Tuning OpenCL Applications with the Periscope Tuning Framework

Enes Bajrovic*, Robert Mijakovic †, Jiri Dokulil*, Siegfried Benkner*, Michael Gerndt†

*Universität Wien, Fakultät für Informatik, Vienna, Austria
†Technische Universität München, Institut für Informatik, Garching, Germany
{enes.bajrovic, jiri.dokulil, siegfried.benkner}@univie.ac.at
{mijakovic, gerndt}@in.tum.de

Abstract—Due to the complexity and diversity of new parallel architectures automatic tuning of parallel applications has become increasingly important for achieving acceptable performance levels as well as performance portability. The European AutoTune project developed a tuning framework which closely integrates and automates performance analysis and performance tuning. The Periscope Tuning Framework relies on a flexible plugin mechanism providing tuning plugins for different tuning aspects. This paper presents plugins for tuning the execution time of OpenCL kernels on three different architectures, namely standard multicore CPUs, Xeon Phi coprocessors, and GPUs. We present OpenCL tuning via the flags used during offline kernel compilation as well as through the selection of the most appropriate NDRange configuration, which defines the organization of parallel threads used for kernel execution. Both tuning plugins show significant performance impact and a clear dependence on the target architecture and thus improve performance portability via automatic tuning.

I. INTRODUCTION

Programmers developing HPC applications strive for achieving the highest possible performance. In addition to program development they spend considerable time in analyzing and tuning parallel applications on the target system. Due to the high pace and increasing diversity in hardware development this tuning step has to be repeated frequently to adapt to new architectures. Application tuning is usually executed manually with the help of performance analysis tools. A broad spectrum of such tools have been developed in the HPC context ranging from time profiling tools, to profiling of hardware performance counters, up to tracing the dynamic execution of the application. Some of the tools already provide automated analysis capabilities, thus going beyond a graphical interface facilitating the inspection of potentially large sets of performance data.

The European AutoTune project [1] extended Periscope [2], one of such automatic performance analysis tools, with fully automatic tuning support. The developed Periscope Tuning Framework (PTF) [3] provides a number of aspect-specific tuning plugins, that capture expert knowledge applied in the tuning of an aspect. Expert knowledge is required to increase the efficiency of the tuning, since a naive approach typically requires searching a highly dimensional tuning space.

This paper presents the support provided by the PTF for tuning of applications with kernels written in OpenCL for various types of standard processors and accelerators. OpenCL [4] is an open cross-platform programming interface for parallel applications on heterogeneous systems. While OpenCL provides functional portability of codes across different multicore and manycore platforms (CPUs, GPUs, accelerators), it does not guarantee performance portability. It has been shown, that the performance of OpenCL codes may vary significantly even between different hardware generations from a single GPU vendor, let alone different vendors. Also, due to the very different architectural characteristics of CPUs, GPUs and co-processors like the Intel Xeon Phi, it is even more challenging to achieve performance portability across such diverse types of architectures. Consequently, autotuning can play an important role towards improving the performance portability of OpenCL-based applications.

The contributions of this paper comprise the development and evaluation of two tuning plugins targeting OpenCL based applications. The first plugin allows to automatically choose the best combination of compiler flags for OpenCL kernels, while the second optimizes the NDRange parameter which controls how parallel threads are organized for kernel execution. A novel search strategy that explores the dimensions of the search space individually has been developed and integrated with the plugins.

This paper is organized as follows: The next section of the paper briefly discusses related work on automatic tuning of parallel applications and OpenCL applications in particular. Section III provides an overview of the PTF and the general concept of tuning plugins. Section IV presents the Compiler Flag Selection plugin used to tune the kernel by choosing the right compiler switches in an offline compilation approach and introduces the individual search strategy for coping with potentially large search spaces. The NDRange plugin is presented in Section V. It modifies the local work group size of OpenCL kernels via runtime tuning actions and evaluates different settings in order to optimize overall application execution time. Section VI presents results from applying both plugins to several OpenCL benchmarks on different parallel architectures.

II. RELATED WORK

Autotuning for parallel applications is of growing interest and many research groups are pursuing different approaches to autotuning [5], [6], [7]. Early work on autotuning was in the
context of self-tuning libraries for linear algebra and signal processing like ATLAS [8], FFTW [9], and SPIRAL [10]. A number of techniques and tools that automatically analyze alternative compiler optimizations and search for their optimal combination [11], [12], [13] have been developed. These tools use either iterative search techniques or machine learning techniques to cope with the potentially huge search space. In the context of our work we have developed the individual search strategy which explores one dimension of the search space at a time, significantly speeding up the search while still delivering close to optimal results in most cases. Another category of autotuning systems focus on exploring a space of application-level parameters that are believed to impact the performance of an application [14], [15], [16]. Several other frameworks have been proposed that try to combine ideas from all the other groups [17], [18]. The X-TUNE project [19] aims at seamlessly integrating programmer-directed and compiler-directed auto-tuning.

In the following we discuss a few related research efforts which focus specifically on autotuning of OpenCL/CUDA applications. While low-level programming interfaces like OpenCL or CUDA provide the means to optimize programs, this comes with the significant drawback that a large amount of development time has to be invested for tuning codes for each specific type of processor/accelerator used. Performance portability, i.e. high performance of the same tuned code observable on different architectures, is difficult [20], [21]. Manual optimizations for different architectures, besides being complex and time-consuming, quickly result in code that is difficult to understand, modify, and maintain.

The ViennaCL linear algebra library [22] provides an optional autotuning feature, which selects optimal execution parameters for the library kernels through exhaustive search. A similar approach was also investigated for MAGMA [23] and other linear algebra libraries [24], [25]. It is also possible to include a code generator in the tuning process [26] and select not just the parameters, but also choose the fastest implementation of the desired operation [27]. Some implementations take a two-staged approach [28], where several codelets are generated for the target platform at installation which are then run at runtime to build the search space of implementation variants. Only recently, approaches which attempt to tune general purpose applications, similar to our effort, have started to emerge, for example using higher-level languages like OmpSs [29] and HMPP [30] or by allowing developers to expose tuning parameters [31].

III. PERISCOPE TUNING FRAMEWORK

Closely integrating performance analysis and tuning, the Periscope Tuning Framework (PTF) goes beyond automatic performance analysis by allowing to automatically tune applications with respect to various performance aspects by means of tuning plugins. Currently, the PTF provides tuning plugins for optimizing MPI communication performance, OpenMP scalability, Pipeline Pattern execution [32], energy efficiency, and OpenCL kernel performance. A rich toolbox for implementing new tuning plugins is provided.

A. Periscope

Periscope comprises a graphical user interface, a frontend, an analysis agent hierarchy (which in case of MPI applications is distributed across the nodes of a cluster), and the MRI monitor that is linked to the application. The analysis capabilities of Periscope are implemented by all these layers. The user interface allows to inspect the found performance properties in Eclipse, the frontend triggers performance analysis strategies that are executed by the analysis agents, and the MRI monitor measures performance data required for the automatic identification of performance properties.

B. Tuning Plugins

Tuning plugins optimize applications for a specific tuning aspect by combining performance analysis and tuning. They follow a predefined tuning model (Figure 1) that defines the sequence of operations that all plugins have to implement.

Each tuning plugin goes through one or more tuning steps. Each tuning step might start with a pre-analysis that runs one of Periscope's performance analysis strategies to gather properties of the application. These properties can then be considered by the plugin when the search space is determined. The plugin uses one of PTF's search algorithms to search for the variant in the space which optimizes the objective function. In a search step, the search algorithm creates a set of variants, represented as tuning scenarios. The variants are then processed and evaluated by the plugin within individual experiments. Before the execution of an experiment starts, the variant can be prepared (e.g., the application can be recompiled). While the experiment is executed, the plugin can request any PTF performance analysis strategy to be run during the experiment to get detailed information on the effect of the tuning.

The plugins typically investigate a number of variants to identify optimizations. In this process, analysis information is used to shrink the search space and to improve the search efficiency of plugins. On request of the plugins, performance analysis strategies are executed and the found performance properties are returned to the plugin to provide dynamic information. In addition to dynamic information, Periscope also

Figure 1. PTF Tuning Model.
for a given $S$ are standard Periscope properties, phase region $T$ tuning objectives $F$ scenarios, where · approach of the individual keep factor is the total number of TPs. is the maximal number of values for any given TP. $O$, $m$ is the number $n$ is usually defined to be the body of the main progress loop of the application, e.g., the time stepping loop of a simulation. Different experiments can be run on consecutive executions of the phase region without the need of restarting the entire application, provided, the instances of the phase region have the same behavior.

The tuning objectives are standard Periscope properties, while the implementation of an optimization is realized by tuning actions. These tuning actions might be executed at runtime to, for example, assign a certain value to a variable. They are propagated by the analysis agents to the MRI monitor, which performs the actions at a predefined state of the program execution.

The PTF offers a set of standard search algorithms, including exhaustive search, probabilistic random search, individual search, multi-objective genetic search, and Active Harmony’s [33] Nelder-Mead Simplex algorithm. The search algorithms are dynamically loaded on request of tuning plugins. Thus, as the plugins itself, search algorithms can be provided in source or binary form.

IV. OPENCL COMPILER FLAG SELECTION TUNING

The OpenCL Compiler Flag Selection plugin (OCL-CFS) supports tuning OpenCL kernels with the help of compilation flags presented to an OpenCL offline compiler. A predetermined list of compiler flags is provided in a compiler-specific configuration template. This template can be enhanced with application specific flags and build information to guide the execution of the plugin.

The plugin constructs flag combinations that are evaluated by recompiling the application’s kernels and automatically executing the application. The recompilation is based on the application’s makefile that exposes an environment variable that takes the selected flag combination. After the plugin has selected a flag combination it touches the kernel source files and triggers the make process. After the build is finished, the application is automatically restarted by the PTF and execution time is measured for the OpenCL kernels. After the evaluation of the selected combinations, the plugin analyzes the measured execution times and outputs the best flag combination per kernel.

A. Performance Analysis

The OCL-CFS plugin and the NDRange plugin use the OpenCL analysis strategy to measure the performance impact on the kernels. This strategy is triggered during the experiment by the analysis agents. The agents instruct the monitoring system, which is linked to the application to measure the execution of the kernels. Based on a wrapper library of the OpenCL API functions, the monitor uses a given OCL event object or creates a new event object to measure the kernel execution time. Once the kernel has been started, the monitor waits for its execution and then retrieves the execution time. At the end of the experiment, which can either be the end of the application or the end of the phase region, the aggregated execution time of each kernel is returned to the analysis agent. The analysis agent creates a performance property for each kernel and propagates it to the plugin. For each tested flag combination the execution time is available for all the kernels and thus, the plugin can determine the best combination for each kernel.

Due to the large number of tuning flags, exhaustive analysis might not be possible. Therefore, the OCL-CFS plugin provides several search algorithms to reduce the number of evaluations. Besides individual search, which is described below, it can apply also a machine learning based random search, as well as the GDE3 genetic search [34].

B. Individual Search Strategy

The individual search assumes that the list of compiler flags or, more general, Tuning Parameters (TP) is ordered according to their importance. It iterates through the list of TPs and incrementally adds a new parameter to the already explored set of parameters. It thus investigates in each search step the effect of the one new tuning parameter in combination with the already processed parameters.

At the end of each search step the range of the current new TP is shrunk to a configurable number of best values. Only these values will be considered for this TP for the successive search steps.

In its most restrictive version, the individual search strategy only needs to test $\sum_i^n m_i$ scenarios, where $m_i$ is the number of values for TP $i$ and $n$ is the total number of TPs.

The general time complexity of the individual search algorithm is $O(k^n \cdot m \cdot n)$, where

- $k$ is the keep factor - the maximal number of values for one TP tested on subsequent steps, with $k < m$,
- $m$ is the maximal number of values for any given TP.

An important observation is that the individual search improves the total tuning time in two ways.

First, the maximum number of tested scenarios is decreased. Although it remains exponential, the base factor $k$ is usually very small. Common used values for $k$ are 1 and 2, whereas $m$ could be any value.

Secondly, based on the greedy approach of the individual search, the scenarios which are pruned from the exhaustive search space are with high probability the scenarios with worse objective values. For those cases where the objective function is the execution time, the pruned scenarios are also the scenarios with highest execution time of the application. Thus, the total search time is once again improved.

The individual search algorithm (see Algorithm 1) is implemented on top of the existing exhaustive search. It first
The vector containing the set of all possible values for the TP is now restricted to the chosen \( k \) values. This operation is called \textit{vector restriction} and results in shrinking the search space. When a new set of scenarios is created with exhaustive search on the next search steps, only those \( k \) best values of the TP are going to be used.

If the new TP only led to scenarios that are not better than the current best scenario, this TP is deleted from the internal list. The algorithm repeats this step until all the given TPs are processed. It then returns the best found configuration.

The quality of the final result depends on the order of the given sequence of TPs. TPs with higher performance impact should be listed at the beginning of the sequence.

The \( k \) coefficient introduces a relaxation of the algorithm, giving the opportunity to cover cases where there are dependencies between the TPs. For example, the value of the current TP associated with the second best scenario might deliver better results when combined with the next TP, than when combining the value of the current TP in the best scenario with the next TP. In this case, setting \( k = 2 \) also covers the better scenario.

\section*{C. OpenCL Offline Compilation}

Before a scenario is executed, the kernels that are to be tuned have to be recompiled with the specified flags. Recompilation of a kernel in the PTF is specific to each OpenCL vendor and is therefore performed by a script that is executed by the plugin. The PTF currently supports two OpenCL vendors, Intel and NVIDIA, and their target devices. Intel provides a kernel builder and an LLVM-based optimizer while NVIDIA does not provide an offline compiler. The compilation of a kernel with the Intel toolchain is depicted in 2(a). First the \texttt{ioc} tool generates the SPIR (Standard Portable Intermediate Representation) file, according to selected OpenCL compiler flags. This intermediate version is input to the \texttt{oclopt} optimizer, which generates on optimized code version in form of a new SPIR file. The optimization flags defined by the OpenCL standard relax mathematical precision and thus trade correctness for speed. The \texttt{oclopt} optimizer provides a large number of standard compiler optimization flags. To perform the optimizations it executes a number of passes including specific analyses and optimizations.

As shown in Figure 2(b), for NVIDIA we have developed an offline compiler which wraps the online compiler and uses besides the OpenCL standard optimizations two additional optimization defined by NVIDIA, i.e., general optimization levels and the maximum register count. The selected optimizations are propagated to NVIDIA's \texttt{ptxas} optimizing backend compiler. In addition, the PTF supports loop unrolling based on NVIDIA's loop unroll pragma extension. The programmer can insert the unroll pragma into the code and direct the unroll factor via a tuning parameter given to the plugin via the configuration file.

\section*{V. OpenCL NDRANGE TUNING}

When launching a kernel in OpenCL, several parameters that control the execution of the kernel have to be specified by the programmer. An important concept is the NDRange, which controls the number of work items to be processed by the kernel and the way these work items are divided into work groups. The NDRange specifies the number of times the kernel is executed (in parallel), processing a different work item each time, but also how parallel execution is organized and how threads are mapped to the available compute units. Each work item corresponds to a point either in 1D, 2D, or 3D space ("ND" in the NDRange). When launching the kernel, it is necessary to specify a range in each dimension, which defines...
the N-dimensional space that is to be processed by the kernel. This is called global work size.

The work items are partitioned into smaller chunks – the n-dimensional space is divided into equally sized n-dimensional work groups. The size of the work groups is also defined by the NDRange parameter and it is called local work size. The global work size in any dimension must be divisible by the local work size in that dimension. A work group shares local memory and can be synchronized using barriers. The way the work group is mapped to the actual hardware is not prescribed by the OpenCL standard and implementations are free to choose the strategy, as long as the basic constraints are observed (i.e., all items in a work group must have access to the same local memory). On NVIDIA devices, the work group is assigned to a single streaming multiprocessor and each work item is mapped to a single thread, which are grouped into warps consisting of 32 threads. On CPUs and the Xeon Phi, the work items are grouped to match the width of the vector units, but since no hardware local memory is present, they can be executed by all available cores. The TBB library, with its scheduler based on task-stealing, is used for the execution.

Usually, the global work size is defined by the problem size, with small adjustments due to issues like padding or alignment. The local work size may also be dictated by the algorithm and the problem size, but in many cases, the local size can be selected from a range of options. The exact range of options may depend on the actual kernel and the device. For example, the kernel may be able to run with local sizes that are powers of two (due to the way the implemented algorithm works), but the device can only support local sizes up to 512, so the options would be 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512. The local work size may also affect the amount of resources (local memory, registers, etc.) needed to process a work group. In some cases, the kernel is written in such a way that it can run (it adapts) with any valid (remember that it has to evenly divide the global work size) local work size or it is completely independent of the local work size. In these cases, the local work size can still significantly affect kernel performance.

The local work size affects the scheduling overhead, cache utilization, memory access patterns, thread organization, thread scheduling etc. An important aspect is the fact that a work group shares the local memory which is much smaller, but also much faster than the global memory. The local memory may be used to cache data from the global memory or to store intermediate results. The required size of the local memory often depends on the work group size, with linear or quadratic memory requirements being the common cases. A larger work group may lead to decreased overhead (e.g., by decreasing the number of times the data must be copied to the cache in the local memory), but the restricted size of the available local memory limits the work group size.

Another possible concern is the way the size and shape of the work group affects memory accesses. In GPUs, the performance of memory access is affected by the way parallel threads access the memory. Certain access patterns are more efficient. For example, on NVIDIA GPUs, a group of threads running in lockstep should read from the same location in constant memory, otherwise the access would be serialized. Normally, the accessed memory location depends on the global coordinates of the work item, which means that grouping of the work items changes the overall memory access pattern. For example, let’s consider a 2D kernel where each work item requires reading a value from a matrix at a position which is the same as the coordinates of the work item. If the device executes 8 threads in lock-step and there is no divergence (branching), then a local work size of 8x1 means that the threads read a part of a row in parallel. If the local work size is 1x8, the threads read a part of a column. With certain types of memory and organization of the matrix, this could produce an 8x difference in the performance of the read operation, since the reads could be performed by one memory transaction one way, but would require serialization and therefore 8 memory operations the other way. As a result, the local work size can significantly influence the performance of the kernel execution and can be used for performance tuning.

A. NDRange Tuning Specification

We support such tuning via an OpenCL NDRange Tuning PTF plugin (OCL-NDR). The goal is to minimize execution time by automatically selecting the best local work size for each kernel in the application. The plugin requires a user-provided NDRange tuning specification (in XML) of feasible local work sizes for OpenCL kernels. For each kernel in the application up to three tuning parameters may be required, one for each local work size in up to three dimensions.

The different NDRange configurations for a kernel (scenarios to be explored by the plugin) may either be enumerated explicitly or by providing a value range in the form of min, max and increment for each NDRange dimension. The second option (explicit enumeration) is provided since OpenCL kernels may be written in such a way that only a specific local work sizes will allow the application to execute correctly (e.g., the algorithm correctness may be affected if the local work size is not divisible by 8). Moreover, the maximum amount of memory may be limited, hence making larger local work sizes unusable. The OpenCL API provides tools to query device limitations and limits for a specific kernel. For example, it is possible to query the maximum work group size a device can support for any kernel and there is a second call that provides the limit for a kernel, taking into account issues like register usage. This kernel-specific limit on the local work size together with the requirement that the local work size must evenly divide the global work size can be used to provide a basis for the tuning specification, but some of the values may be also removed due to the limitations of the implemented algorithm.

B. Tuning Strategy

The OCL-NDR plugin processes the user provided NDRange tuning specification and constructs a search space, which can then be explored by one of the search strategies available within the PTF. Both the way how the search space
is constructed and how it is explored can be influenced by the user through the tuning specification. Currently two different tuning strategies are supported, a local and a global strategy.

Using the local tuning strategy, each OpenCL kernel in the application is tuned separately. For each kernel, a kernel-specific search space is constructed according to the tuning specification and then explored with a search strategy in order to find the NDRange configuration that minimizes the execution time of the kernel. The same process is repeated for all other kernels in the application.

With the global tuning strategy a global search space is determined by the cross product of all kernel-specific search spaces. This global search space is then explored by a search strategy with the objective of minimizing overall application execution time.

Provided that the user specified tuning specification covers all feasible NDRange configurations for all kernels, the global tuning strategy in combination with exhaustive search will find the scenario with minimal execution time. However, if the application contains many kernels the search space might become prohibitively large. With the local search strategy the number of scenarios to explore is given by the sum of kernel-specific scenarios, which in most cases will be much smaller.

If the kernels in an application are independent of each other, the local tuning strategy usually will deliver the same results as the global strategy.

Both strategies rely on Periscope’s OpenCL performance analysis facilities to obtain the execution time of the individual kernels and the whole application. The OpenCL NDRange plugin currently can use either the exhaustive search strategy or the individual search strategy.

VI. Evaluation

Evaluation of the two OpenCL tuning plugins has been performed with several benchmarks from the Rodinia suite [35] and a Direct Coulomb summation benchmark. We have performed the experiments on different architectures including a traditional GPU (NVIDIA K20m, 2496 CUDA cores), an Intel Xeon Phi (5110P, 1.053 GHz, 60 cores), an integrated GPU (Intel HD 4000, 16 execution units), and a CPU (dual Intel Xeon E5-2650, 2.00 GHz, 8 cores).

A. OpenCL Compiler Flag Selection Tuning Plugin

In the following we present evaluation results1 for the OpenCL compiler flag selection tuning plugin for three different benchmarks from the Rodinia benchmark suite[35]. Evaluation was performed using mainly the exhaustive strategy to investigate the complete search space.

The flag combinations explored by the OpenCL compiler flag selection tuning plugin are shown in Table I.

Table I

<table>
<thead>
<tr>
<th>Flag comb.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-O2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-O3</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-loop-vectorize</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-prefetch</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1) LavaMD: The LavaMD benchmark calculates particle potential and relocation due to mutual forces between particles within a large 3D space. This space is divided into cubes, or large boxes, that are allocated to individual cluster nodes. The large box at each node is further divided into cubes, called boxes. 26 neighbor boxes surround each box (the home box). Home boxes at the boundaries of the particle space have fewer neighbors. Particles only interact with those other particles that are within a cutoff radius since ones at larger distances exert negligible forces. Thus the box size is chosen so that the cutoff radius does not span beyond any neighbor box for any particle in a home box, thus limiting the reference space to a finite number of boxes.

Figure 3 shows the measured kernel execution times for experiments with three different grid sizes, (a) 5, (b) 10, (c) 20 on the Intel Xeon CPU and the Xeon Phi. The figure shows that the best combination of flags mainly depends on the type of device, and to a lesser extent on the data set. The best combination of flags on the Intel Xeon Phi is O3 with loop vectorization and prefetching for dataset (a), O2 for dataset (b) and O1 with prefetching for dataset (c). The global best combination for the Xeon Phi, independent of the dataset, is 2 with loop vectorization and prefetching. On the other hand, the best combination of flags on the Intel CPU for dataset size (a) is O2 with prefetching, for (b) it is O3 with prefetching and for the dataset (c) it is O3. It is worth mentioning that selecting the flags combination highly depends on the device.

---

1 We applied to OpenCL compiler flag selection tuning plugin also to the Needleman-Wunsch benchmark and the Direct Coulomb Summation code, but since no performance differences could be observed for different compiler flags combinations we omit the results for these two benchmarks.
selected. The execution time improved on Xeon Phi between 11% and 29%, while on the CPU the improvement was much bigger, between 38% and 56%, depending on the dataset.

2) PathFinder: The PathFinder benchmark uses a dynamic programming algorithm to find a path on a 2-D grid from the bottom row to the top row with the smallest accumulated weights, where each step of the path moves straight ahead or diagonally ahead. It iterates row by row, each node picks a neighboring node in the previous row that has the smallest accumulated weight, and adds its own weight to the sum.

Figure 4 shows the kernel execution times for three grids with different widths, (a) 100K, (b) 200K, (c) 400K on the Intel Xeon CPU and the Xeon Phi. The figure shows that again the best combination of flags weakly depends on the dataset size but more depends on the type of device. The best combination of flags on the Xeon Phi is O1 depending on the dataset size with prefetching, (a) and (c), or with loop vectorization, (b). The global best combination for the Xeon Phi, independent of the dataset, is O1 with prefetching. On the other hand, the best combination of flags on the Xeon CPU is again O3 but depending on the dataset size with loop vectorization and prefetching, (a) and (c), or with loop vectorization, (b), and with prefetching, (c). The global best combination for the Xeon CPU, independent of the dataset, is O3 with loop vectorization and prefetching. It is worth mentioning that selecting the most aggressive optimization results in almost the slowest performing code. The difference between the slowest and the fastest version is between 26% and 30% depending on the dataset.

3) HotSpot: The HotSpot benchmark estimates processor temperature based on an architectural floorplan and simulated power measurements. The thermal simulation iteratively solves a series of differential equations. Each output cell in the computational grid represents the average temperature value of the corresponding area of the chip.

Figure 5 shows the kernel execution times for three different grid sizes, (a) 64 x 64, (b) 512 x 512, (c) 1024 x 1024. The experiments were performed only for the Intel Xeon CPU, as on the Xeon Phi the kernel did not run. The figure shows that the best combination of flags slightly depends on the dataset size. The best combination of flags for dataset size (a) is O2 with prefetching, for (b) it is O1 with prefetching, and for (c) it is O1 with loop vectorization. The global best combination of flags is O2 with prefetching, as it results in only 1.6% of lost performance on average, compared to the best result per dataset. It is worth mentioning that selecting the most aggressive optimization results in almost the slowest performing code. The difference between the slowest and the fastest version is between 26% and 30% depending on the dataset.

B. OpenCL NDRange Tuning Plugin

For the NDRange tuning plugin we present evaluation results for the Needleman-Wunsch benchmark from the Rodinia benchmark suite and for a Direct Coulomb Summation benchmark.

1) Direct Coulomb summation: This code computes a potential field created by point charges (atoms). The grid is evenly spaced in 2D, the atoms are placed randomly at distinct grid locations. For each pair of grid point and atom, we compute the distance between the two and divide the atom’s charge by the distance to get the atom’s contribution to the

---

2NDRange tuning for the LavaMD, Pathfinder, and Hotspot benchmarks did not show significant performance improvements or was not applicable due to the specific coding style and are thus omitted.
potential at the grid point. The contributions are then added together to produce the final value.

The algorithm is implemented as a 2D OpenCL kernel, where each work item represents one point in the grid. The 2D grid data is linearized into a single buffer. The atom properties (x-coordinate, y-coordinate, and charge) are each stored as an array of values. The kernel contains a for-loop, which iterates over the particles and computes the contribution for each particle. The aggregated result is then saved to the appropriate place in the grid buffer.

The size of the grid used in the experiments was 1024x1024, which means the global size and therefore the maximal local work size is 1024x1024. This gives us at most 11x11 (2^0...2^10) different local work size combinations to test. The total work group size (number of items in the work group) $size = size_x \times size_y$ ($size_x$ and $size_y$ are sizes in the x and y dimensions) is limited by the amount of available local resources (registers and local memory).

We have performed an exhaustive search of the work group sizes, performing an experiment for each size. Each experiment was repeated 100 times and the displayed time is the average. The measured time is a wall clock time and it does not include data transfers. If a specific work group size is given in the results, it is in the format $(size_x, size_y)$. For example $(512,1)$ is a work group with 512 items in the x-dimension and 1 item in the y-dimension. For each device, we have also allowed the vendor runtime to try choosing the size. If no local work size is specified, the runtime selects a valid

Figure 6. Performance of the Coulomb kernel on different OpenCL devices with different NDRange configurations.
value, but it is not guaranteed to be optimal. We will refer to this value as the default. In our experiments, the default size was always \((d_{\text{fs}}, 1)\), where \(d_{\text{fs}}\) tends to be close to the maximal possible value.

Figure 6 shows the most common patterns encountered in our experiments. The graphs for the Xeon Phi and CPUs look like a cliff. To get good performance, \(size_{\text{x}}\) has to be above a certain threshold. This is caused by the vector instructions available on CPUs and the Xeon Phi. The work group must be wide enough to allow to use the vector instructions. Since the vector units of the Xeon Phi are wider than on the CPUs, the threshold is at a higher value. On the GPUs, each item in a work group is usually assigned to a GPU thread. If the number of items in the work group is smaller than the number of threads that work in lockstep, not all threads will be used, leading to significant inefficiency. On the other hand, the GPUs can usually efficiently map all work groups with the same number of items but different shapes (e.g., \((8, 1)\), \((4, 2)\), \((2, 4)\), and \((1, 8)\)). As a result, there is a “mountain” around \((1, 1)\). In some cases, it may be more efficient to use smaller work groups, for example due to the extra overhead caused by executing a work group with more items than there are threads. When these effects combine, we may get a graph similar to 6(c), with a “valley” just below the mountain. In our case, this is the point where a work group contains just enough items to use all threads, but no more than that.

The speedups achieved for the Coulomb code on the Xeon Phi and the CPU are negligible. However, the NVIDIA GPU and especially the Intel GPU can significantly benefit from selection of the right work group size, providing speedups of 7% and 56%, respectively.

The individual search strategy with a keep factor of 1 reduced the number of scenarios to 20 and 22, respectively, (from 100 and 121 with exhaustive search), but nevertheless found configurations with a performance very close (within 5%) to the best configurations found by exhaustive search.

2) Needleman-Wunsch: The Needleman-Wunsch benchmark determines the optimal global alignment of amino acid sequences of two proteins by comparing all base pairs. The algorithm uses a matrix and calculates the score for every element in the matrix, and then traces back in order to get the optimal path for the alignment. The OpenCL version from the Rodinia benchmark suite has been adapted to support tuning with the OCL-NDR plugin. The code utilizes the local memory of the device, which limits the range of possible values for the local work group size. Furthermore, the code is written in such a way that the sequence size, which was 2880 in our experiments, has to be divisible by the local work group size. Taking into account these restrictions, 11 possible local work group sizes in the NDRange tuning specification were explicitly enumerated \(\{18, 12, 16, 20, 24, 32, 36, 40, 48, 60, 64\}\). Exhaustive search was used to explore the search space.

The results on three different OpenCL devices are shown in Figure 7. On the NVIDIA Tesla K20m (green line) best performance was achieved with a local work group size of 20, while a significant performance degradation could be observed for work group sizes of 48 and above. As observed with the Coulomb application, smaller work groups yielded better results on average. On the Intel Xeon Phi 5110P coprocessor (blue line) the best execution times were achieved with local work group sizes of 48 and 32. For the local work group size of 64 we ran out of memory. On the Intel Xeon E5-2650 CPU (red line) we observed steady performance improvement for local work group sizes up to 32, which was the best performing scenario. For larger local work group sizes performance remained the same or degraded.

On the Xeon Phi the best version is 50% faster than the version with local work group size 8 and 24% faster than the version with local work size 16, which is the default local work group size used in the Rodinia OpenCL code. On the Xeon CPU the effect is less pronounced, with a speedup of 25% compared to the slowest configuration, and 5% compared to the Rodinia default. Finally, on the NVIDIA GPU we observed a speedup of up to 67% compared to the slowest configuration that used local size of 64, and a speedup of 30% compared to the default local work group size of 16.

VII. CONCLUSION AND FUTURE WORK

This paper presented the tuning plugins developed for the Periscope Tuning Framework to automatically tune OpenCL kernels on various accelerators. The OCL-CFS plugin tunes the OpenCL compiler flags in an offline compilation approach while the OCL-NDR plugin tunes the NDRange parameter of kernel execution. Both plugins showed significant performance impact and a clear dependence on the target architecture and thus improve performance portability via automatic tuning. The plugins make use of the overall services of Periscope to perform the tuning. These include predefined search algorithms as well as static and dynamic application information. With these plugins the range of target architectures addressed
by the PTF was significantly extended to cover also HPC systems equipped with GPUs and Xeon Phis as accelerators.

Future extensions could cover further tuning aspects, such as directive-based tuning of the OpenCL code, e.g., with respect to register usage, and tuning the host-device data transfers.

The Periscope Tuning Framework including the described OpenCL tuning plugins (amongst several other plugins for optimizing performance and energy efficiency of MPI and OpenMP applications) is available as open source at PTF Release available as open source at http://periscope.in.tum.de.

REFERENCES