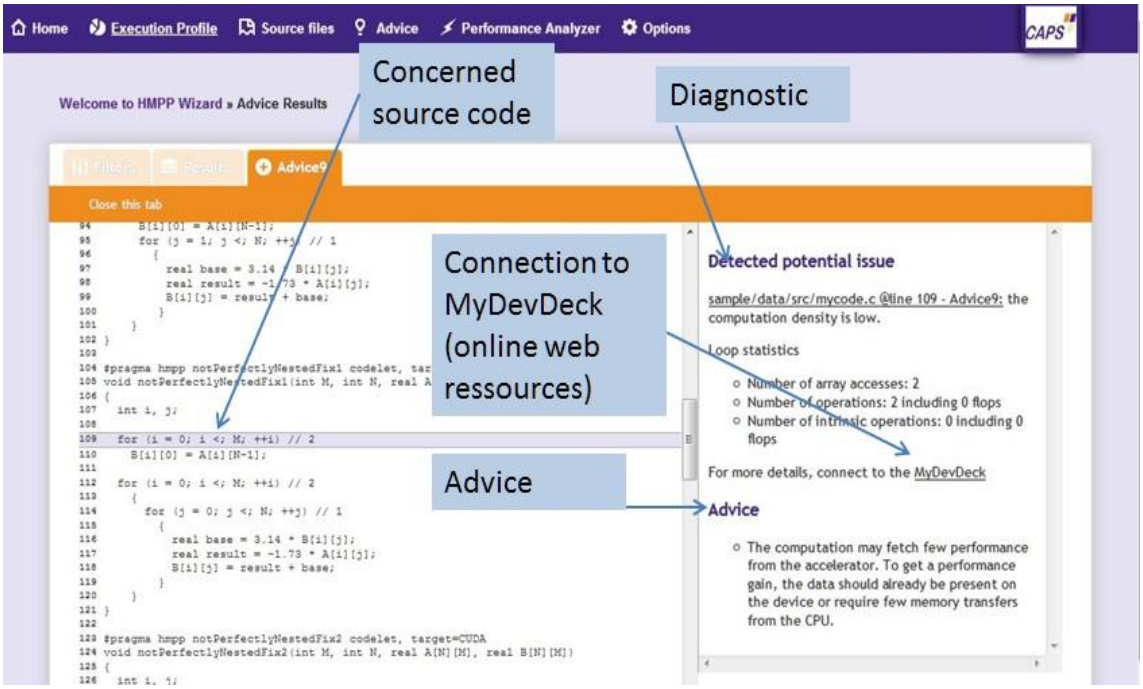


Figure 7 - Advice detail

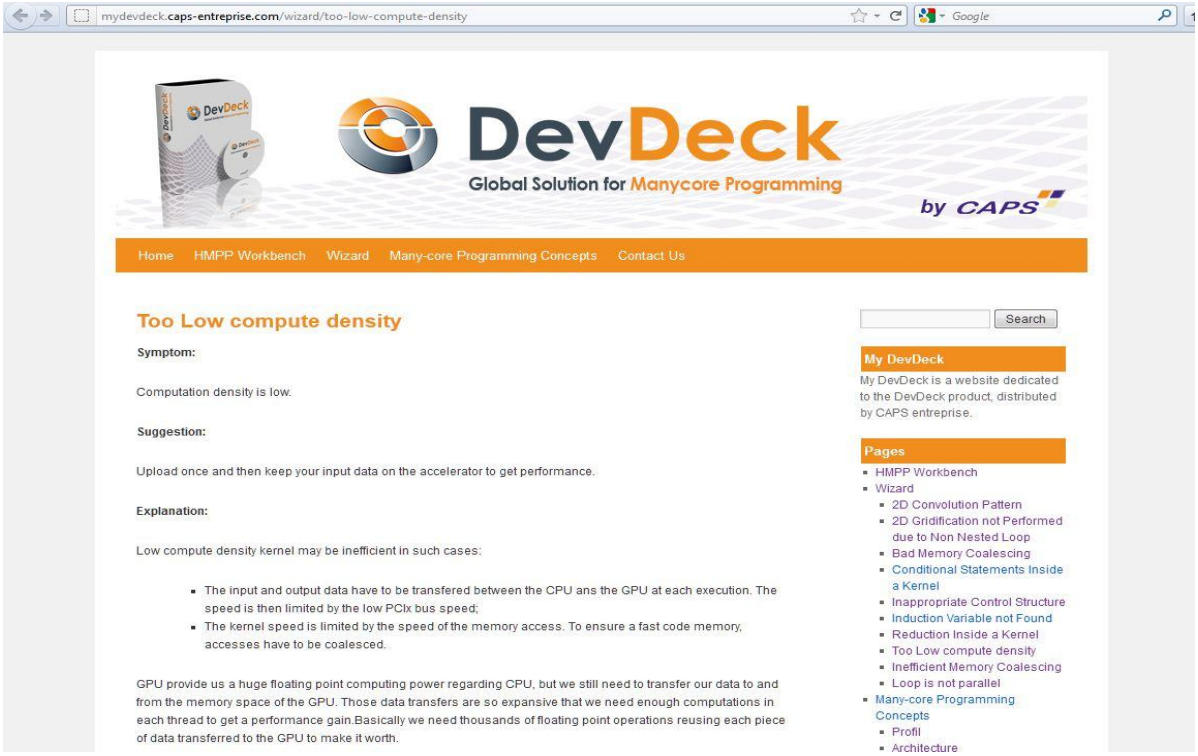


Figure 8 - Online Ressources - MyDevDeck website

4.2.4 Execution profile View

4.2.4.1 Profiling Selection

The `Execution Profile` main tab allows you to select the execution time of the 5 hotspots, the 5 other ones or all functions. In the following display, different runs have been executed ("`run1`, `run2`") and gathered in a same report.

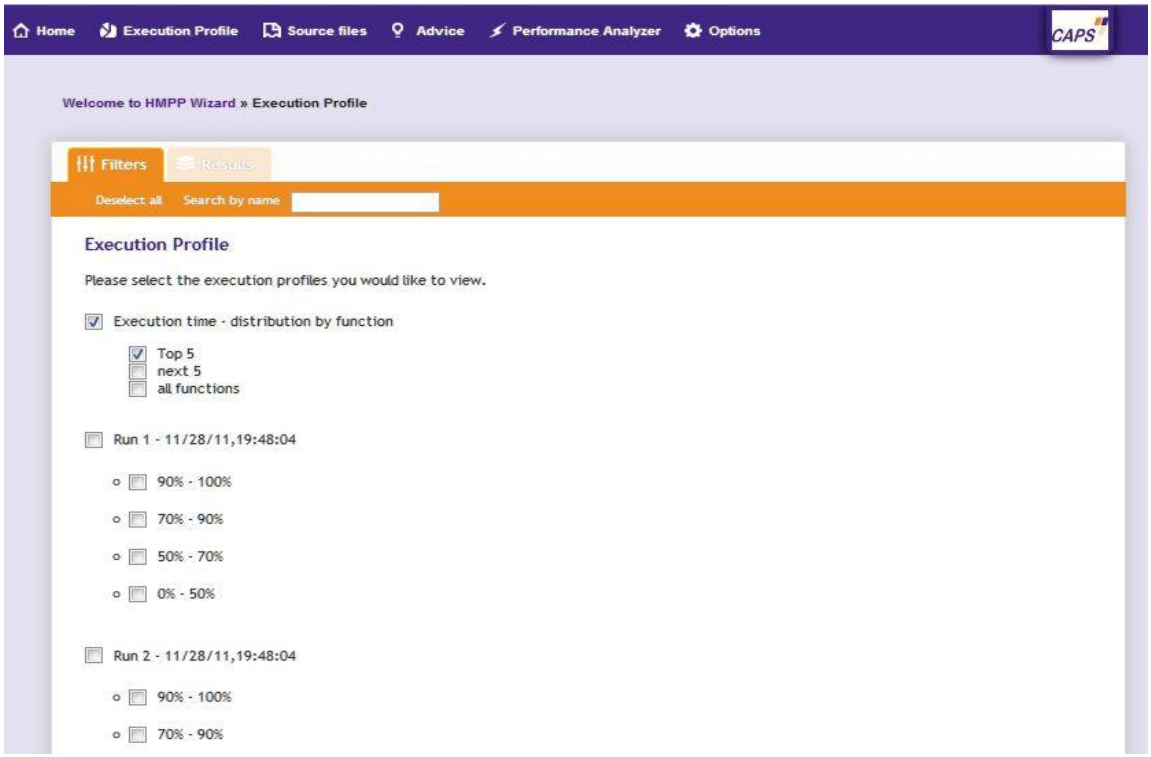


Figure 9 - Execution Profile - Filter view

4.2.4.2 Execution Profile of the Selection

The Result tab of the Execution Profile contains a graph that displays the accumulated execution time of functions selected in the left part of the tab.

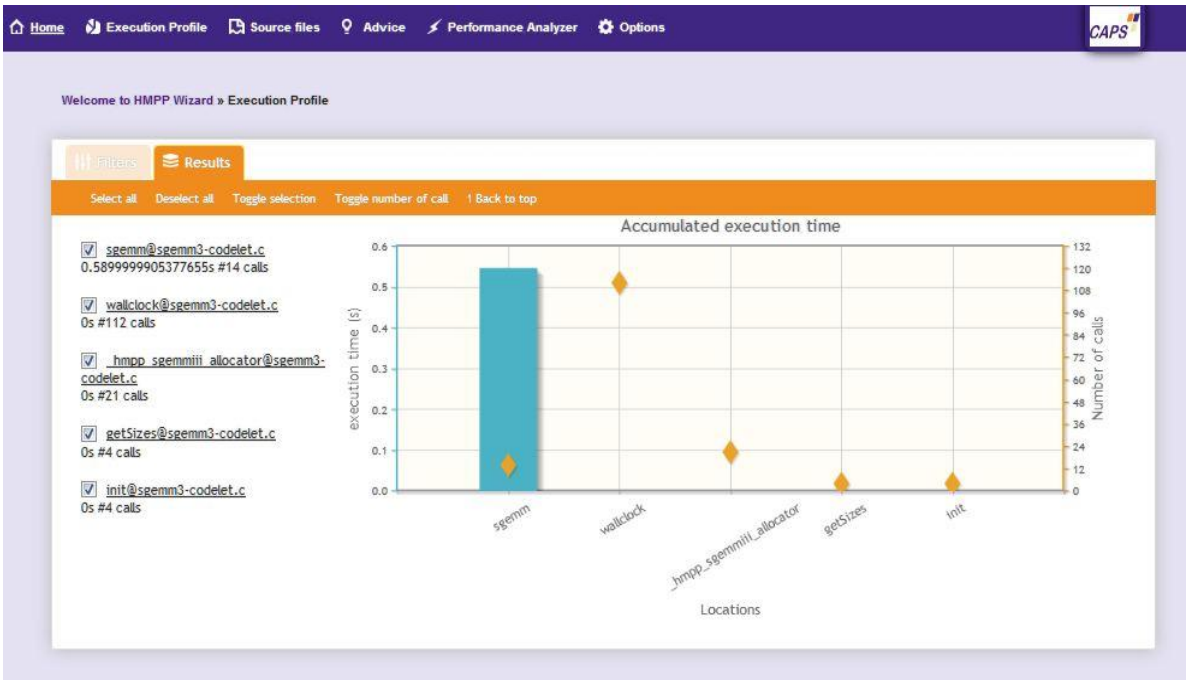


Figure 10 - Execution Profile - Result view

4.2.5 Performance Analyzer View

It offers to the user a visualization of the execution of the kernels on the GPU. The filter tab enables the selection of the directory containing profiled files. Only the files and directories having profiling information on the accelerator are displayed.

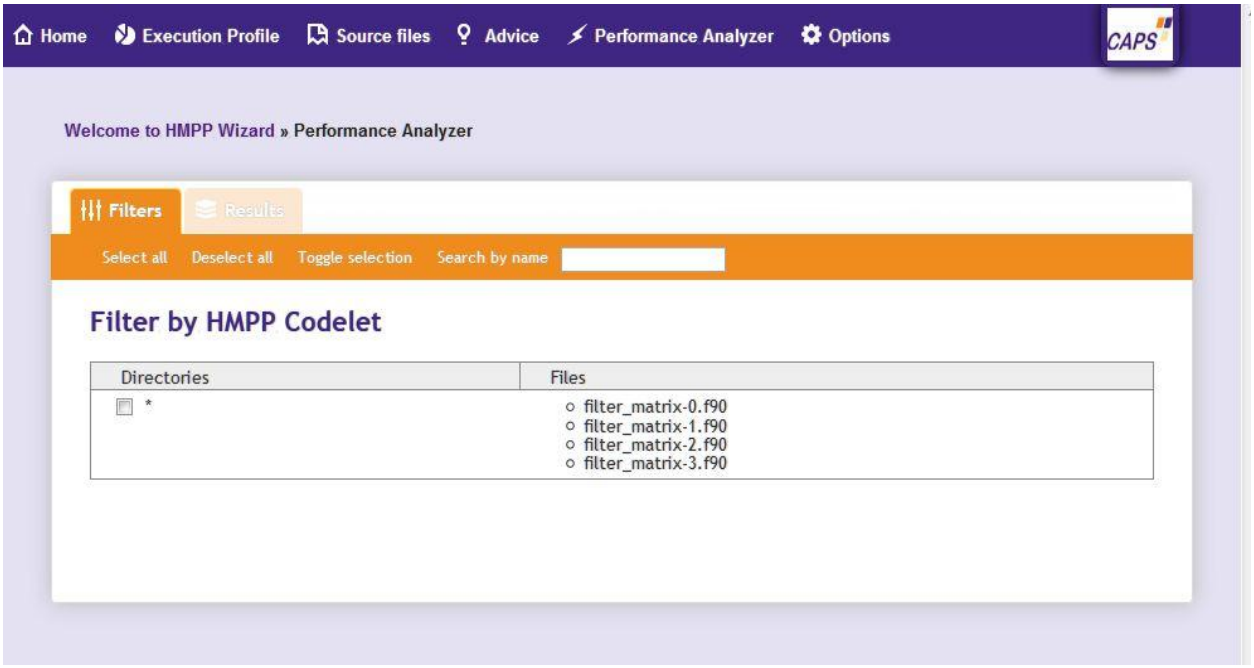


Figure 11 - Performance Analyzer - Filter view

Then for each file, you can display the GPU execution time spent by each codelet.

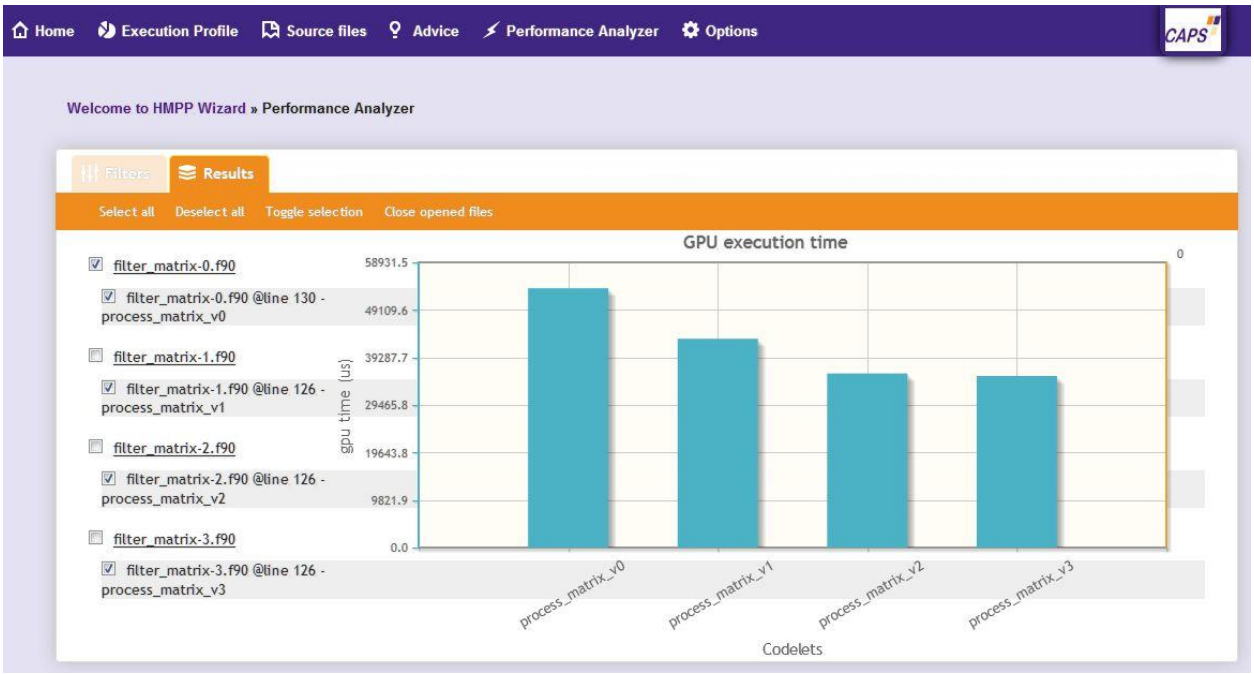


Figure 12 - Performance Analyzer - GPU execution time

By clicking on one file displayed, user has access to all the detailed information.

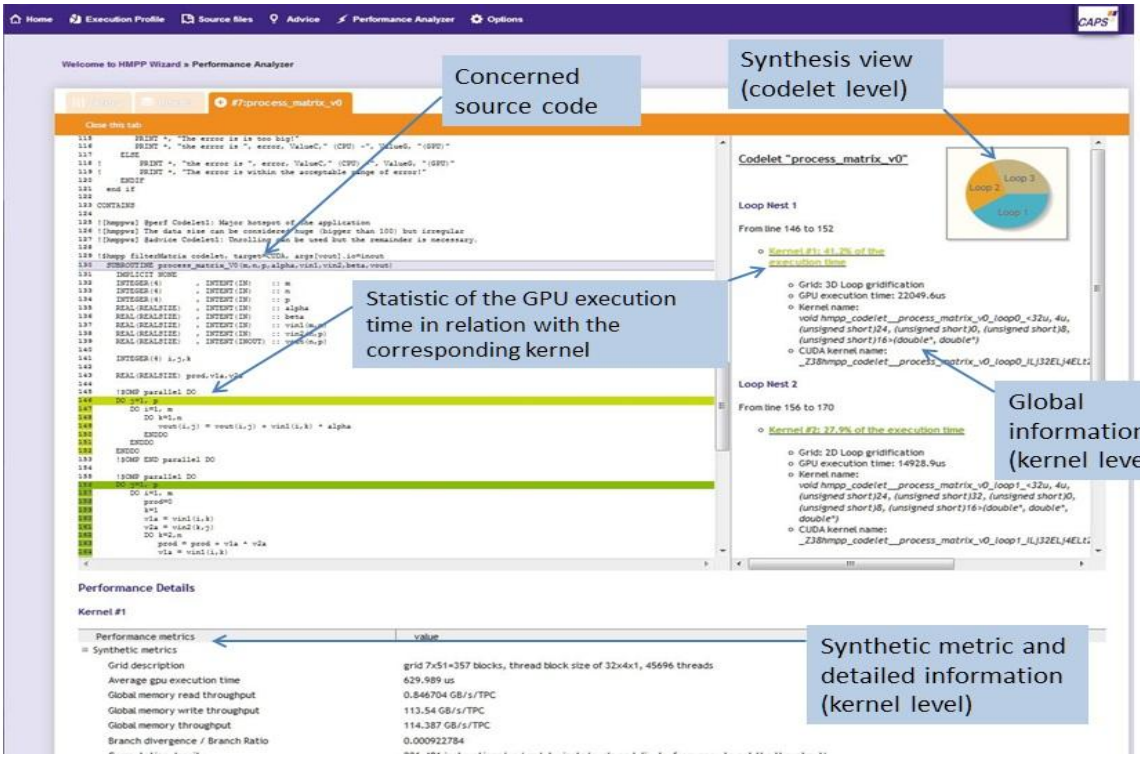


Figure 13 - PErformance Analyzer - GPU Kernel detailed view

4.2.6 Option Dialog

By clicking on the Options item, a dialog window appears letting you select the language and number of categories to display per page.

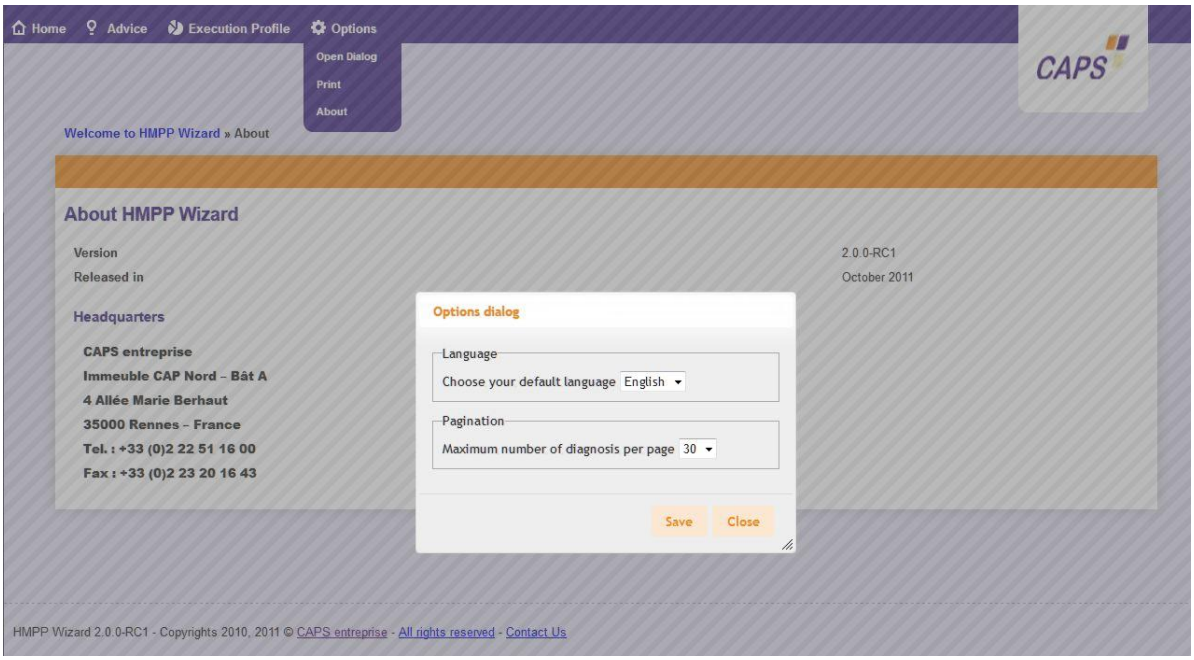


Figure 14 - Options page

5 Bibliography

[R1]	HMPPWorkbench-2.5_HMPP_Directives_ReferenceManual.pdf, CAPS entreprise, 2011.
[R2]	HMPPWorkbench-2.5_HMPPCG_Directives_ReferenceManual.pdf, CAPS entreprise, 2011
[R3]	HMPPWorkbench-2.5_Windows_Manual.pdf, CAPS entreprise, 2011
[R4]	HMPPWorkbench-2.5_Linux_Manual.pdf, CAPS entreprise, 2011
[R5]	CAPS Methodology Code Porting, CAPS entreprise, 2011.
[R6]	HMPPWorkbench-2.5_License_InstallationGuide.pdf, CAPS entreprise 2011
[R7]	Compute_Visual_Profiler_User_Guide.pdf, User Guide, NVIDIA, May 2011
[R8]	http://sourceware.org/binutils/docs-2.18/gprof/index.html, gprof official website

6 Appendix

6.1 Programming Diagnoses and Advices

Next sections describe diagnoses and advices provided by HMPP Wizard. These one are provided into two separate sections:

- One dedicated to the diagnoses that prevent HMPP to generate efficient GPU code
- One dedicated to performance improvement of HMPP generated code

6.1.1 Undefined Control Structure (switch case, goto/labels)

Diagnosis	Unstructured flow operations disrupt the compiler analysis and make the code generation difficult or impossible.
Advice	Use structured flow operations ('if' statements, ...) instead.

Diagnosis reason

Scope	Criteria
Codelet	A fully qualified HMPP codelet
Kernel	A qualified kernel with a number of dimensions `n` greater than 1
Loop or Grid	None
Access	None
Other	The loop body contains non regular flow operations that inhibit standard dependency analysis and sometimes code generation.

6.1.2 Induction Variable not found

Diagnosis	Induction variable not found
Advice	Re-write the loop: the current form hides the induction variable
Advice	Replace the loop operation by a proper 'for' loop

Diagnosis reason

Scope	Criteria
Codelet	A qualified HMPP codelet
Kernel	Kernel with a number of dimensions `n` greater than 1
Loop or Grid	A loop failing the validation phase with one of the following reasons: • Loop format not supported or not recognized (do while loops) • Start expression not recognized • Increment constant not recognized • Stop expression not recognized • Symbols in start, stop and increment expressions are not the same.
Access	None
Other	None

6.1.3 Loop may not be Parallel

Diagnosis	The loop is not detected nor specified parallel
Advice	Check the parallelism of the loop

Diagnosis reason

Scope	Criteria
Codelet	A qualified HMPP codelet
Kernel	Kernel with a number of dimensions `n` greater than 1
Loop or Grid	• A loop not detected parallel • **And** not indicated parallel with hmppcg directives
Access	None
Other	None

6.1.4 Conditional Statement Inside a Kernel

Diagnosis	Multiple `if` statement found. Conditional expressions may impact the performance.
Advice	Use masks instead of guards in computations
Advice	Split the kernel in several loop nests having the same execution flow (taking the same path)

Diagnosis reason

Scope	Criteria
Codelet	A fully qualified HMPP codelet
Kernel	A qualified kernel with a number of dimensions `n` greater than 1
Loop or Grid	None
Access	None
Other	The loop body contains at least two encapsulated `if` statement in the kernel body.

6.1.5 2D Gridification not performed due to non-nested loop

Diagnosis	Computations between loops prevent the gridification
Advice	Move the computation inside the inner loop with a conditional
Advice	Move the computation before or after the loop nest in a new loop nest

Diagnosis reason

Scope	Criteria
Codelet	A qualified HMPP codelet
Kernel	• Kernel with a number of dimensions `n` greater than 1 • **And** with at least 2 qualified induction variables
Loop or Grid	• None
Access	• None
Other	Computations found between loops

6.2 Optimization Diagnoses and Advices

This section describes advices related to the performance improvement of HMPP generated code.

6.2.1 Reduction Inside a Kernel

Diagnosis	The symbol <symbol name> should be reduced inside the loop nest
Advice	Use the `hmppcg parallel reduce` directive

Diagnosis reason

Scope	Criteria
Codelet	A qualified HMPP codelet
Kernel	• A qualified kernel with a number of dimensions `n` greater than 1 • **And** with at least 2 qualified induction variables
Loop or Grid	• A loop not detected parallel • **And** not specified parallel with hmppcg directives
Access	• Access to a `n-1` dimension array • **And** each dimension is a linear expression using one qualified induction variable • **And** the memory access is a write access using an accumulation operation (+=, -=, …).
Other	None

6.2.2 Bad Memory Coalescing

This advice relies on the gridification and the CUDA memory coalescing access patterns.

Diagnosis	The computation using <symbol name> is globally not well coalesced
Advice	Insert a loop interchange HMPP directive in the loop nest
Statistic	Number of subscript accesses over X
Statistic	Number of subscript accesses over Y
Statistic	Number of constant subscript accesses over the grid of threads
Transformation	<Code output with the appropriate hmppcg permute directive>

Diagnosis reason

Scope	Criteria
Codelet	A fully qualified HMPP codelet
Kernel	• A fully qualified kernel with a number of dimensions `n` equal to 2 or 3 • **And** with 2 or 3 qualified induction variables
Loop or Grid	• A qualified grid of dimension 2 or 3
Access	• A write access to a `n` dimension array • **And** for each dimension a linear expression that uses exactly one qualified induction variable • **And** a set of read accesses to a `n` dimension array • **And** for each dimension of each read access, a linear expression that uses exactly one qualified induction variable • **And** a number of read accesses with the lowest subscript accessed over Y higher than the number of read accesses with the lowest subscript accessed over X
Other	None

6.2.3 Inefficient Memory Coalescing

Diagnosis	A stride greater than one in the X dimension of the gridification prevents a good memory coalescing
Advice	Reduce the stride of the loop over the induction variable

Diagnosis reason

Scope	Criteria
Codelet	A fully qualified HMPP codelet
Kernel	• A fully qualified kernel with a number of dimensions `n` than 1 • **And** with at least one qualified induction variable
Loop or Grid	• A qualified grid of dimension 1 or higher • A step of the iteration space over X higher than 1
Access	None
Other	None

6.2.4 Too Low Computation Density

This advice is emitted when the computation density is too low compared to the memory accesses. The method that uses an arbitrary formula that still needs to be tuned.

Diagnosis	The computation density is low
Advice	Input data should already be on the accelerator to get performance
Statistic	Number of array accesses
Statistic	Number of operations
Statistic	Number of intrinsic operations

Diagnosis reason

Scope	Criteria
Codelet	A fully qualified HMPP codelet
Kernel	A qualified kernel with a number of dimensions `n` greater than 1
Loop or Grid	None
Access	None
Other	The computation density score is computed using a formula based on a statistic analysis of the kernel. Elements contributing to a high score: • Number of operations • Number of intrinsic operations Elements contributing to a low score: • Number of memory accesses • Number of array accesses • Number of conditionals • Number of flow operations

6.2.5 2D Convolution Patterns

Diagnosis	A 2D convolution pattern was found writing <symbol name> over <induction variable Y> and <induction variable X>
Advice	You should consider using an optimized kernel for this access, more information in the HMPP Cookbook

Statistic	Minimum stencil offset over each induction variable
Statistic	Maximum stencil offset over each induction variable

Diagnosis reason

Scope	Criteria
Codelet	A fully qualified HMPP codelet
Kernel	• A fully qualified kernel with a number of dimensions `n` equal to 2 • **And** with 2 qualified induction variables
Loop or Grid	• A qualified grid of dimension 2
Access	• A memory access pattern made of a set of computations using a set of data from 2D region around a destination of the computation. The size of the memory region is the stencil.
Other	None

6.3 HMPP Performance Analyzer Metrics

The HMPP Performance Analyzer metrics are divided in two parts:

- **Synthetic metrics** based on the statistical information of the profiling traces. These metrics are calculated depending on the presence of certain events in the profiling file. They provide pertinent information to the kernel behavior analysis.
- **Raw metrics** extracted from the profiling traces. The metrics are exhaustively listed from the data found in the profiling traces.

6.3.1 Synthetic Metrics

6.3.1.1 Kernel Name

The name used to identify the kernel is the C++ demangled name as generated by HMPP. The real kernel name is the name used by the dynamic library and the NVidia Profiler.

6.3.1.2 Average GPU Execution Time

Display the arithmetic average of kernel execution time on the GPU. This metric is systematically selected by the NVidia™ profiler and therefore always available.

6.3.1.3 Gridification / Grid / Thread Block Size

Display a summary of the kernel grid properties. If sizes of both blocks and threads are available, then the kernel number of blocks in the grid and number of threads is computed.

For example:

```
grid 113x113=12769 blocks, thread block size of 8x8x1, 817216 threads
```

If only one size, blocks or threads, is available, only their respective number is computed.

6.3.1.4 Global Memory Read Throughput

Display the read throughput of the global memory in Giga bytes per second per TPC. The computation is the following: (gld_32*32 + gld_64*64 + gld_128*128) / gputime. The TPC is the Texture Processing Cluster: a group composed of 1 to 3 multi-processors. The ‘gld_xx’ metrics count the number of memory transactions operated by the TPC.

This metric is for the moment restricted to CUDA architectures 1.2 and 1.3.

6.3.1.5 Global Memory Write Throughput

Display the write throughput of the global memory in Giga bytes per second per TPC. The computation is the following: (gst_32*32 + gst_64*64 + gst_128*128) / gputime. The TPC is the Texture Processing Cluster: a group composed of 1 to 3 multi-processors. The ‘gst_xx’ metrics count the number of memory transactions operated by the TPC.

This metric is for the moment restricted to CUDA architectures 1.2 and 1.3.

6.3.1.6 Global Memory Throughput

Display the overall throughput of the global memory in Giga bytes per second per TPC. The computation is the following: (gld32*32 + gld_64*64 + gld_128*128 + gst32*32 + gst_64*64 + gst_128*128) / gputime. The TPC is the Texture Processing Cluster: a group composed of 1 to 3 multi-processors. The 'gld_xx' and 'gst_xx' metrics count the number of memory transactions operated by the TPC.

This metric is for the moment restricted to CUDA architectures 1.2 and 1.3.

6.3.1.7 Load/Store Execution Density

Display the number of load or store transactions executed by µs in the kernel. The computation is the following: (gld32 + gld_64 + gld_128 + gst32 + gst_64 + gst_128) / gputime.

6.3.1.8 Branch Divergence / Branch Ratio

Display the ratio between the number of divergent branches over the number of branches. This ratio should be kept low unless the number of branches is small.

6.3.1.9 Computation Density

Display the number of instructions executed by µs in the kernel.

This metric is proportional with the instruction throughput ratio of the kernel. The real throughput ratio can be computed multiplying the computation density by the maximum number of instructions issued per cycle and divided by the frequency.

6.3.1.10 Load/Store Code Density

Display the ratio between the number of load or store transactions over the total number of instructions. The computation is the following: (gld_32 + gld_64 + gld_128 + gst_32 + gst_64 + gst_128) / instructions.

6.3.2 Raw Metrics

The raw metrics are provided by NVIDIA profiling traces. For further details, please refer to the standard CUDA SDK documentation.

HMPP Performance Analyzer exhaustively lists all metrics found in the traces of each kernel:

- Number of measures: the number of profiling trace entries used to calculate the statistics.
- Total: the sum of all trace values.
- Arithmetic average: the total divided by the number of measures.
- Maximum value among all trace values.
- Minimum value among all trace values.

6.4 Performance Analyzer : CUDA counters

Below, some information regarding the various counters used for each run. For further details, please refer to [R7].

6.4.1 1.3 NVIDIA GPU Architecture

Run #1	Gridsize	Number of blocks in a grid along the X and Y dimensions for a kernel launch.
	Threadblocksize	Number of threads in a block along the X, Y and Z dimensions for a kernel launch.
	Regperthread	Number of registers used per thread for a kernel launch.
	divergent_branch	Number of divergent branches within a warp. This counter is incremented by one if at least one thread in a warp diverges (that is, follows a different execution path) via a data dependent conditional branch. The counter is incremented by one at each point of divergence in a warp.
	branch	Number of branches taken by threads executing a kernel. This counter is incremented by one if at least one thread in a warp takes the branch. Note that barrier instructions (__syncThreads()) also get counted as branches.
	sm_cta_launched	Number of threads blocks launched on a multiprocessor.
	instructions	Number of instructions executed.
Run #2	Gridsize Threadblocksize Regperthread	See above
	local_load	Number of local memory load transactions. Each local load request will generate one transaction irrespective of the size of the transaction.
	local_store	Number of local memory store transactions; incremented by 2 for each 32-byte transaction, by 4 for each 64-byte transaction and by 8 for each 128-byte transaction for compute devices having compute capability 1.x. It is incremented by 1 irrespective of the size of the transaction for compute devices having compute capability 2.0.
	gld_32b	Number of 32 byte global memory load transactions; incremented by 1 for each 32 byte transaction.
	gld_64b	Number of 64 byte global memory load transactions; incremented by 1 for each 64 byte transaction.
Run #3	Gridsize Threadblocksize Regperthread	See above

	gld_128b	Number of 128 byte global memory load transactions; incremented by 1 for each 128 byte transaction.
	gst_32b	Number of 32 byte global memory store transactions; incremented by 2 for each 32 byte transaction.
	gst_64b	Number of 64 byte global memory store transactions; incremented by 4 for each 64 byte transaction.
	gst_128b	Number of 128 byte global memory store transactions; incremented by 8 for each 128 byte transaction.

6.4.2 2.0 NVIDIA GPU Architecture

Run #1	Gridsize	Number of blocks in a grid along the X and Y dimensions for a kernel launch.
	Threadblocksize	Number of threads in a block along the X, Y and Z dimensions for a kernel launch.
	Regperthread	Number of registers used per thread for a kernel launch.
	threads_launched	Number of threads launched on a multiprocessor.
	warps_launched	Number of warps launched on a multiprocessor.
	fb_subp0_read_sectors	Number of read requests sent to sub-partition 0 of all the DRAM units
	fb_subp0_write_sectors	Number of write requests sent to sub-partition 0 of all the DRAM units
Run #2	Gridsize Threadblocksize Regperthread	See above
	gld_request	Number of global memory load requests.
	gst_request	Number of global memory store requests.
	local_load	Number of local memory load transactions. Each local load request will generate one transaction irrespective of the size of the transaction.
	local_store	Number of local memory store transactions. It is

		incremented by 1 irrespective of the size of the transaction for compute devices having compute capability 2.0.
	fb_subp1_read_sectors	Number of read requests sent to sub-partition 1 of all the DRAM units
	fb_subp1_write_sectors	Number of read requests sent to sub-partition 1 of all the DRAM units
Run #3	Gridsize Threadblocksize Regperthread	See above
	l1_global_load_hit	Number of global load hits in L1 cache.
	l1_global_load_miss	Number of global load misses in L1 cache.
Run #4	Gridsize Threadblocksize Regperthread	See above
	l1_local_load_hit	Number of local load hits in L1 cache.
	l1_local_load_miss	Number of local load misses in L1 cache
Run #5	Gridsize Threadblocksize Regperthread	See above
	inst_executed	Number of instructions executed, do not include replays.
	l1_local_store_hit	Number of local store hits in L1 cache.
	l1_local_store_miss	Number of local store misses in L1 cache.
Run #6	Gridsize Threadblocksize Regperthread	See above
	branch	Number of branches taken by threads executing a kernel. This counter is incremented by one if at least one thread in a warp takes the branch. Note that barrier instructions (__syncThreads()) also get counted as branches.
Run #7	Gridsize Threadblocksize	See above

	Regperthread	
	divergent_branch	Number of divergent branches within a warp. This counter is incremented by one if at least one thread in a warp diverges (that is, follows a different execution path) via a data dependent conditional branch. The counter is incremented by one at each point of divergence in a warp.

Headquarters - FRANCE
Immeuble CAP Nord
4A Allée Marie Berhaut
35000 Rennes
France
Tel.: +33 (0)2 22 51 16 00
info@caps-entreprise.com

CAPS - USA
4701 Patrick Drive Bldg 12
Santa Clara
CA 95054

Tel.: +1 408 550 2887 x70
usa@caps-entreprise.com

CAPS - CHINA
Suite E2, 30/F
JuneYao International Plaza
789, Zhaojiabang Road,
Shanghai 200032
Tel.: +86 21 3363 0057
apac@caps-entreprise.com