PTF Parallel Patterns Plugin User's Guide

PTF Version: 1.1

Parallel Patterns Plugin Version: 1.1

Research Group Scientific Computing, University of Vienna

13.04.2015

Contents

Chapter 1

Introduction

The parallel pattern plugin addresses automatic performance tuning of high-level pipeline patterns for accelerated parallel systems. The plugin builds on a component-based task-parallel programming framework that has been developed within the European project PEPPHER[1], which addressed programmability and performance portability for single-node heterogeneous manycore systems.

Within the PEPPHER framework tasks correspond to multi-architectural components that encapsulate different implementation variants of performance-critical application functionality. Such component implementation variants may be optimized for different execution units of a heterogeneous target architecture (e.g., CPU core, GPU, Xeon PHI or similar). High-level coordination primitives as well as patterns are provided to construct applications from such components. A sophisticated runtime system is utilized to select and dynamically schedule component implementation variants for efficient parallel execution on heterogeneous many-core architectures.

The parallel pipeline patterns, as supported by the PEPPHER framework, are based on while-loops with source- code annotations (Listing 1.1). Pipeline stages usually correspond to calls to multi-architectural components, for which multiple implementation variants may be provided. Such component implementation variants may be optimized for different execution units of a heterogeneous target architecture (e.g.: systems equipped with GPUs).

[1] S. Benkner, S. Pllana, J. L. Träff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney and V. Osipov, "PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems", IEEE Micro, vol. 31, no. 5, p. 2841, 2011.

```
...
#pragma pph pipeline with buffer (UNORDERED, N*2)
while ( inputstream >> file ) {
  ReadImage(file, image);
  ResizeAndColorConvert (image , outimage);
  #pragma pph stage replication(rfactor)
  DetectFace(outimage);
  #pragma pph stage with buffer (PRIORITY, N*2)
  WriteFaceDetectedImage(file, outimage);
}
...
```

Listing 1.1: "Example of a high-level pipeline pattern code for image processing. Pipeline consists of four stages. For the compute intensive DetectFace stage different component implementation variants for GPU and CPU exist and the PTF determines the best replication factor such that all execution units of the target architecture are exploited and execution time is minimized."

Such high-level pipeline code is transformed by a source-to-source compiler into a C++ code that utilizes a coordination layer for managing parallel execution of pipelines. The coordination layer (also referred to as the pipeline coordination layer or VPattern Library) manages all aspects of pipelined execution on a heterogeneous many-core architecture, including the automatic management of buffers for data passed between pipeline stages, the replication of individual stages, and the coordination of task-parallel execution of pipeline stages. Internally, the pipeline coordination library utilizes the StarPU[2] heterogeneous runtime system, which is responsible for dynamically selecting suitable component implementation variants for pipeline stages and for scheduling their execution to the different execution units of a heterogeneous many-core system in a performance- and resource-efficient way. StarPU also manages data transfers between execution units, ensures memory coherency, and provides support for different scheduling strategies, with the goal of utilizing all execution units of the target architecture.

The pipeline coordination layer exposes a set of tuning parameters, thus allowing external tuners like Periscope Tuning Framework to automatically tune the performance of applications using this pattern.

The plugin supports two modes of execution. In the first mode, overall pipeline execution time of the application using different sets of tuning parameters is being measured and tracked. The search through all possible configurations is performed exhaustively. In the second mode, the focused

[2]C. Augonnet, S. Thibault, R. Namyst and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," Concurrency and Computation: Practice and Experience, vol. 23, no. Special Issue: EuroPar, pp. 187-198, 2009.

tuning of the pipelines is enabled. In this mode, the plugin additionally uses pipeline-specific performance analysis to detect performance-limiting stages in the pre-analysis phase and constrains the search space accordingly. The plugin produces a tuning advice, pointing out the configuration with the shortest execution time, as well as the list of all executed scenarios.

Chapter 2

Quick Start

Tuning plugin for high-level pipeline patterns is being installed along with the Periscope Tuning Framework. Please refer to the *PTF Installation Guide* for a complete description of the installation process.

2.1 Quick installation

In order to use the plugin, PTF should be configured with –enable-vpattern and following software packages need to be available on the system:

- VPattern Library >= 0.9

2.2 Running Pipeline Pattern Plugin

The plugin for high-level pipeline patterns runs as a plugin within the Periscope Tuning Framework. It can be started using `psc_frontend` (see also *PTF User's Guide*) by setting the `tune` flag to `pipeline`.

```
--tune=pipeline
```

For an arbitrary example, one would call from within the folder containing the executable:

```
psc_frontend --apprun=./<executable_filename> --sir=./<sir_filename>
--mpinumprocs=1 --tune=pipeline --force-localhost
```

This will start the measurements and the plugin tuning strategy for the specified application using one process.

Additionally, the `--vpattern-focused` command-line switch may be used to enable focused tuning that tries to detect performance-limiting pipeline stages and to constrain the search space accordingly.

2.3 Execution results

Upon successful completion of the tuning, the pipeline plugin displays at the standard output the list of all (*scenarios*) that were used in the search along with the corresponding execution times (*severity*). It also outputs the scenario with the best execution time. Additionally, the plugin outputs all tuning parameters and corresponding values for each executed scenario.

This is an example output of the above call to `psc_frontend`:

```
AutoTune Results:
------------------------
Search Steps:  6
Found Optimum Scenario:  4
  STAGEREPLICATIONFACTOR_1_55:  3
  ...
  NCPUS: 2
  NGPUS: 1
  SCHEDULING_POLICY: 1

All Results:
Scenario | Severity
0 | 10.545700
1 | 10.165600
2 | 8.308380
3 | 6.061190
4 | 5.958680
5 | 10.591900
------------------------
```

The optimum scenario is the tuning advice of the plugin and it consists of the concrete values for tuning parameters that provided the best execution time for executed application.

Chapter 3

Autotuning Approach

The tuning plugin for high-level pipeline patterns follows the general PTF plugin approach (see also *PTF User's Guide*).

3.1 Tuning parameters

The plugin supports three types of tuning parameters that address tuning of pipeline-, runtime-, and machine-specific aspects of the pipeline execution.

The pipeline-specific tuning parameters target pipeline structural tuning parameters such as stage replication factors, and sizes of input and output buffers. Each stage in the pipeline may have two or more such parameters. Therefore, the total number of tuning parameters depends on the structure of the pipeline.

Machine-specific parameters NCPUS and NGPUS are used to describe available hardware resources such as the number of available CPU cores (or available hardware threads) and number of usable graphic cards.

Finally, the runtime-specific parameters address runtime specific parameters such as the scheduling policy used by the runtime system.

The list of all possible tuning parameters is given in the following listing:

- *stage replication factor* - the number of stage instances that may be executed in parallel
- *sizes of buffers* - size of I/O buffers that hold data packets passed between pipeline stages
- *number of CPU cores* - also the number available CPU hardware threads to be used for execution

- *number of GPUs* - number of available graphic cards to be used for execution
- *scheduling strategy* - a scheduling policy used by StarPU runtime for scheduling component calls to available execution units of the target system

Each pipeline-specific parameter may be provided explicitly by the user in the original code or directly by altering the generated SIR file. The pipeline-specific tuning parameters may be expressed in the form of concrete values or tuning ranges. Listing 3.2 demonstrates user-provided range for the stage replication factor with *min=2*, *max=10* and *step=2*, and output buffer size set to *32*. In the high-level code, this is expressed in the following way:

```
#pragma pph stage replicate (2:16:2) buffer (32, ...)
func1(...)
```

Listing 3.1: "User-provided ranges in PEPPHER high-level codes. The stage replication is hinted to have values between 2 and 10 incremented by 2. The buffer size is set to a concrete value of 32."

In the SIR file, end-users may alter corresponding *min*, *max* and *step* values for each tuning parameter:

```
...
<plugin pluginId="Pipeline">
  ...
  <selector tuningActionType="VAR"
            tuningActionName="STAGEREPLICATIONFACTOR_1_55"
            min="1" max="8" step="1"/>
  ...
  <selector tuningActionType="VAR"
            tuningActionName="NCPUS"
            min="1" max="8" step="1"/>
  ...
</plugin>
...
```

Listing 3.2: "In order to influence the tuning process the SIR file may be altered manually by changing the corresponding *min* *max* and *step* values for each tuning parameter."

In order to construct a variant space, the plugin processes the SIR file during the initialization phase and collects information about tuning parameters and pipeline regions. All tuning parameters together define the tuning space.

3.2 Search strategy

In order to find the best set of tuning parameters for an application, a search through the tuning space has to be performed.

In the default mode, the plugin utilizes the exhaustive search strategy and evaluates all possible tuning scenarios in the tuning space. The total number of scenarios for each pipeline is a cross product of all tuning parameters.

In the focused tuning mode, the plugin uses pre-analysis to detect performance-limiting stages, and constrain the search space. The total number of scenarios will be equal to the cross product of the number of values used for the limiter stage replication factors tuning parameter, limiter stage buffer size tuning parameter, runtime scheduling policy tuning parameter, hardware description tuning parameters (NCPUS and NGPUS) and user-provided tuning parameters (if any).

3.3 Tuning scenario

Based on the chosen strategy, consecutive *tuning scenarios* are then being generated at run time and the performance of the application is being evaluated for each of these scenarios.

In the plugin for high-level pipeline patterns one scenario represents one specific combination containing concrete values of all tuning points.

3.4 Tuning action

Applying one specific scenario to the application implies passing concrete values of tuning parameters to the pipeline coordination layer.

Chapter 4

How To Use the Tuning Advice

Upon successful completion, the plugin outputs a list of all tested scenarios as well as the `id` of the best scenario and its configuration.

Given the plugin output:

```
AutoTune Results:
-------------------------
Search Steps:  6
Optimum Scenario:  4
  STAGEREPLICATIONFACTOR_1_55:  3
  NCPUS: 2
  NGPUS: 1
  SCHEDULING_POLICY: 1
```

"Optimum scenario" is the tuning advice of the plugin and it consists of the concrete values for tuning parameters that provided the best execution time for executed application. These values should be inserted into the original application.