Automatic Performance Tuning of Pipeline Patterns for Heterogeneous Parallel Architectures

E. Bajrovic[1] **and S. Benkner**[1]
[1]Research Group Scientific Computing, Faculty of Computer Science, University of Vienna, Vienna, Austria

Abstract— *Heterogeneous parallel architectures combining conventional multicore CPUs with GPUs and other types of accelerators promise significant performance gains compared to homogeneous systems. However, exploiting the full potential of such systems is becoming more and more challenging often forcing programmers to combine different programming models and parallelization strategies. A promising approach to coping with the increased programming complexity is the use of parallel patterns for expressing certain types of computations at a high-level of abstraction while relying on the compiler and runtime system to map such patterns onto a heterogeneous system. In this paper we present an approach for automatic performance tuning of high-level pipeline patterns for heterogeneous parallel systems in the context of a task-parallel component-based programming model. Our automatic performance tuning approach attempts to automatically determine the best combination of pattern-specific parameters, parameters exposed by the runtime system, and machine-specific parameters such that execution is optimized for a given workload and target architecture. Experimental results on two state-of-the-art heterogeneous systems demonstrate the effectiveness of our approach.*

Keywords: parallel programming, patterns, autotuning, heterogeneous manycore architectures

1. Introduction

Over the last decade we have seen dramatic changes in the architecture of parallel systems due to the introduction of multicore processors and the shift to heterogeneous parallel systems that comprise different types of execution units specialized for efficiently processing different types of computational workloads. Typically, heterogeneous parallel architectures combine conventional multicore CPUs with GPUs and other types of accelerators. Such systems have become increasingly important since they promise significant performance gains compared to homogeneous systems. However, exploiting the full potential of such systems often requires combining different programming models and parallelization strategies, which significantly increases the complexity of application development.

A promising approach for coping with the complexity of programming heterogeneous parallel architectures is the use of parallel patterns or skeletons [1], [2], [3], [4], for expressing certain types of computations at a high-level of abstraction while relying on the compiler and runtime system to map such patterns onto a heterogeneous system. However, mapping high-level parallel patterns efficiently to different types of heterogeneous target architectures often requires fine-tuning of various parameters at the application level (e.g. replication factors of pipeline stages) or runtime level (e.g. the scheduling strategy), which usually is a time-consuming tasks and requires detailed knowledge of the involved compiler(s) and runtime system(s) as well as of the target architecture. As a consequence, automatic performance tuning techniques (also referred to as *autotuning*) to automatically search for the best combination of such parameters have become of growing interest.

In this paper we present an approach for automatic performance tuning of high-level pipeline patterns for accelerated parallel systems. Our work builds on a component-based task-parallel programming framework that has been developed in the context of the European PEPPHER project [5], which addressed programmability and performance portability for single-node heterogeneous manycore systems. Within the PEPPHER framework, pipeline patterns are realized based on while-loops with source-code annotations [6]. 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., for a homogeneous multicore CPU, for a GPU, or for other types of accelerators. At runtime, for each call to a component a task is generated, yielding a dynamic task graph. It is then up to the runtime system to schedule the task graph for efficient parallel execution on the different execution units of a heterogeneous target system. The runtime system chooses for each task a suitable component implementation variant and dynamically scheduling its execution onto a free execution unit of the target architecture such that all available execution units are utilized and overall performance is optimized.

Within the European Autotune project [11] we have integrated the PEPPHER high-level programming framework with the Periscope Tuning Framework [11] for online performance analysis and tuning. Our automatic performance tuning approach takes into account pattern-specific parameters, parameters exposed by the runtime system, as well as machine-specific characteristics in order to optimize the execution of applications with pipeline patterns on heterogeneous parallel systems equipped with CPUs and GPUs.

The remainder of this paper is organized as follows: In Section 2 we provide an overview of the PEPPHER framework and describe the support for high-level pipeline patterns. In Section 3 we describe our pipeline coordination layer which manages the execution of pipeline patterns at runtime and which exhibits different parameters amenable to autotuning. Section 4 provides an overview of the Persicope tuning framework and outlines the tuning of pipeline patterns. In Section 5 we present experimental results using a real-world face detection application. Section 6 discusses related work followed by a conclusion in Section 7.

2. High-Level Programming Framework

The European research project PEPPHER [5] developed a methodology for improving programmability and performance portability for single-node heterogeneous many-core systems. The PEPPHER methodology is characterized by a component-based programming approach in combination with an asynchronous task-parallel execution model.

2.1 Multiarchitectural Components

The central idea is to provide performance-critical parts (typically functions) of applications as components with multiple implementation variants, called multi-architectural components. Each such variant is tailored for a different type of target architecture (CPU, GPU, accelerator) that may be utilized within a heterogeneous many-core system. Component implementation variants may be sequential or parallel and may be implemented with different programming APIs including C/C++, OpenMP, CUDA and OpenCL. All implementation variants of a specific component must adhere to the same component interface. Components and implementation variants are accompanied with meta-data, supplied via external XML descriptors. Such descriptors specify the data read and/or written by a component and provide information about the target platform(s) [7] and about specific resource requirements or constraints.

For constructing applications from components a set of coordination primitives has been developed. Programmers may construct applications at a high level of abstraction by invoking component functionality from C/C++ codes via their interfaces and by using source code annotations (pragmas) to delineate asynchronous (or synchronous) component calls. With this approach, a sequential program spawns component calls, which are then scheduled for task-parallel execution by the runtime system. A source-to-source compiler transforms annotated component calls such that they are registered with the runtime system and generates corresponding glue-code.

2.2 Pipeline Patterns

In addition to the basic coordination primitives for designating asynchronous (or synchronous) component calls we have developed high-level language support for pipeline patterns. Pipeline patterns are expressed using annotated while loops where the loop body comprises calls to multi-architectural components.

An example of a high-level C++ pipeline code for face detection in a stream of images is shown in Figure 1. The first pipeline stage reads images from an input file, the middle stages perform image transformation and face detection, and the last stage outputs the result images where all detected faces are marked with rectangles. For the `detectFace` stage, two different component implementation variants are provided within the PEPPHER framework, one optimized for execution on a conventional CPU core and one optimized for GPUs. These implementation variants have been reengineered from the OpenCV image processing library [10]. By means of annotations, the user can specify what kind and size of buffers should be generated for passing data between pipeline stages. Moreover, the user can specify a replication factor for individual pipeline stages in order to influence the degree of parallelism during execution. By changing the replication factors and buffer sizes the user can quickly experiment with different configurations of the pipeline. However, the goal of our autotuning approach is to automatically determine the best values for these tuning parameters such that overall execution time is minimized.

```
N = get_max_execution_units ();

#pragma pph pipeline with buffer ( PRIORITY , N*2 )
while ( inputstream >> file ) {
        readImage ( file , image );
       #pragma pph stage replicate (N)
        {
            resizeAndColorConvert ( image );
            detectFace ( image , outimage );
        }
        writeFaceDetectedImage ( file , outimage );
}
```

Fig. 1: A pipeline pattern for face detection in a stream of images.

2.3 Transformation System

A source-to-source compiler transforms pipeline patterns into a C++ code that utilizes a coordination layer for managing parallel execution on heterogeneous many-core architectures at runtime. Pipeline constructs are analyzed in order to determine the structure of the pipeline (stage interconnection) by analyzing the data types of objects passed between pipeline stages. For each stage interconnection corresponding buffer data structures are generated.

The generated target code contains calls to the pipeline coordination layer which comprises various classes for coordinating the execution of pipeline stages on top of the StarPU runtime system [8]. At run-time, component invocations result in tasks that are managed by the StarPU runtime system and executed non-preemptively.

3. Pipeline Coordination Layer

The pipeline coordination layer manages all aspects of 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 layer utilizes the StarPU [8] heterogeneous runtime system.

StarPU 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 and provides support for different scheduling strategies, with the goal of utilizing all execution units of the target architecture. Data transfers for tasks are determined based on the XML meta-data provided in component descriptors, and their costs are taken into account during scheduling by StarPU. In the following we outline the major aspects of the pipeline coordination layer and its interaction with the underlying StarPU runtime system.

The pipeline coordination layer provides several classes for coordinating the execution of pipeline stages. The *PipelineManager* class is used to control multiple pipeline patterns within an application. The *Pipeline* class provides methods for starting, pausing and resuming the execution of a pipeline pattern, and for dynamically reconfiguring the tuning parameters of a pipeline (i.e., changing the replication factor and buffer sizes). The *Stage* class encapsulates information on the stage functionality (i.e., the component that is invoked), connected buffers, predecessor and successor stages, and the stage replication factor. The *Stage* class provides two methods for coordinating the execution of a pipelined application: the method *execute_async()* for posting a stage for execution to the runtime system and the method *callback()* for transferring control back to a stage object after its associated component has finished execution. Every stage instance is executed in an asynchronous fashion.

Figure 2 illustrates how a pipeline pattern is being executed on top of the StarPU runtime system. The *PipelineManager* creates a pipeline object and the corresponding stage and buffer objects. It then starts the pipeline by invoking the *run_pipeline()* method of the pipeline object. The pipeline object then calls the *execute_async()* method for each stage object, which initiates the execution of stages, each within its own thread. Each instance of a stage execution pops input data from the corresponding stage input buffer, and then delegates the actual execution of the stage to StarPU by calling the *post()* method. From this point on StarPU is responsible for executing the stage instance by selecting an appropriate stage implementation variant and scheduling its execution on a suitable execution unit (CPU or GPU). When the stage instance has finished execution, StarPU calls the method *callback()* to pass control back to the stage object which then initiates execution of the next stage instance.

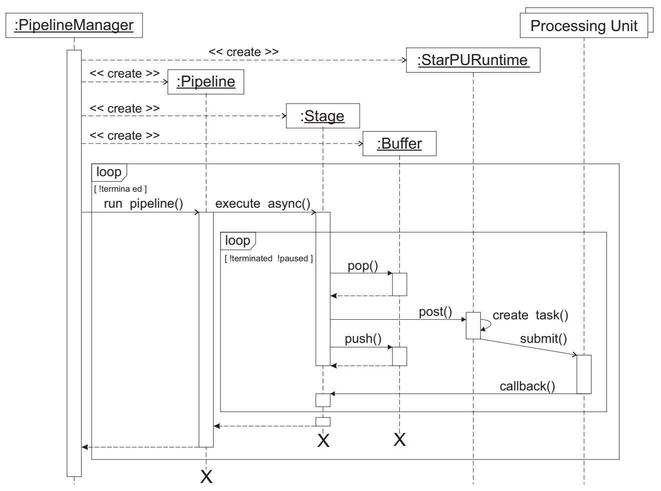


Fig. 2: Pipeline Execution Model.

StarPU relies on a representation of the program as a directed acyclic graph (DAG) where nodes represent component calls (tasks) and edges represent data dependences. The runtime system dynamically schedules component calls to the available execution units of a heterogeneous many-core architecture such that (1) independent component calls execute in parallel on different execution units and (2) the "best" implementation variants for a given architecture are selected based on historical performance information captured in performance models. StarPU also manages data transfers between CPUs and GPUs, ensures memory coherency, and provides support for different scheduling strategies. Besides the well-known EAGER scheduling policy, StarPU also features the Heterogeneous Earliest Finish Time (HEFT) [9] policy. The HEFT policy considers inter-component data dependencies and schedules components to workers taking into account the current system load, available component implementation variants, and historical execution profiles, with the goal of minimizing overall execution time by favoring implementations variants with the lowest expected execution time.

3.1 Tuning Parameters

The pipeline coordination layer enables dynamic reconfiguration by exposing a set of tuning parameters, thus allowing users or external tools to tune the execution of the pipeline in order to achieve a desired goal (e.g., to maximize pipeline throughput). The following tuning parameters are provided: (1) the stage replication factor, which determines the number of stage instances that may be executed in parallel, (2) the sizes of buffers to hold data packets passed between pipeline stages, (3) the number of CPU cores to be used, (4) the number of GPUs to be used, and (5) the scheduling strategy used by StarPU for scheduling component calls to free execution units of the target system.

All these parameters have a profound influence on the performance of applications that rely on pipeline patterns. Finding the best parameter combination for a given application, problem size, and machine configuration is an elaborate and time-consuming task for users and thus should be automated as far as possible.

3.2 Performance Metrics

In order to support automatic performance tuning the coordination layer provides integrated support for measuring the following performance metrics of pipeline patterns:

- Stage execution time - the execution time of an individual instance of a pipeline stage.
- Buffer input processing time - the time to process the input objects of one buffer.
- Buffer output processing time - the time to process the output objects of one buffer.
- Buffer size - the size of individual buffers.
- Pipeline execution time - the overall execution time of one pipeline pattern.

4. The PTF Tuning Framework

The Periscope Tuning Framework (PTF) [11] is an extension of the Periscope online performance analysis tool [12]. PTF aims at providing an integrated infrastructure for performance analysis and automatic performance tuning that can incorporate expert knowledge to guide the search for performance problems and tuned code versions.

PTF facilitates the development of tuning plugins that include codified expert knowledge about the performance characteristics and computational patterns of the target applications and the specific tuning problem. Besides the pipeline tuning plugin presented in this paper, several other tuning plugins (e.g. for tuning of MPI parameters, for master/-worker patterns, and for energy tuning via dynamic voltage and frequency scaing) have been developed in the context of the AutoTune project [11].

Based on performance analysis, tuning plugins identify tuning alternatives, so-called tuning scenarios (i.e. different configurations of tuning parameters), and then proceed to evaluate them. The evaluation of tuning scenarios may be performed online, i.e. during a single application run, which reduces the time required to find the best tuning scenario dramatically.

Automatic performance analysis is based on formalized performance properties, e.g., load imbalance or slow pipeline stages (limiter stages). One or more analysis agents may be used to search for performance properties in the program execution under investigation. Analysis agents communicate with the monitor via the monitor request interface (MRI) linked with the application process(es) to be tuned. The MRI monitor performs the measurements of performance data requested by the analysis agent and transfers the measured performance data to the PTF.

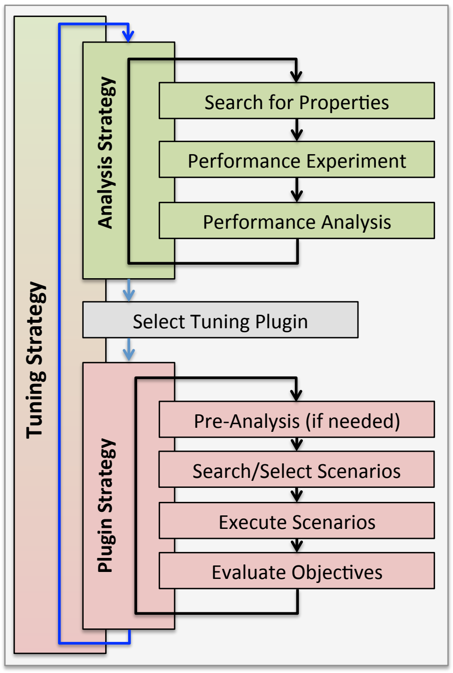


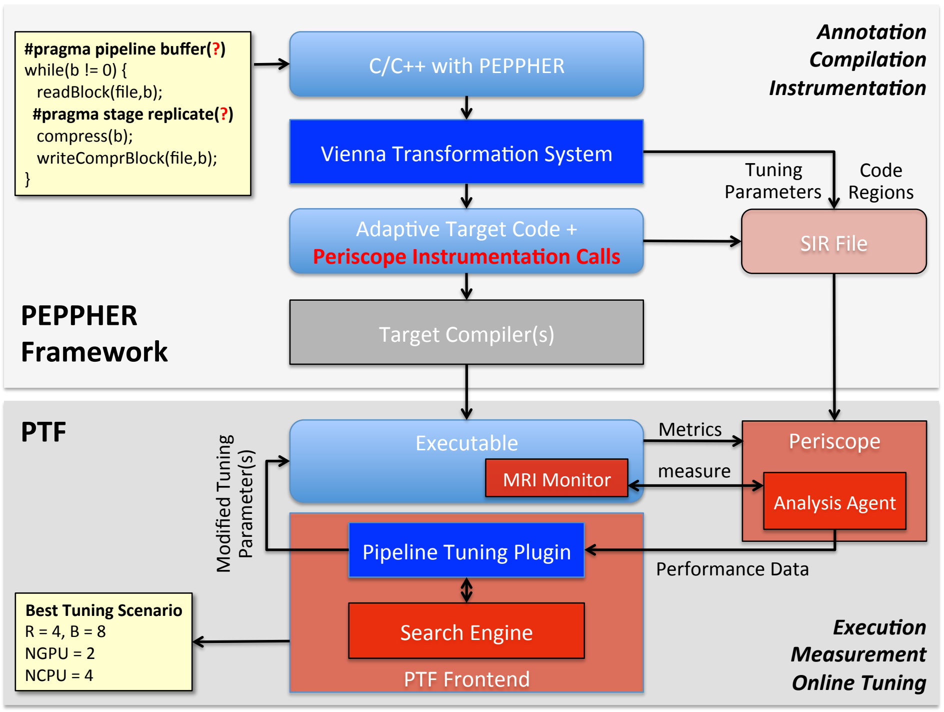
Fig. 3: The PTF Tuning Model

4.1 The PTF Tuning Model

Figure 3 illustrates the PTF tuning model. As shown in the figure, a tuning strategy is comprised of an analysis strategy and a plugin strategy, which may be performed iteratively, depending on the concrete nature of the tuning problem. The analysis strategy guides performance analysis and the search for performance properties, while the plugin strategy guides the search for optimized tuning scenarios. Once the tuning process is finished, a tuning report will be generated that documents the tuning actions recommended by PTF.

The tuning process is usually proceeded with a preprocessing step of the application source files (not shown in Fig. 3). Preprocessing comprises code instrumentation required for performance analysis and static analysis and is either performed by PTF, which includes and integrated instrumenter for C/C++ and FORTRAN, or by external tools. In the case of pipeline patterns, instrumentation is performed by the PEPPHER source-to-source transformation system. During the instrumentation phase, also a SIR file (Standard Intermediate Representation) is generated, which includes static information about the instrumented code regions to be utilized by PTF for performance analysis and tuning.

The analysis strategy guides the search for certain (predefined) performance properties, by performing one or more performance experiments and analyzing the corresponding performance measurements. In case of a pipeline pattern, the analysis strategy searches for a limiter stage, which takes much more time than other stages. If a limiter stage is found, the pipeline tuning plugin is triggered and attempts to



```
#pragma pipeline buffer(?)
while(b != 0) {
  readBlock(file,b);
 #pragma stage replicate(?)
  compress(b);
  writeComprBlock(file,b);
}
```

```
Best Tuning Scenario
R = 4, B = 8
NGPU = 2
NCPU = 4
```

Fig. 4: Integration of the PEPPHER framework for high-level pipeline patterns with the Periscope Tuning Framework. Blue components are specific to the PEPPHER framework, while red components are specific to PTF.

increase the replication factor of the limiter stage such that overall execution time is reduced. In addition to this specific plugin strategy, we have realized a general plugin-strategy finding a configuration of the five pipeline tuning parameters that minimizes overall execution time.

As shown in Figure 3, a plugin strategy is comprised of an optional pre-analysis phase, a phase for searching, selecting and analyzing tuning scenarios within the set of overall tuning scenarios, and phases for executing tuning scenarios and evaluating the tuning objectives such that the best tuning scenarios is identified. For preparation and creation of new tuning scenarios, plugins can access a search interface, which enables to apply different search strategies for finding promising tuning scenarios. In our current implementation of the pipeline tuning plugin we have used exhaustive search. In the future we plan to integrate alternative search strategies.

4.2 Pipeline Tuning Workflow

In the following we describe the major steps of the pipeline tuning workflow according to Figure 4.

First, the application source code is processed by the PEPPHER transformations system, which translates high-level pipeline patterns into a representation that uses the pipeline coordination layer and inserts monitoring calls for obtaining the pipeline-specific performance metrics. In addition, a SIR file (XML intermediate program representation) with information about the relevant tuning parameters and code regions is created. The generated code is then compiled with target specific compilers and linked with the PTF performance monitoring (MRI) libraries. During program execution, the linked MRI monitor is used for communicating measured performance metrics to the PTF and for (re-)setting the values of tuning parameters. The tuning plugin decides the measurements that will be considered for application tuning, and the modified tuning parameters. The pipeline tuning plugin constructs the set of tuning scenarios, executes them by dynamically reconfiguring the pipeline tuning parameters, and reports the best tuning scenario.

5. Experimental Evaluation

For evaluation we use the OpenCV face detection application outlined previously in Figure 1. The application processes a set of 350 images of nHD (640x360) resolution, each containing an arbitrary number of human faces. For the computationally most demanding component `detectFace()`, which is called in the middle stage, two different implementation variants, one for a single CPU core and one for a GPU, were utilized. These implementation variants have been re-engineered from the OpenCV library, which includes both a sequential C++ version and a CUDA version, but which had to be slightly adapted to the PEPPHER component model.

We present speedup measurements and autotuning results on two different CPU/GPU systems. The first machine is equipped with two quad-core Intel Xeon X5550 CPUs

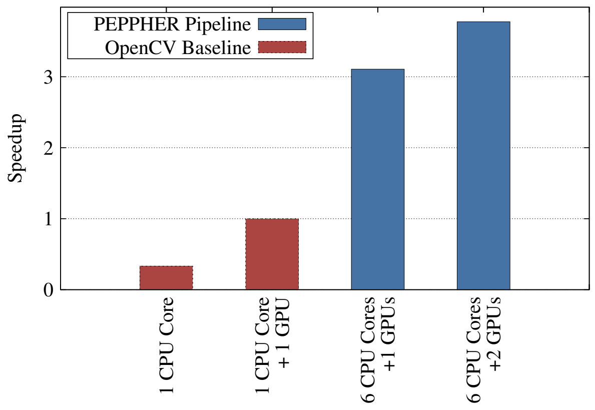


Fig. 5: Speedup results for face detection application on a machine with two quad-core CPUs and two Tesla GPUs relative to the OpenCV baseline version.

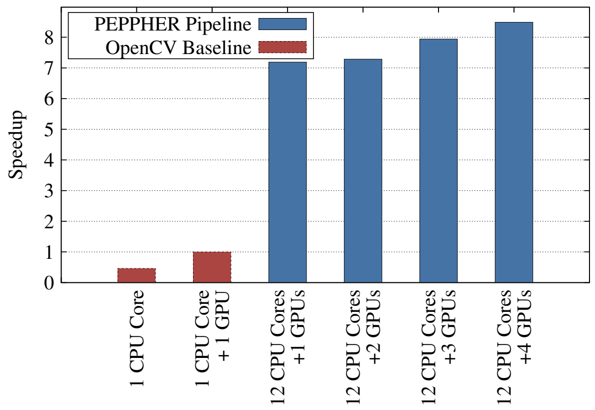


Fig. 6: Speedup results for face detection application on a machine with two octa-core CPUs and four Kepler GPUs relative to the OpenCV baseline version.

(2.66GHz, 24GB RAM) and NVIDIA Tesla C2050 and C1060 GPUs, respectively. The second machine is equipped with two octa-core Intel Xeon E5-2650 CPUs (2.0 GHz, 128GB RAM) and 4 NVIDIA Kepler K20 GPUs. As shown in Figures 5 and 6, we executed the face detection pipeline on different machine configurations and utilized PTF to automatically determine the best combination of tuning parameters such that execution time was minimized.

Figure 5 shows speedup results on the first machine equipped with Tesla GPUs. The second red bar in the figure is the OpenCV baseline version, i.e. using the original OpenCV library which supports using just 1 CPU and 1 GPU. The two blue bars show the results of the autotuned PEPPHER pipeline versions using 6 CPU cores and one or two GPUs, respectively. These results clearly demonstrate that our high-level component-based approach can effectively utilize all execution units of the system. Note also that no source code changes were necessary to run on the two different machine configurations with one and two GPUS, respectively. Using the whole machine a speedup of about 4 has been obtained compared to the OpenCV base version and a speedup of about 12 compared to the single core version.

Using the PTF tuning plugin, we used exhaustive search to find the best configuration for the available tuning parameters. As described in Section 3.1, we considered the following five tuning parameters: (1) stage replication factor of the `detectFace()` stage, (2) input buffer size of the `detectFace()` stage, (3) number of CPU cores, (4) number of GPUs, and (5) the scheduling policy - EAGER (simple greedy scheduler) versus HEFT (Heterogeneous Earliest Finish Time) [8]. In total PTF explored 360 possible configurations, spending about 6 hours in doing so. Finding the best parameter configuration manually would require significantly more time, usually several days of reconfiguration and performance measurement.

In Table 1, we summarize the explored values for each tuning parameter when tuning the face detection application on the first machine. With the best parameter configuration using the whole system (i.e., all CPU cores and all GPUs) the execution time of the face detection application over the complete data set was 8.2 seconds. It used a replication factor of 8, 6 CPU cores, 2 GPUs, buffer size of 32 and HEFT scheduling policy. The slowest configuration that utilized the whole system resulted in an execution time of 19.6 seconds.

Tuning parameter	**Possible values**	**Best configuration**
Replication factor	1, 2, 4, 8	8
Number of CPU cores	1, 2, 4, 6, 8	6
Number of GPUs	0, 1, 2	2
Scheduling policy	"EAGER", "HEFT"	"HEFT"
Buffer size	8, 16, 32	32

Table 1: Possible values of tuning parameters (first machine).

Figure 6 shows speedup results on the second machine equipped with Kepler GPUs. Again, the second red bar in the figure is the OpenCV baseline version. The four blue bars show the results of the PEPPHER pipeline version using one up to four GPUs as well as 12 CPU cores. Again no code changes were required to run the application on these different machine configurations. Using 12 CPU cores and 1 GPU delivers a speedup of about 7 over the OpenCV baseline version that uses one CPU and one GPU. Adding more GPUs results in only modest speedup increases, which can be mainly attributed to the rather low resolution of the images. We expect that for higher resolutions greater speedups with multiple GPUs would be possible due to the increased computational complexity.

Also on the second machine we used to PTF to find the best configuration of tuning parameters by exploring 1470 different combinations. The best parameter configuration using all CPU cores and all GPUs of the second machine resulted in an execution time of 4.6 seconds, with a replication factor of 16, 12 CPU cores, 4 GPUs, buffer size of 32 and HEFT scheduling policy. The slowest configuration that utilized the whole system resulted in an execution time of 15.3 seconds.

6. Related Work

Due to the increasing complexity and diversity of parallel architectures, there is a growing interest in automatic performance tuning techniques. Existing autotuning efforts include self-tuning specialized libraries (e.g., linear algebra or signal processing) like ATLAS[14] or FFTW[15], tools that automatically search for best combination of compiler optimization parameters [16], [17], and tools that search for best values of application-level parameters [18], [19].

Our approach mainly deals with tuning of parameters exhibited by a runtime library (i.e. our coordination layer) and thus our work is more akin to the third group (application tuning) than the first one (self-tuning libraries), because we execute computational components that are not part of the library and can behave very differently from each other. Our efforts are close to the emerging area of (possibly automated) tuning of OpenCL or CUDA parameters [20], [21].

7. Conclusion

In this paper we presented our work on autotuning support for high-level pipeline patterns for heterogeneous many-core architectures. We have developed a pipeline tuning plugin for the Periscope Tuning Framework, in order to automatically determine the best combination of performance relevant tuning parameters exhibited by the pipeline coordination layer.

In our future work we will experiment with different search strategies and investigate methods for continuous online tuning, such that pipeline patterns are automatically adapted to changing work loads or varying target machine configurations. Moreover, we will extend our work to other common parallel patterns and other architectures [22].

Acknowledgment

This work was supported by the European Commission's FP7 project AutoTune under grant no. 288038 (see http://www.autotune-project.eu).

References

[1] T. Mattson, B. Sanders, and B. Massingill, "Patterns for Parallel Programming," Addison-Wesley, 2005.

[2] N. Bell and J. Hoberock, "Thrust: A Productivity-Oriented Library for CUDA," in *GPU Computing Gems, Jade Edition* (W. mei Hwu, ed.), Morgan Kaufmann, 2011.

[3] J. Enmyren and C. W. Kessler, "Skepu: a multi-backend skeleton programming library for multi-gpu systems," in *Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications*, HLPP '10, (New York, USA), ACM, 2010.

[4] J. Dokulil, E. Bajrovic, S. Benkner, M. Sandrieser, and B. Bachmayer, "HyPHI – Task Based Hybrid Execution C++ Library for the Intel Xeon Phi Coprocessor," in *42nd International Conference on Parallel Processing (ICPP-2013)*, Lyon, France, 2013.

[5] 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, pp. 28–41, 2011.

[6] S. Benkner, E. Bajrovic, E. Marth, M. Sandrieser, R. Namyst, and S. Thibault, "High-Level Support for Pipeline Parallelism on Manycore Architectures," in *Euro-Par 2012 Parallel Processing - 18th International Conference*, vol. 7484, pp. 614–625, 2012.

[7] M. Sandrieser, S. Benkner and S. Pllana, "Using Explicit Platform Descriptions to Support Programming of Heterogeneous Many-Core Systems," *Parallel Computing*, Volume 38, Issues 1-2, Pages 52-56, January-February 2012.

[8] 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, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011.

[9] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," Parallel and Distributed Systems, IEEE Transactions on, vol. 13, no. 3, 2002.

[10] B. Gary, *Learning openCV: Computer Vision with the openCV Library*. O'Reilly USA, 2008.

[11] R. Miceli, G. Civario, A. Sikora, E. Cesar, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin, and F. Bodin, "AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications," in *Applied Parallel and Scientific Computing*, vol. 7782 of *LNCS*, pp. 328–342, Springer, 2013.

[12] S. Benedict, V. Petkov, and M. Gerndt, "PERISCOPE: An Online-Based Distributed Performance Analysis Tool," in *Tools for High Performance Computing 2009* (M. S. Mueller, M. M. Resch, A. Schulz, and W. E. Nagel, eds.), pp. 1–16, Springer, 2010.

[13] M. Gerndt and E. Kereku, "Selective Instrumentation and Monitoring," in *Proceedings of 11th Workshop on Compilers for Parallel Computers (CPC 04), Kloster Seeon*, pp. 61–74, 2004.

[14] C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS Project," *Parallel Computing*, vol. 27, 2001.

[15] M. Frigo and S. Johnson, "FFTW: an adaptive software architecture for the FFT," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3, pp. 1381–1384, 1998.

[16] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August, "Compiler optimization-space exploration," in *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, 2003.

[17] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff, "Automatic selection of compiler options using non-parametric inferential statistics," in *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pp. 123–132, 2005.

[18] I.-H. Chung and J. K. Hollingsworth, "Using information from prior runs to improve automated tuning systems," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*.

[19] Y. L. Nelson, B. Bansal, M. Hall, A. Nakano, and K. Lerman, "Model-guided performance tuning of parameter values: A case study with molecular dynamics visualization," in *International Parallel and Distributed Processing Symposium*, pp. 1–8, 2008.

[20] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, "Program optimization space pruning for a multithreaded GPU," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 195–204, 2008.

[21] Y. Liu, E. Zhang, and X. Shen, "A cross-input adaptive framework for GPU program optimizations," in *International Parallel and Distributed Processing Symposium*, 2009.

[22] J. Dokulil and S. Benkner. "Automatic Tuning of a Parallel Pattern Library for Heterogeneous Systems with Intel Xeon Phi," in *12th International Symposium on Parallel and Distributed Processing with Applications (ISPA 2014)*, Milan, Italy, 2014.