Automatic Online Tuning (AutoTune)

D3.2
GPGPU Performance and Energy Efficiency Properties
Houssam Haitof
Eduardo César

Date of preparation (latest version): February 25th, 2014
Copyright © 2014 The AutoTune Consortium

The opinions of the authors expressed in this document do not necessarily reflect the official opinion of the European Commission.
PROJECT INFORMATION

Project acronym: AutoTune
Project full title: Automatic Online Tuning
Grant agreement no: 288038
Call (part) identifier: FP7-ICT-2011-7
Funding scheme: Collaborative project

DOCUMENT INFORMATION

Deliverable Number: D3.2
Deliverable Name: Report on GPGPU performance and energy efficiency properties
Authors: Houssam Haitof (TUM), Robert Mijaković (TUM), Laurent Morin (CAPS), Enes Bajrovic (UNIVIE), Carmen Navarrete (LRZ), Eduardo César (UAB)
Responsible Author: Houssam Haitof (TUM), Eduardo César (UAB)
e-mail: eduardo.cesar@uab.cat
Reviewed by: Michael Gerndt (TUM)
Approved by: Matthias Brehm (LRZ)
Keywords: Periscope Tuning Framework, Online Performance Analysis
WP/Task: WP3 / Task 3.2, 3.3, 3.4

DISSEMINATION LEVEL

<table>
<thead>
<tr>
<th>PU</th>
<th>Public</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PP</td>
<td>Restricted to other programme participants (including the Commission Services)</td>
<td></td>
</tr>
<tr>
<td>RE</td>
<td>Restricted to a group specified by the consortium (including the Commission Services)</td>
<td></td>
</tr>
<tr>
<td>CO</td>
<td>Confidential, only for members of the consortium (including the Commission Services)</td>
<td></td>
</tr>
</tbody>
</table>
DOCUMENT STATUS

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Status</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2013-10-15</td>
<td>Prepared for 2nd Review Meeting</td>
<td>Revision required based on feedback from the project reviewers: “…the full specification of Property Expressions should be given, i.e., syntax of metric operators, quality operators, etc. In addition, semantics of new extensions like the “pipeline” of “Code Region” should be presented”</td>
</tr>
<tr>
<td>1.5</td>
<td>2014-02-17</td>
<td>Description of semantics of new extensions</td>
<td>UNIVIE modified section 6 to include the semantics of the “pipeline” extension of “Code Region”</td>
</tr>
<tr>
<td>1.7</td>
<td>2014-02-18</td>
<td>Added TPI and analysis refinement</td>
<td>TUM added the current definition of the TPI and inserted details about the refinement step in the energy analysis.</td>
</tr>
<tr>
<td>1.9</td>
<td>2014-02-20</td>
<td>Full specification of Property Expressions</td>
<td>New Section (4.1.4) is included by CAPS partner with the full syntax description of Quality Indexes and Metrics.</td>
</tr>
<tr>
<td>2.0</td>
<td>2014-02-21</td>
<td>Revised version</td>
<td>Current document responsible (UAB) generates version for internal review.</td>
</tr>
<tr>
<td>2.0</td>
<td>2014-02-23</td>
<td>Review comments</td>
<td>Review comments sent back to UAB.</td>
</tr>
<tr>
<td></td>
<td>2014-02-25</td>
<td>Approved</td>
<td></td>
</tr>
</tbody>
</table>

COPYRIGHT NOTICES

© 2014 AutoTune Consortium Partners. All rights reserved. This document is a project document of the AutoTune project. All contents are reserved by default and may not be disclosed to third parties without the written consent of the AutoTune partners, except as mandated by the European Commission contract 288038 for reviewing and dissemination purposes. All trademarks and other rights on third party products mentioned in this document are acknowledged as own by the respective holders.
Executive Summary

The AutoTune project is developing the Periscope Tuning Framework (PTF) that combines performance analysis and tuning to facilitate performance engineering of parallel applications. The approach is based on tuning plugins that encapsulate the knowledge about the tuning of applications with respect to a specific aspect. The selection of the plugins and the tuning process of plugins can significantly profit from the performance analysis capabilities of PTF. This document summarizes the work done in WP 3 Performance Analysis in the second year of the project.

In the first year, the monitoring system of PTF was extended for measuring GPU performance data, energy consumption, and performance metrics related to high-level heterogeneous patterns. In the second year, the work focused on implementing performance properties and search strategies for OpenCL/HMPP codes as well as for energy consumption tuning.

The performance analysis strategy for OpenCL/HMPP has been implemented in coordination with the development of the HMPP Codelet Tuning Plugin. A flexible and easy to use design is used to integrate PTF with the CAPS compiler system. The analysis strategy is not bound to any specific accelerator architecture but covers a broad spectrum of current and future architectures and supports the OpenHMPP and the OpenACC programming models.

The analysis strategy supporting energy consumption tuning was implemented based on the Enopt library developed at LRZ for energy measurements on SuperMUC. The strategy searches for new properties to guide the energy tuning plugin in finding good frequency and governor settings for regions of the application that minimize the energy consumption. It identifies regions that are worth to be explored with respect to the consumed energy based on the energy consumption and the execution granularity. The major advantage of combining the energy analysis and tuning is the reduction of the search space to be explored by the plugin.

In addition, the SIR representation of the target application was extended for the HMPP Codelet Tuning Plugin and the High Level Pattern Tuning Plugin. It is now generated by the HMPP compiler and the pattern code transformation system. The extended information in the SIR file is an input to these two plugins.

PTF was extended to combine analysis and tuning in two ways. First, a tuning plugin can run a pre-analysis on the original execution at the beginning of a tuning step to obtain data to guide the tuning process. Second, an analysis can be run while a certain tuning scenario is executed. This approach will result in performance properties for the scenario that give detailed insight into the effect of the tuning actions.
# Table of Contents

1. Introduction ................................................................................................................. 9
2. Periscope Tuning Framework ....................................................................................... 9
   2.1 Analysis Agent .......................................................................................................... 9
   2.2 Strategies .................................................................................................................. 10
   2.3 Properties ................................................................................................................ 11
   2.4 Standard Intermediate Representation (SIR) .......................................................... 11
3. Extensions to the tuning plugin interface for analysis strategies ............................... 11
4. Performance analysis strategy for OpenCL/HMPP ...................................................... 13
   4.1 The HMPP Profiling Module design ....................................................................... 13
   4.1.1 The Event Recorder Module .............................................................................. 14
   4.1.2 The Quality Index Builder and the definition of Quality Expressions ............... 15
   4.1.3 Quality Indexes and the HMPP Profiling Module .............................................. 15
   4.1.4 Quality Indexes full grammar specification ...................................................... 16
   4.1.5 OpenCL/HMPP Analysis Strategy ................................................................... 19
   4.2 Evaluation of the performance analysis strategy for OpenCL/HMPP for NVidia GPUs 20
5. Energy efficiency analysis strategy ............................................................................ 20
   5.1 SuitedForEnergyConfiguration property ............................................................. 22
   5.2 EnergyInefficient property ...................................................................................... 22
   5.3 MemoryBound property .......................................................................................... 23
6. Extensions to the SIR for the High Level Pattern Tuning Plugin ............................... 23
7. Summary and future work ........................................................................................... 25
8. Bibliography ................................................................................................................. 26
1 Introduction

The AutoTune project develops the Periscope Tuning Framework (PTF) including several plugins targeting performance improvements as well as to reduce energy consumption of applications. One of the main advantages of PTF over other tuning frameworks is its capability to combine tuning and analysis strategies to simplify and speed up the tuning process. To support the plugins with required information a number of performance analysis strategies are developed.

This document starts with an overview of the relevant components of PTF. It then describes the general extensions to PTF enabling to integrate performance analysis strategies with tuning plugins. These extensions allow to automatically perform an analysis at the beginning of each tuning step. This can result in properties that guide the performance tuning by the plugins. We also integrated support to run an analysis within an experiment executing a certain tuning scenario. This is used to get more detailed information about the application behavior for this scenario.

Next, the analysis strategies for the HMPP Codelet Tuning plugin and the Energy Consumption via CPU Frequency Tuning plugin are presented. These sections cover the provided properties, the external components used in the implementation, extensions required to the standard intermediate program representation (SIR), its generation, and their integration with the plugins. Finally the extensions to the SIR for the Pattern Tuning plugin are presented.

2 Periscope Tuning Framework

This section gives a short overview of the main components of the analysis infrastructure of PTF. PTF is based on Periscope, a distributed automatic online performance analysis system developed by Technische Universität München (TUM) at the Chair of Computer Architecture (LRR).

PTF consists of four major entities, a user interface, the frontend, a hierarchy of analysis agents, and the monitoring library. The PTF runtime is controlled by the frontend, it loads the tuning plugins and controls the overall tuning process. It starts a set of agent processes and the parallel application. The agents propagate commands and gather data in a hierarchical and scalable manner. The main focus of this report is on analysis strategies that guide the analysis agent.

2.1 Analysis Agent

Our distributed performance analysis and tuning system is composed of a set of analysis agents (figure 1) that cooperate in the detection and tuning of performance problems. The leaf agents (the lowest level of the hierarchy) are responsible for the collection and analysis of performance data from one or more application processes. They are guided by analysis strategies which search automatically for predefined performance properties, e.g., load imbalance at a barrier in a certain process. The measurements of performance data required to prove properties are performed by the MRI Monitor attached to the application.

The agent’s main components are the agent control, i.e. Process-local Analysis Decision, the analysis strategies, and the experiment control, i.e. the Performance Database and the Data Provider.

The analysis consists of one or more search steps. Each search step starts by determining the set of candidate properties which will be checked in the next experiment based on the set of properties found in the previous
step. At the beginning, the set of evaluated properties is empty. After the candidate set was determined, the agent control starts a new experiment. The experiment control accesses all the properties in the candidate set and checks whether the required performance data for proving the property are available in the Performance Database. If not, the Performance Database sends measurement requests to the Data Provider which configures the monitor via MRI measurement requests. The requests, e.g. measure the execution time for a region, are stored and processed by the monitor linked to the application.

At the end of the experiment, the candidate properties are evaluated and found properties are stored in a set of found properties. Found properties with a severity above a predefined threshold are propagated through the hierarchy to the frontend. The found properties are finally reported through the console and a special properties file if PTF is just used for performance analysis. If PTF is used to tune an application based on the tuning plugins developed in AutoTune, the found properties are stored in an analysis property pool located in the frontend. From there, the tuning plugin can access the properties and take the analysis results into account in its tuning strategy.

Figure 1: The analysis agents’ internal structure with components and interfaces.

2.2 Strategies

The analysis strategies determine the overall search for performance problems. They define in which order an analysis agent investigates the multidimensional search space of properties and program regions. Periscope supports two types of analysis strategies, single and multi-step strategies. Single-step strategies
generate all hypotheses in the initial candidate set and evaluate them after the first experiment. Multi-step strategies start searching for high level properties by generating hypotheses for outer regions and refine found performance problems by continuing with more precise properties, either with respect to the cause of the problem, e.g., which cache level is responsible, or to the region, e.g. analyzing nested regions.

### 2.3 Properties

A performance property (e.g. load imbalance, communication, cache misses, redundant computations, etc.) characterizes a specific performance behavior of a program and can be checked by a condition. Conditions are associated with a confidence value (between 0 and 1) indicating the degree of confidence about the existence of a performance property. In addition, for every performance property a severity value is provided that specifies the importance of the property. The higher the severity, the more important or severe a performance property is.

A performance property is called a performance problem, if and only if its severity is greater than a tool-defined threshold. The most important performance problem is called the performance bottleneck.

### 2.4 Standard Intermediate Representation (SIR)

Automatic performance analysis and tuning does not only require dynamic information but also benefits from information about the program’s static structure, e.g., the nesting of regions or data structure usage information. In our environment such information is generated by the instrumenter in form of a Standard Intermediate Representation (SIR). The SIR was defined by the APART group and is an XML format. The SIR file of an application contains the structure of the instrumented application, the instrumented code regions and data structures. To support automatic tuning, the SIR was extended to contain also the information about new region types and the tuning parameters. The extensions of the SIR file are presented in the context of the tuning plugins in the following sections.

### 3 Extensions to the tuning plugin interface for analysis strategies

The goal of the AutoTune project is to combine performance analysis and tuning in a single tool. This requires to provide support for the tuning plugins in PTF to start performance analysis strategies and to obtain the resulting properties that can be taken into account in the tuning process. We extended the Tuning Plugin Interface (TPI) with two ways to call analysis strategies. First, the plugin can run an analysis on the standard execution of the code in each tuning step. We call this the tuning step pre-analysis. Second, a performance analysis can be combined with the execution of a tuning experiment. This results in performance properties relevant to the program regions in a certain tuning scenario.

The tuning step pre-analysis allows plugins to run a plugin-specific or a general Periscope analysis first, before the tuning is performed based on the analysis results. For example, the MPI analysis of Periscope indicates the relative importance of MPI bottlenecks for individual call sites. This information can be used by the plugin to select those call sites that will be tuned subsequently with respect to the MPI library parameters.

The source code in Listing 1 shows the extended Tuning Plugin Interface:
class IPlugin {
    public:
        // Plugins are required to implement the full API
        virtual void initialize(string sirFilePath) = 0;
        virtual void startTuningStep(void) = 0;
        // Plugin function for preanalysis
        virtual bool analysisRequired(StrategyRequest** strategy) = 0;
        virtual void createScenarios(void) = 0;
        virtual void prepareScenarios(void) = 0;
        // Plugin function defining an experiment
        virtual void defineExperiment(int numprocs, bool *analysisRequired,
                                        StrategyRequest** strategy) = 0;
        virtual bool restartRequired(string *env, int *np, string *cmd,
                                      bool *instrumented) = 0;
        virtual bool searchFinished(void) = 0;
        virtual void finishTuningStep(void) = 0;
        virtual bool tuningFinished(void) = 0;
        virtual void getAdvice(void) = 0;
        virtual void finalize(void) = 0;
        virtual void terminate(void) = 0;
    }
}

Listing 1: Extended Tuning Plugin Interface.

To enable this pre-analysis in each tuning step, the plugin provides a function that is called by the PTF frontend after starting a tuning step. The function analysisRequired of the TPI returns true, if a preanalysis is to be executed. The StrategyRequest defines which analysis strategy is requested. Before the PTF frontend triggers the creation of scenarios, it will execute the requested analysis. The resulting performance properties are returned to the plugin for inspection in a special property pool.

The second TPI extension allows to execute a Periscope analysis as part of the scenario execution experiment. The motivation for this feature is to combine the effect of the execution variant on the tuning objective value for the tuned region with an in-depth analysis of the application behavior. This can be for example PTF’s MPI analysis strategy to not only obtain the execution time of the phase region of the application but also to receive performance properties for the MPI calls. A comparison of the returned properties with the properties obtained in the tuning step pre-analysis for the original program execution can give a detailed insight on the effect of selecting special algorithms implementing collective MPI operations.

The analysis for a tuning experiment is requested via an extension to the defineExperiment function of the TPI. It now provides two new output parameters. The strategy request is return via strategy and the flag analysisRequired indicates whether an analysis is to be executed during the experiment.

The two analysis strategies developed for the OpenCL/HMPP plugin and the Energy Tuning Plugin are examples for plugin-specific analysis strategies. The extensions to the TPI are documented in D4.2 on the prototype plugins.
4 Performance analysis strategy for OpenCL/HMPP

The performance analysis strategy for OpenCL/HMPP has been implemented in coordination with the development of the HMPP Codelet Tuning Plugin. It exploits all the efforts made for the integration of the CAPS compiler inside the Periscope Tuning Framework, and all the work done to make the system flexible and easy to use. It searches for performance properties in OpenCL/HMPP codes. For a better understanding of the rational and technical developments presented in this document, we refer to deliverable D4.2.

The original design of the performance analysis for OpenCL/HMPP has been improved to take into account the evolution of the HPC market concerning the usage of directive based programming models and the emerging hybrid many-core architectures like the XeonPHI. The OpenHMPP standard is now evolving with the OpenACC standard that is quickly spreading in the domain of hybrid many-core programming. Furthermore, the nature of the problems addressed in the programming of very different hybrid many-core accelerators, such as GPUs or the XeonPHI, forces the tool developers to propose a flexible approach in the design of the performance analysis strategy. The developments made in this project take into account the integration of these evolutions to propose a useful performance analysis system for the development of plugins. We made the choice to implement a performance analysis strategy non-specific to a given architecture family but easy to configure by expert users for future architectures and the OpenACC programming model.

The core of an analysis strategy in PTF is the definition of properties and the automatic search for such properties during program execution. In a first section, we present a parameterized OpenCL/HMPP performance analysis based on an XML description of the properties to be evaluated at runtime, and the specification of the advices or diagnosis that can be inferred about the application. This approach was implemented in the HMPP Profiling Module, a library fully integrated in PTF and provided with the CAPS compiler. In a second section, we present a property defined using this approach and targeting the performance analysis of CUDA kernels relative to memory transfers.

4.1 The HMPP Profiling Module design

The CAPS compiler measurement system and the definition of the properties are regrouped in the library HMPP Profiling Module. This module provides support for the measurement of metrics, e.g., the execution time of codelets, and the evaluation of performance properties based on these measurements. The module includes a Quality Index Builder that creates properties from abstract expressions, called Quality Expressions. The library also supports the CAPS PTF Interface that was developed to integrate the CAPS compiler with PTF.

In Figure 2, a synthetic view of the HMPP Profiling Module linking the CAPS runtime and the MRI Monitor is shown. Three major parts compose the library. First, an Event Recorder module makes the link with the CAPS runtime. This link enables the monitoring of all CAPS runtime and application events during the application execution, such as the start of a CUDA kernel or the transfer of data between the CPU and the GPU. Then, the Property Builder takes Quality Expressions from PTF and creates the necessary event record entries and measurement table entries to evaluate the given Property Expression. Finally, the CAPS PTF interface links Periscope’s MRI Monitor with the application. This part is Periscope-specific and might be
extracted to make an independent library in the final implementation. Details are provided in the prototype integration section.

Figure 2: HMPP Profiling Module

### 4.1.1 The Event Recorder Module

The Event Recorder enables the monitoring of all CAPS runtime and application events during the application execution using the HMPP Profiling API [3]. The HMPP profiling interface enables the analysis of a wide set of events. Each generated event describes the beginning, the asynchronous wait, or the end of an operation in the application, e.g., the start and end of a CUDA kernel. An event is defined by its family and its name. A family marks a group of consistent events, for example “all events related to OpenACC directives” or “all events related to low level CUDA operations”.

Once an event is chosen, we also have to choose the data we want to monitor about the event. This is defined by a metric and an aggregation operator on that metric. So, in addition to the event monitoring system, the module provides a set of metrics and operators for all recorded events; each metric covering various aspects of the behavior of the application.

In the implementation, a metric is defined by a name and an aggregation operator. For example, the metric’s name can be “time” for the timing of events (based on time stamps), or “size” for the operation size related to the event (typically for transfers or allocations).

Aggregation operators are mathematical operations applied to the data set generated during the profiling. For example, they can be one of the following: the maximum value, the minimum value, the average value, the accumulated value.

These metrics and aggregation operators will be extended in the future to cover more aspects of the application behavior.
4.1.2 The Quality Index Builder and the definition of Quality Expressions

The Property Builder receives from the MRI Monitor a string expression containing the properties to evaluate. It is a regular computational expression made of operators and operands. It uses as operands the events, the metrics, and the aggregation operators available in the Event Recorder, and computes them. By default none of them are activated, they are activated one by one depending on the property requested.

The simplified view of the syntax of a Property Expression is the following (the full one implements the precedence of the property operators):

```
<event name>:<metric operator><metric name>[([<quality operator><event name>:<metric operator><metric name>]) | <immediate>]*
```

Listing 2: CAPS Quality Expression syntax. In red, events; in green, the aggregation operator; in blue the metric; and in orange, an operator applied to the property.

For example, to express the couple “maximum size of a CUDA transfers”, and “CUDA kernel average time”, the expression is the following:

```
CudaMemoryTransfer:size;CudaMemoryTransfer:-time
```

Note that in that example, we use a “list” quality operator with the character “;”.

4.1.3 Quality Indexes and the HMPP Profiling Module

The current prototype implements three out of the four event families designed for the HMPP Profiling module. We do not foresee any critical issues for the implementation of the OpenCL family that should be implemented in the next period.

The event families currently available in the HMPP Profiling Module are:

- OpenHMPP Events
- OpenACC Events
- Cuda Events

Likewise, the current prototype offers three kinds of metrics, and for each from 3 to 4 aggregation operators. We do not plan to extend the current list of metrics during the next period. The available metrics are the following:

- Time values (accumulated time, average time, maximum time, minimum time)
- Size values (accumulated size, average size, maximum size, minimum size)
- Bandwidth values, i.e. aggregation of sizes divided by the event times (average bandwidth, maximum bandwidth, minimum bandwidth)

Currently, only the list operator “;” is supported in Quality Expressions. During the next period, a wider list of property operators will be implemented, and in particular the relational operators that will permit to create triggers.
4.1.4 Quality Indexes full grammar specification

The Property Expression syntax is built using a typical grammar for the computation of arithmetic expressions with support for event measurements. Not all the grammar is yet available in the prototype but the full implementation will be finished during the next period. The grammar is described using the BNF format. Rules in blue are implemented.

- The start unit typically returns a list of quality indexes, but can return an empty list.

```
<QualIndexExpression> ::= /* <Nothing>. */
| <QualIndexExpressionList>
```

- A list of quality indexes is linear delimited by “;”. A non-terminal `<QualIndexEntryUnit>` is defined as an entry for expressions in parentheses (defined later in the grammar).

```
<QualIndexExpressionList> ::= <QualIndexExpressionList> ';
<QualIndexEntryUnit>
| <QualIndexEntryUnit>
<QualIndexEntryUnit> ::= <QualIndexEntryCompare>
```

- The following rules define a set of comparison operators used to detect when some measurements reach some given thresholds. In the future, the grammar should be extended to have conditional operators in order to adjust measurements with the actual behavior of the events measured.

```
<QualIndexEntryCompare> ::= <QualIndexEntryCompare> '<'
<QualIndexEntryAddSub>
| <QualIndexEntryCompare> '>
<QualIndexEntryAddSub>
| <QualIndexEntryCompare> '"'<
<QualIndexEntryAddSub>
| <QualIndexEntryCompare> '">'
<QualIndexEntryAddSub>
| <QualIndexEntryCompare> '"="
<QualIndexEntryAddSub>
| <QualIndexEntryCompare> '"="
<QualIndexEntryAddSub>
```

- The typical arithmetic operations are supported with the appropriate precedence.
No unary operator is yet supported. However, for the definition of thresholds in comparisons, absolute double floating point values are defined. Precedence of computations can be controlled by the typical parenthesis syntax. A measurement is defined by the non-terminal `<QualIndexMeasure>`.

Measurements are defined by an event `<EventName>` and by a computation `<MetricComputation>` performed on the fly by the profiling module. The combination is stored as a double floating point value.

It is planned to support four major event families: the two high level programming standards OpenHMPP an OpenACC, and the two low level programming models used by the CAPS many-core compiler to program accelerators, CUDA and OpenCL.

Relying on the notion of Codelet, OpenHMPP events are grouped around the management of the accelerator device, the data, and the Codelet computation.
OpenACC events concern the major syntax of the programming model around parallel sections and kernel sections. Events related to the management of the memory are also available.

In addition to the transfers and to the kernel calls, CUDA events are more low level and are able to measure the cost of the GPU management: the streams and the contexts.

The OpenCL events support the same kind of events as CUDA with the addition of the on the fly “build” cost of CL kernel source codes.

Metric computations currently operate on the execution time of the events, and on the size of the object. Derived from these computation, the bandwidth is also available.
The time metric is available for all events. It is computed on the fly by the profiling module and supports four aggregation operators: ‘|’ for the sum, ‘~’ for the average (the sum divided by the event count), ‘+’ for the maximum value, ‘-’ for the minimum value.

The size metric is not applicable for all events. It is used typically for events related to the memory or objects (allocation, transfers). The same operators as for “time” are supported.

The bandwidth metric is applicable when the size metric can be evaluated. All operators except the sum are provided (the sum makes no sense for a bandwidth).

Metrics shall be easily extended if new computations are needed. The event families do not provide all hardware events but the events are grouped in categories to simplify their usage. This segmentation can be modified or extended in the future if the categories do not provide the necessary information for advanced users.

### 4.1.5 OpenCL/HMPP Analysis Strategy

A new OpenCL/HMPP Analysis Strategy has been implemented in PTF. It is triggered by the HMPP Codelet Tuning Plugin and identifies performance properties in HMPP codes. The Quality Expressions needed for the evaluation of the performance criteria are automatically forwarded by the plugin to the analysis strategy. The analysis strategy creates Periscope properties from these expressions and goes through the normal monitor configuration, application execution, and retrieval of the measurements. Then, the HMPP Profiling Module is in charge of gathering the measurements and providing the measurements to the analysis strategy. The strategy verifies the candidate properties and returns the found properties to the Codelet Tuning Plugin. Finally, the properties can be output to inform the application developer or used in a future version of the plugin to guide the search.
4.2 Evaluation of the performance analysis strategy for OpenCL/HMPP for NVidia GPUs

As described in the previous section, the objective of the performance analysis strategy for OpenCL/HMPP is to provide a parameterized framework for OpenACC and OpenHMPP applications for different families of accelerators. The parameters are specified via an XML file that will provide the Quality Expressions and the associated diagnosis. The current prototype does not fully support the load/unload of the XML file but we validated the approach with a hardcoded diagnosis in the HMPP Codelet Tuning Plugin specifically for NVidia GPUs. Further potential performance issues will also be analyzed in the near future to complete the proof-of-concept of the performance analysis strategy.

The proposed hardcoded diagnosis is the ratio between computation and data transfers of OpenHMPP Codelets using the NVidia CUDA architecture. The ratio must be lower than a threshold (10 in our case). The Quality Indexes used are:

- “CudaSynchronize:~time”: expresses the average time taken to finish the execution of the GPU kernels in the region,
- “CudaMemoryTransfer:~time”: expresses the average time taken to operate all memory transfers on the GPU for the execution of the GPU kernels in the region.

When a GPU kernel is executed, the memory transfer overhead must be as low as possible. The Periscope property is defined by the following formulas:

\[
\begin{align*}
\text{Condition} & = \text{severity} > 0.1 \\
\text{Confidence} & = 1 \\
\text{Severity} & = \frac{\text{CudaMemoryTransfer:~time}}{\text{CudaSynchronize:~time}}
\end{align*}
\]

The condition checks whether the computation time of the kernel is too small by comparing the severity to a predefined threshold. The confidence is always 1 and the severity is the ratio between the memory transfer time and the kernel execution time. This property might indicate that the kernel should better be run on the CPU instead of on the accelerator.

5 Energy efficiency analysis strategy

The Energy Consumption via CPU Frequency Tuning plugin tunes the application’s energy consumption by setting the processor frequency and governor appropriately. The search space, consisting of the different processor frequencies and the available governors, grows very fast when the settings are to be investigated for individual program regions. It is the goal of the energy analysis strategy to support the plugin in shrinking the search space by identifying regions with bad energy efficiency or which are not suited for this tuning technique. This is the case of regions with a very small granularity where the overhead for energy measurement and executing the tuning actions is too high.

The energy analysis strategy is a multi-step analysis strategy that searches for regions with three types of properties, i.e. SuitedForEnergyConfiguration, EnergyInefficient, and MemoryBound.
The *SuitedForEnergyConfiguration* property is used to search for regions that are suited for energy tuning, i.e., regions whose execution time is higher than some predefined minimum time. This minimum time is determined by physical limitations of a measurement device, i.e., time at which energy consumption is updated, or the time required to apply changes of frequency and/or voltage. Currently, energy measurements on SuperMUC are done with the machine-specific *enopt* library. The library was updated since the first year of the project and now provides access to measurements based on RAPL counters [1] available in Intel Sandy Bridge CPUs and the node’s paddle cards.

The *EnergyInefficient* property for a region is defined as a number of floating point operations per joule. Finally, the *MemoryBound* property is used to search for regions that are suited to be executed with lower frequency due to a large number of long latency memory accesses.

To meet its goal, the strategy uses region information stored in the standard intermediate program representation (SIR) file. From the region information and its codified expert knowledge, the analysis strategy creates a set of hypotheses. These hypotheses are evaluated for outer regions and refined for inner regions until the specified minimum time is met. To continue the search to regions on deeper nesting level is not necessary since the granularity cannot grow. Regions that have a larger execution time per invocation than the minimum execution time can be analyzed and affected by tuning actions. The found properties are reported back to the tuning plugin.

```plaintext
1: Create_next_candidate_properties_set(){
2:    ...
3:    For (p in foundProperties){
4:       If (p is SuitedForEnergyConfiguration){
5:          Create EnergyInefficient and MemoryBound properties for p.region
6:          For (reg nested in p.region){
7:              Create SuitedForEnergyConfiguration properties for reg
8:          }
9:         If (p.region is subroutine call){
10:            Create SuitedForEnergyConfiguration prop for called subroutine
11:        }
12:    }
13: }
14: }
```

Listing 3: Pseudo code for the refinement step in the energy analysis strategy.

The pseudo code snippet above explains the refinement of properties in the energy analysis strategy. The strategy executes multiple search steps. In each search step the next candidate properties set is created. In line 3 it loops over the properties found in the previous step. If a *SuitedForEnergyConfiguration* property was found, the region (*p.region*) is suited for energy tuning via DVFS due to its granularity. The routine first creates the more detailed properties for the same region, i.e., the *EnergyInefficient* and *MemoryBound* properties (line 5). The loop in line 6 iterates over the textually nested regions (*reg*) and creates a new *SuitedForEnergyConfiguration* property for each region in line 7. If *p.region* is a subroutine call, then there are no textually nested regions but the called region can be inspected. Therefore, the routine creates a new *SuitedForEnergyConfiguration* property in line 10 for the called subroutine. Logically, the code takes care
of the fact that multiple calls can exist and the new *SuitedForEnergyConfiguration* property was already checked for the subroutine in a previous refinement step for another call site.

The minimum execution time in the *SuitedForEnergyConfiguration* property is constrained by the physical limitations [2] (see table 1) of the device that measures energy consumption (table 1) or the device that applies tuning actions, i.e. changes the CPU frequency. Changes of the frequency are affected by latency time needed to apply the frequency, i.e., changing voltage and frequency operating point (P-state) in a CPU. SuperMUCs thin islands, Sandy Bridge CPUs, have a maximum latency time of 10 ms. Therefore, it is not possible to tune regions with an execution time that is not significantly higher than 10 ms.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Technology</th>
<th>Sampling time</th>
<th>Domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAPL</td>
<td>MSR</td>
<td>1 ms</td>
<td>Socket, DRAM</td>
</tr>
<tr>
<td>Paddle Card</td>
<td>Ibmaem</td>
<td>333.3 ms</td>
<td>Node</td>
</tr>
<tr>
<td>PDU</td>
<td>Power meter</td>
<td>6 s</td>
<td>Rack</td>
</tr>
</tbody>
</table>

Table 1: Comparison of physical limitations of devices and technologies used to measure energy consumption of applications on SuperMUC.

More information on the *enopt* library can be found in deliverable D4.2. Currently, only measurements based on RAPL counters and paddle cards are available through the library. For practical reasons the PDU measurements are not of interest.

## 5.1 SuitedForEnergyConfiguration property

The reported severity of the property is defined as execution time measured for each invocation of the region. The monitoring library reports execution time as accumulated time during execution. Calculating execution time per call requires that the execution time is divided by the number of calls of the region during the accumulation period. The property condition is set to 1 if and only if it is higher than the threshold defined by physical limitations of the measurement device, e.g. resolution of RAPL counters. The confidence level is always set to 1.

- \( \text{Condition} = 1 \text{ iff } t_{\text{ExecPerCall}} > t_{\text{DeviceResolution}} \)
- \( \text{Confidence} = 1 \)
- \( \text{Severity} = t_{\text{ExecPerCall}} = \frac{t_{\text{exec}}}{N_{\text{calls}}} \)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_{\text{ExecPerCall}} )</td>
<td>Average time to execute a region once</td>
</tr>
<tr>
<td>( t_{\text{DeviceResolution}} )</td>
<td>Minimum time to get one sample from a device energy counters or to apply changes in frequency or voltage</td>
</tr>
<tr>
<td>( t_{\text{exec}} )</td>
<td>Accumulated time to execute a region</td>
</tr>
<tr>
<td>( N_{\text{calls}} )</td>
<td>Number of times a region was executed</td>
</tr>
</tbody>
</table>

## 5.2 EnergyInefficient property

The reported severity of the property is defined as a number of floating point operations per joule. The energy efficiency property is valid only for regions that take significantly longer than the minimum
resolution of the measurement device. The condition is set to 1 if and only if it is lower than one quarter of the maximum theoretical rate for a region. The confidence level is always set to 1.

- \( \text{Condition} = 1 \text{ iff } N_{\text{FLOP}}/J < \frac{1}{4} N_{\text{FLOP}/J_{\text{max}}} \wedge t_{\text{ExecPerCall}} > t_{\text{RAPLResolution}} \)
- \( \text{Confidence} = 1 \)
- \( \text{Severity} = N_{\text{FLOP}}/J = \frac{N_{\text{FLOPS}}}{E_{\text{total}}} \)

### Symbol Meaning

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N_{\text{FLOP}}/J )</td>
<td>Number of floating point operations executed per joule for a region</td>
</tr>
<tr>
<td>( N_{\text{FLOP}/J_{\text{max}}} )</td>
<td>Maximum theoretical number of floating point operations executed per joule</td>
</tr>
<tr>
<td>( N_{\text{FLOPS}} )</td>
<td>Total number of floating point operations for a region</td>
</tr>
<tr>
<td>( E_{\text{total}} )</td>
<td>Total amount of energy dissipated for a region</td>
</tr>
</tbody>
</table>

#### 5.3 MemoryBound property

The reported severity of the property is defined as the number of last level cache misses per floating point operation. For memory bound regions the CPU frequency can be reduced because the cores spend most of the time waiting for data from main memory. Consequently, the execution time should not be considerably affected. The memory bound property is valid only for regions with an execution time that is much higher than the time needed to modify the P-State of a CPU. The property condition is set to 1 if and only if the execution time for a region is much higher than the time needed to modify the P-State of a CPU and the ratio between the L3CacheMisses and the floating point operations is bigger than a threshold. The confidence level is always set to 1.

- \( \text{Condition} = 1 \text{ iff } t_{\text{ExecPerCall}} > 10 t_{\text{P-StateChange}} \) and severity > threshold
- \( \text{Confidence} = 1 \)
- \( \text{Severity} = N_{\text{L3CacheMisses/FL}} = \frac{N_{\text{L3CacheMisses}}}{N_{\text{FLOPS}}} \)

### Symbol Meaning

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_{\text{P-StateChange}} )</td>
<td>Time needed to modify/apply a P-State on a CPU</td>
</tr>
<tr>
<td>( N_{\text{L3CacheMisses/FL}} )</td>
<td>Number of L3 cache misses per floating point operations</td>
</tr>
<tr>
<td>( N_{\text{L3CacheMisses}} )</td>
<td>Total number of L3 cache misses occurred for a region</td>
</tr>
</tbody>
</table>

#### 6 Extensions to the SIR for the High Level Pattern Tuning Plugin

The SIR file is used by the plugin for high-level parallel patterns to communicate the information about the location of the pipeline regions in the code and their related tuning points for stage and buffer tuning. The corresponding SIR file is based on the high-level code representation and it is generated during the code transformation phase. In the current plugin prototype, the SIR file is extended to support the following extensions specific to pipeline patterns:

- The \textit{codeRegion} element now supports \textit{pipeline} type.
- The \textit{selector} element, inside the \textit{plugin} element, is used to describe the tuning points.
Listing 4 shows relevant parts of the SIR file.

```xml
...<codeRegion type="pipeline" name="" id="...">...
<position startLine="..." endLine="...">...
<file name="..."/>
<position>
<plugin pluginId="Pipeline">
<selector tuningActionType="VAR">
  tuningActionName="VTTp1s2REPLICATION"
  min="2"
  max="10"
  step="2"/>
...</plugin>
...</codeRegion>...
```

Listing 4: Extensions of the SIR file. Only elements specific to pipeline tuning plugin are shown.

As observable in Listing 6, a tunable program comprises a while loop indicated with a `pipeline` pragma code-annotation. Such regions (annotated while) further comprise pipeline stages (component invocations), which also can be annotated with pragmas. These annotations can be used to set parameters such as stage replication factors or stage-buffer sizes. In order to enable the automatic tuning with a restricted search space for such parameters, the pragma annotations allow the definition of a value range for the end-user. This information (relevant code region, tuning parameters and value ranges) is stored in the SIR file for further processing by the PTF and the pipeline tuning plugin.

For example, Listing 6 shows a stage annotation within the pipeline region with a stage `replicate` keyword and a tuning range of (2:10:2). This means that the stage replication for that particular stage can be set to 2, 4, 6, 8 or 10. The according representation in the SIR file can be seen in the “selector” node in Listing 4. This XML node stores the variable name (tuningActionName), the min integer value (2), the max integer value (10) and the step size (2).

In order to support ranges for the tuning parameters, we implemented the following extensions to the `SelectorType` in the XML schema:

- `min` – smallest suggested value of the tuning parameter. Integer type (xs:integer) values
- `max` – largest suggested value of the tuning parameter. Integer type (xs:integer) values
- `step` – suggested step size. Integer type (xs:integer) values
Listing 5 shows the relevant sections from the XML schema:

```xml
...  
<xs:complexType name="SelectorType">
  <xs:sequence>
    <xs:element name="codeVariant" type="psc:CodeVariantType"  
      maxOccurs="unbounded" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="tuningActionName" type="xs:string"/>
  <xs:attribute name="numberOfVariants" type="xs:positiveInteger"/>
  <xs:attribute name="min" type="xs:integer"/>
  <xs:attribute name="max" type="xs:integer"/>
  <xs:attribute name="step" type="xs:integer"/>
</xs:complexType>
```

Listing 5: Partial XML code outlines the SelectorType extensions.

The semantics of ranges for tuning parameters is in accordance with the triplet notation known from Fortran, i.e., it represents the sequence of values given by \{min, min+step, min+2*step, ..., min+k*step\} where k is the smallest number such that min+(k+1)*step > max (if step >= 0) or min+(k+1)*step < max (if step <= 0).

This approach follows the extensions for user-provided ranges for the tuning parameters in the high-level coordination language. In this context the range is defined by its minimum, maximum and the step size. An example of such user-provided ranges is shown in Listing 6, which specifies the following values that should be explored by PTF for the replication factor \{2, 4, 6, 8, 10\}.

```c
...  
#pragma pph pipeline  
...  
#pragma pph pipeline  
while (...) {  
  ...  
#pragma pph stage replicate (2:10:2)  
  func(...);  
  ...  
}  
...  
```

Listing 6: Example of high-level code with pipeline-specific annotations.

## 7 Summary and future work

In the second year of the AutoTune project, the set of analysis strategies was extended to provide analysis capabilities required by the tuning plugins. This work uses the monitoring capabilities implemented in the first year of the project. These analysis strategies are used to provide support to the HMPP Codelet Tuning
plugin and energy efficiency via CPU frequency plugin. The analysis strategies support tuning plugins to reduce the search space by means of removing complete regions and/or limiting the size of a tuning parameter for a region. They therefore speed up the tuning process. In addition, the SIR representation was extended for the high-level pattern tuning plugin and the HMPP Codelet Tuning plugin. Its generation is now done by the CAPS compiler and the pattern transformation system.

In the third year, this workpackage will develop an advanced MPI performance analysis strategy to support MPI tuning plugin. For this task a set of new MPI performance properties will be specified. These properties will guide the selection of the tuning parameters, such as communication buffer size or the eager protocol threshold.

8 Bibliography

[2] IBM Systems and Technology Group, A Performance Guide For HPC Applications On the IBM® System x® iDataplex® release 1.0.2 , Chapter 9