Notification

This is the author version of: Lu Li, Usman Dastgeer, Christoph Kessler: Adaptive off-line tuning for optimized composition of components for heterogeneous many-core systems. Accepted for Proc. Seventh International Workshop on Automatic Performance Tuning (iWAPT-2012), 17 July 2012, Kobe, Japan. To appear in: Proc. VECPAR-2012 Conference, Kobe, Japan, July 2012.

Publisher: the post-workshop issue of Springer's Lecture Notes in Computer Science (LNCS), the volume number assigned by Springer to the LNCS is 7851.

Adaptive off-line tuning for optimized composition of components for heterogeneous many-core systems

Lu Li, Usman Dastgeer, and Christoph Kessler

PELAB, IDA,Linköping University
S-581 83 Linköping, Sweden
{lu.li,usman.dastgeer,christoph.kessler}@liu.se

Abstract. In recent years heterogeneous multi-core systems have been given much attention. However, performance optimization on these platforms remains a big challenge. Optimizations performed by compilers are often limited due to lack of dynamic information and run time environment, which makes applications often not performance portable. One current approach is to provide multiple implementations for the same interface that could be used interchangeably depending on the call context, and expose the composition choices to a compiler, deployment-time composition tool and/or run-time system. Using off-line machine-learning techniques allows to improve the precision and reduce the run-time overhead of run-time composition and leads to an improvement of performance portability. In this work we extend the run-time composition mechanism in the PEPPHER composition tool by off-line composition and present an adaptive machine learning algorithm for generating compact and efficient dispatch data structures with low training time. As dispatch data structure we propose an adaptive decision tree structure, which implies an adaptive training algorithm that allows to control the trade-off between training time, dispatch precision and run-time dispatch overhead.
We have evaluated our optimization strategy with simple kernels (matrix-multiplication and sorting) as well as applications from RODINIA benchmark on two GPU-based heterogeneous systems. On average, the precision for composition choices reaches 83.6 percent with approximately 34 minutes off-line training time.

Keywords: Autotuning, Heterogeneous architecture, GPU

1 Introduction

Recently GPU-based heterogeneous multi-core system have been given much attention, because GPUs have shown remarkable performance advantage over CPUs for suitable computations with sufficiently large problem size. However, effective utilization of those systems often requires much programming effort

(programmability problem), and moreover, we often observe a performance decrease when porting the code to a new platform without re-optimization (performance portability problem).

For building performance portable applications, one solution is to provide multiple implementation variants of the same functionality that may execute on different platforms, internally use different programming models, different algorithms and/or different compilation settings, or encapsulate library calls or accelerator-specific code. The execution time of such variants will generally depend on the resources available for execution (e.g., cores or accelerator) and other call context properties such as problem sizes, but also on tunable parameters of the implementation variants themselves such as buffer sizes or tiling factors.

The PEPPHER [6,4] component model provides a XML-based metadata language that allows to specify descriptors that externally annotate PEPPHER components and interfaces. A component is an annotated software module adhering to an PEPPHER interface for which multiple implementation variants may be available. Beyond the traditional functional interface properties such as parameter types and direction, component metadata of an implementation variant includes the implemented interface (functionality), dependences on other PEPPHER components or third-party software packages, compilation commands, tunable parameters, platform and resource requirements, and possibly also statically provided performance models that allow to predict average-case execution time as a function of values taken from a call context instance. Hence, PEPPHER allows to delay and expose the selection decisions to later stages (e.g., at runtime) when more information about the invocation context and resource availability (e.g. from the run-time environment) is available. In this way, the selection of an implementation variant for an interface function call is completely automatized and not hardcoded in the application, allowing for automated re-optimizing of the selection mechanism when porting a PEPPHER application to a new platform.

In order to better utilize different kinds of processing units by appropriate automatic selection, a reasonably good performance model for predicting the fastest implementation variant for a given context instance is required. The two trends for building such performance models are towards an analytical model and an empirical model. It is normally considered that modern computer systems (including heterogeneous ones) are too complex for a reasonably good analytical performance model, thus empirical models constructed from measurements of test code on the target system have become more practical nowadays. Machine-learning techniques have shown potential for building such empirical performance models. In essence, machine learning constructs from results of example runs a surrogate function that approximates an unknown selection function for a (new) target architecture.

Empirical automated performance tuning (or autotuning for short) of best-variant selection by measurements and learning can be performed on-line or off-line. On-line learning is done at runtime, after first instrumented invocations of components have been executed with random selection decisions, and represents

the selection function in an internal data structure, such as a hash table as applied in StarPU [3].

On-line machine learning performs selection decisions purely based on recorded performance history data and thus does not require any additional performance modeling information by the component provider, but can not offer good prediction results until enough representative example measurements are collected, and incurs additional runtime overhead for that. Off-line tuning can ease the problem by actively invoking those representative training examples manually or automatically; however, the number of training examples generated with a straightforward strided scanning of context property values (e.g., problem sizes) grows very large if suitable precision of performance prediction and best-variant selection shall be achieved.

In this work we suggest a new approach to off-line tuning with a novel adaptive generation of training data and representation of the constructed selection (dispatch) function. In our approach, the training time can be reduced remarkably while a reasonable prediction precision can still be achieved. It can also be integrated with compile time tools such as composition tools, thus enhance static composition by better precision. Furthermore, it can be integrated with run-time systems such as StarPU by dynamically exposing only the best implementations of the different kinds of processing units to reduce run-time selection overhead.

The remainder of this paper is organized as follows: Section 2 introduces the PEPPHER component model and composition tool. In section 3 we discuss our adaptive offline tuning approach in detail. In section 4 we show and discuss experimental results. Section 5 lists related work; section 6 concludes and discusses future work.

2 PEPPHER Components and Composition

A *PEPPHER component* is an annotated software module that implements a specific functionality declared in a *PEPPHER interface*. A PEPPHER interface is defined by an *interface descriptor*, an XML document that specifies the name, parameter types and the access types (read, write or both) of a function to be implemented, and in addition specifies which performance metrics (e.g. average case execution time) the prediction functions of component implementations must provide. Interfaces can be generic in static entities such as element types or code; genericity is resolved statically by expansion, as with C++ templates.

Applications for PEPPHER are currently assumed to be written in C/C++. Several component variants may implement the same functionality (as defined by a PEPPHER interface), e.g. by different algorithms or for different execution platforms. These implementation variants can exist already as part of some standard library (e.g. CUBLAS components for CUDA) or can be provided by the programmer. The PEPPHER framework provides support for implementation repository to manage evolution of implementation variants to increase the re-use potential in the long run. Also, more component implementation variants

may be generated automatically from a common source module, e.g. by special compiler transformations or by instantiating or binding tunable parameters. These variants differ by their resource requirements and performance behavior, and thereby become alternative choices for composition whenever the (interface) function is called.

In order to prepare and guide variant selection, component implementations need to expose their relevant properties explicitly to the composition tool. Each PEPPHER component implementation variant thus provides its own *component descriptor*, an XML document that contains information (meta-data) about properties such as the provided and required interface(s), source files, compilation commands and resource requirements, tunable parameters, further constraints on composition, and a reference to a performance prediction function.

The `main` module of a PEPPHER application is also annotated by its own XML descriptor, which states e.g. the target execution platform and the overall optimization goal.

The PEPPHER framework automatically keeps track of the different implementation variants for the identified components, technically by storing their descriptors in repositories that can be explored by the composition tool. The composition tool reads the metadata of interfaces and components used in the application and generates, for each call to a PEPPHER interface, the necessary code for pre-selecting (dispatching) a suitable implementation variant and creating a task for the PEPPHER runtime system that will execute that call. Composition points of PEPPHER components are restricted to calls on general-purpose execution units only. Consequently, all component implementations using hardware accelerators such as GPUs must be wrapped in CPU code containing a platform-specific call to the accelerator.

Component invocations result in *tasks* that are managed by the PEPPHER run-time system and executed non-preemptively. PEPPHER components and tasks are *stateless*. However, the parameter data that they operate on do have state. For this reason, parameters passed in and out of PEPPHER components may be wrapped in special portable, generic, STL-like *container* data structures such as `Vector` and `Matrix` with platform-specific implementations that internally keep track of, e.g., in which memory modules of the target system which parts of the data are currently located or mirrored (*smart containers*). The container state becomes part of the call context information as it is relevant for performance prediction.

Composition tool *Composition* is the selection of a specific implementation variant (i.e., callee) for a call to component-provided functionality and the allocation of resources for its execution. Composition is made *context-aware* for performance optimization if it depends on the current *call context*, which consists of selected input parameter properties (such as size) and currently available resources (such as cores or accelerators). The context parameters to be considered and optionally their *ranges* (e.g., minimum and maximum value) are declared in the PEPPHER interface descriptor. We refer to this considered subset of a call context instance's parameter and resource values shortly as a *context in-*

stance, which is thus a tuple of concrete values for context properties that might influence callee selection. Hence, composition maps context instances to implementation variants [12].

Composition can be done either statically or dynamically. *Static composition* constructs off-line a *dispatch function* that is evaluated at runtime for a context instance to return a function pointer to the expected best implementation variant [12]. *Dynamic composition* generates code that delegates the actual composition to a context-aware runtime system that records performance history and construct a dispatch mechanism on-line to be used and updated as the application proceeds.

Composition can even be done in multiple stages: First, static composition can narrow the set of candidates for the best implementation variant per context instance to a few ones that are registered with the context-aware runtime system that takes the final choice among these at runtime.

Dynamic composition is the default composition mechanism in PEPPHER. In the special case where sufficient meta-data for performance prediction is available for all selectable component variants, composition can be prepared completely statically and co-optimized with resource allocation and scheduling, thus bypassing the runtime system; see e.g. [11,12].

The *PEPPHER composition tool* [6] deploys the components and builds an executable PEPPHER application. It recursively explores all interfaces and components that (may) occur in the given PEPPHER application by browsing the interfaces and components repository.

The composition tool processes the set of interfaces (descriptors) bottom-up in reverse order of their components' *required interfaces* relation (lifted to the interface level) [12]. For each interface (descriptor) and its component implementations, the composition tool performs the following tasks:

1. It reads the descriptors and internally represents the metadata of all component implementations that match the target platform, expands generic interfaces and components, and generates platform-specific header files from the interface descriptor.
2. It looks up prediction data from the performance data repository or runs microbenchmarking code on the target platform, as specified in the components' performance meta-data.
3. It generates composition code in the form of *stubs* (proxy or wrapper functions) that will perform context-aware composition at runtime. If sufficient performance prediction metadata is available, it constructs performance data and dispatch tables for static composition by evaluating the performance prediction functions for selected context scenarios [11,12], which could be compacted by machine learning techniques [5]. Otherwise, the generated composition code contains calls to the PEPPHER run-time system to delegate variant selection to runtime, where the runtime system can access its recorded performance history to guide variant selection, in addition to other criteria such as operand data locality.

4. It calls the native compilers, as specified for each component, to produce a binary of every patched component source.

Finally, it links the application's main program and its compiled components together with the generated and compiled stubs, the PEPPHER library and the PEPPHER runtime system to obtain an executable program.

3 Adaptive off-line tuning

3.1 Motivation

Consider a typical example where a component's implementation variants for execution on different kinds of processors show performance advantages for different variants with respect to different input sizes, as shown in Figure 1. In a subrange of call context instance values (here, of the number of array elements to sort) where one implementation variant runs fastest among all implementations variants we call that implementation variant the *winner* for that range of input sizes.

We can map a n-dimensional range to a n-dimensional space. A specific context instance can also be considered as a point in a n-dimensional space. Some points or hyperplanes divide winning ranges of different implementations, we call those the transition points or hyperplanes. Ideally if all those points or hyperplanes can be found effectively, we can construct a compact representation which requires small overhead for both store and look-up, and it will provide 100 percent precision of winner prediction.

One may argue that the characteristics shown in Figure 1 may not apply for other problems. In this paper, we test three other benchmark applications, and these applications surprisingly conform to the characteristics of Figure 1, which shows an interesting property: The winning range for each implementation variant is convex, i.e., if two points on a one-dimensional space have the same winner, then it wins on all points between these. Our pruning strategy in this paper is based on this convexity assumption: for n-dimensional space, if all vertices of a space have the same winner, then it wins on all points in the space. Based on this assumption, we construct an algorithm and data structure to approximate and represent these transition points.

3.2 Hybrid static/dynamic composition with off-line training

Unlike static composition, dynamic composition can be guided by access to the run-time context for each invocation, and thus owns prerequisites for better selection precision at the cost of some run-time overheads. The hope is that the time saved by invoking the fastest implementation variant is larger than the overhead of the dynamic selection process, and thus portable performance is increased.

Dynamic composition with on-line training by the runtime system shows some disadvantages: it requires a certain number of representative executions

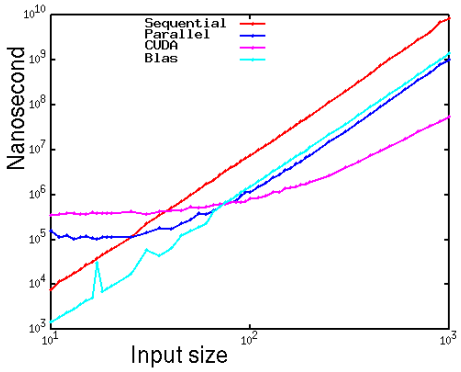

Fig. 1. Performance for matrix-matrix multiplication variants

before it can offer acceptable selection precision for dynamic composition; however, it is often not guaranteed that those representative executions will happen during a sufficiently long period of time. As an alternative, we consider off-line training and dynamic composition. In off-line training, measuring performance for every possible runtime context instance (which would offer perfect selection and precise representation of this information) is often not feasible, thus a dynamic composer is forced to make predictions based on a limited set of training examples.

The space $\mathcal{C} = I_1 \times ...I_D$ of context instances for a component with D attributes in the context instances is spanned by the D context attribute axes with considered (user-specified or default) finite intervals I_i of discrete values, for $i = 1, ..., D$. A continuous subinterval of an I_i is called a *range*, and any cross product of such subintervals on the D axes is called a *subspace* of $\mathcal{C}$. Hence, subspaces are "rectangular", i.e., subspace borders are orthogonal to the axes of $\mathcal{C}$.

In an experimental version of our composition tool, we offer a precision-controllable offline-trainer and dynamic composer based on ranges, i.e. it tries to automatically approximate the (usually, non-rectangular and possibly non-convex) subsets in $\mathcal{C}$ where one particular implementation variant performs better than all the others, by a set of subspaces.

Our idea is to find sufficiently precise approximations by adaptively recursive splitting of subspaces by splitting the intervals I_i, $i = 1, ..., D$. Hence, subspaces are organized in a hierarchical way (following the subspace inclusion relation) and represented by a 2^D-ary tree (cf. binary space partitioning trees and quadtrees/octrees etc.).

Our algorithm for off-line measurement starts from a trivial tree $T_{\mathcal{C}}$ that has just one node, the root (corresponding to the whole $\mathcal{C}$), which is linked to its 2^D corner points (here, the 2^D outer corners of $\mathcal{C}$) that are stored in a separate table of recorded performance measurements. The implementation variants of the component under examination are run with each of the corresponding 2^D context instances, possibly multiple times for averaging, using a context instance generator provided with the metadata of the component; a variant whose exe-

cution exceeds a timeout for a context instance are aborted and not considered further for that context instance. Now we know the winning implementation variant for each corner point and store it in the performance table, too, and $T_{\mathcal{C}}$ is properly initialized.

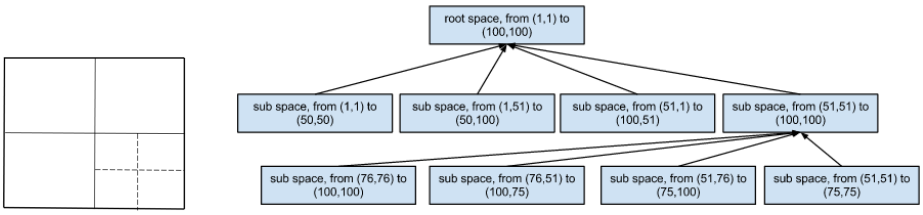

Fig. 2. Cutting a space recursively into subspaces, and the resulting dispatch tree.

Consider any leaf node v in the current tree T_t representing a subspace $S_v = R_1^v \times ... \times R_D^v$. If the same specific implementation variant runs fastest on all context instances corresponding to the 2^D corners of S_v, we stop further exploration of that subspace and will always select that implementation whenever a context instance at run-time falls within that subspace. Otherwise, the subspace S_v may be refined further. Accordingly, the tree is extended by creating new children below v which correspond to the newly created subspaces of S_v.

By iteratively splitting the ranges in FIFO order, we generate an adaptive tree structure to represent the performance data and selection choices, which we call *dispatch tree*.

The user can specify a *maximum depth* (training depth) for this iterative refinement of the dispatch tree, which implies an upper limit on the runtime lookup time, and also a maximum tree size (number of nodes) beyond which any further refinement is cut off. Third, the user may specify a timeout for overall training time, after which the dispatch tree is considered final.

Run-time lookup searches through the dispatch tree starting from the root and descending into subspace nodes according to the current runtime context instance. If the search ends at a *closed leaf*, i.e., a leaf node with equal winners on all corners of its subspace, the winning implementation variant can be looked up in the node. If the search ends in an *open leaf* with different winners on its borders (e.g., due to reaching the specified cut-off depth), we perform an approximation within that range by choosing the implementation that runs fastest on the subspace corner with the shortest Euclidean distance from the run-time context instance.

The deeper the algorithm explores the tree, the better precision the dynamic composer can offer for the composition choice; however, it requires more off-line training time and more runtime lookup overhead as well. We give the option to let the user decide the trade-off between training time and precision by setting the cut-off depth, size and time in the component interface descriptor.

3.3 Example for hybrid composition with adaptive off-line training

Let us consider a matrix-matrix multiplication example with two implementation variants, the well-known sequential version and a parallel version parallelized by pthreads with a fixed number of 4 threads. In the off-line training phase, performance data is measured by one execution per context instance; at execution time of the composed code with dynamic selection, performance is averaged over 10 runs per context instance.

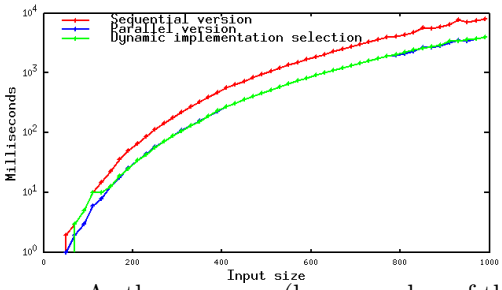

Fig. 3. Execution time for hybrid composition with a 41-node lookup tree determined by the adaptive refinement training algorithm with cut-off depth 3. — The hardware we use is a multicore system with 16 CPUs, where each CPU is an Intel(R) Xeon(R) CPU E5520 running at 2.27GHz with 8192 KB cache. The operating system is Linux 3.0-ARCH and the compiler is gcc 4.6.1.

As the resources (here, number of threads for OpenMP) was fixed, a context instance is just a triple consisting of the three problem sizes that define the operand matrix dimensions. The training space of context instances was chosen as $[1:1000, 1:1000, 1:1000]$, i.e., comprising 10^9 possible context instances (input sizes). As tree data structure we used an octree with simultaneous refinement of subspaces along all three dimensions. The cut-off depth for the tree was set to 3. With these settings, the off-line training time (i.e., for the tree construction including the measurements on the target system) takes 228 seconds and the constructed tree has 41 nodes, where the adaptive tree refinement is done mostly for subspaces with smaller problem sizes. By comparing the composed code at runtime with the actually fastest component for each context measured for square test matrices (see Figure 3), we find that the tree lookup yields a dynamic selection precision of 92%. From Figure 3 we can also see that the overhead for performing dynamic selection is rather negligible. For some context instance the dynamically selected implementation variant runs even faster than the same one without dynamic selection; such anomalies are mostly due to the operating system's interruptions during measuring; in principle, the composed code should always run slightly slower than the best individual component, due to run-time lookup overhead.

3.4 Selection

At initialization time, we read the dispatch tree from the file generated in the training phase, and add a translation table that maps each implementation variant's symbolic name to its function address.

The wrapper function generated by the composition tool selects the relevant parameter values from the call context and uses them to look up the dispatch

tree and thereby the right function address, which is filled in the descriptor for the task to be submitted to the runtime system. For open leaves, it chooses the winner of the corner that has the shortest Euclidean distance from the actual context instance.

4 Experimental results

Platform We use two GPU based heterogeneous systems called Fermi and Cora. A brief description of the two platforms is shown in Table 1.

Table 1. Platform description

Machine name	CPU cores	CPU type	GPUs	GPU type	OS	Compiler
Fermi	16	Intel(R) Xeon(R) CPU E5520 @ 2.27GHz	2	two Tesla M2050	3.2.1-2-ARCH	gcc 4.6.2 and nvcc V0.2.1221
Cora	16	Intel(R) Xeon(R) CPU X5550 @ 2.67GHz	3	two nVidia Tesla C2050 and one Tesla C1060	RHEL 5.6	gcc 4.1.2 and nvcc V0.2.1221

Benchmark For the evaluation we have chosen 4 benchmark problems: matrix-matrix multiplication, sorting, and two RODINIA benchmarks: path finder and backpropagation. A detailed description is shown in Table 2.

Table 2. Benchmark test settings

Benchmark	feature modeling	Range	space size	Implementation variants
Matrix-matrix multiplication	row size, column size of first matrix; column size of second matrix	(1, 1, 1) to (3000, 3000, 3000)	2.7E+10	Sequential implementation, CUDA implementation, Blas implementation, Pthread implementation
Sorting	array size; discretization of array values distribution (sampled number of inversions)	(1,0) to (100000,10)	1000000	bubble sort, insertion sort, merge sort, quick sort, CUDA thrust sort (only on Fermi)
Path finder	row; column	(1,1) to (10000,20000)	200000000	OMP implementation, CUDA implementation
Back propogation	array size	(1000) to (100000)	99000	OMP implementation, CUDA implementation

Methodology We first train each benchmark problem with training depth from 0 to 4. If the training time exceeds 3 hours then we terminate the training process. Each benchmark is trained twice, with one version which prunes closed space in the tree representation and another which performs no pruning at all.

The test points are chosen evenly from the training space so that every subspace in the dispatch tree is used for performance prediction.

Experimental results on two machines The test results for 4 benchmarks on Fermi are shown in Table 3. In particular, for backpropagation, the performance behavior for different training depths on Fermi are shown in Figure 4.

The results for the 4 benchmarks on Cora are shown in Tables 4.

Table 3. Test results for 4 benchmarks on Fermi (td: Training depth; tt: Training time; ato: average time overhead on dynamic selection; nn: Number of nodes generated in the tree representation)

Matrix-matrix multiplication on Fermi, 343 test points								
td	pruning closed space				no pruning for closed space			
	tt (s)	Precision (%)	ato (μs)	nn	tt (s)	Precision (%)	ato (μs)	nn
0	85	51	17	1	88	50	15.9	1
1	755	48	21	9	762	48	20.4	9
2	6118	62	23	73	6252	62	23	73
Sorting on Fermi, 110 test points								
	pruning closed space				no pruning for closed space			
0	233	36	4	1	233	36	3.6	1
1	1035	61	4.9	5	1035	64	4.9	5
2	2485	80	5.5	17	4071	80	5.6	21
Back propagation on Fermi, 20 test points								
	pruning closed space				no pruning for closed space			
0	7	55	9	1	6	55	9	1
1	7	80	11	3	6	80	10	3
2	8	90	13.6	5	8	90	11.8	7
3	7	95	12	7	13	95	12.5	15
4	8	100	13.1	9	18	100	14	31
Path finder on Fermi, 200 test points								
	pruning closed space				no pruning for closed space			
0	36	59	12.6	1	29	59	12.7	1
1	161	77	16.5	5	122	77	14.5	5
2	371	86	16.5	17	497	86	15.8	21
3	609	95	16.8	45	1992	95	20.9	85

Discussion The sorting, pathfinder and backpropagation benchmarks have shown a good result. The result for the matrix-matrix multiplication benchmark is a little disappointing, because it has a relatively large training space. Most subspaces in its tree are open ones and for the points near their corners the Euclidean distance criterion can give a better approximation while in the large central area of these subspaces, the precision can not be guaranteed. Since we train on a large space, which means large input sizes, a single training execution may take a long time; for this reason, training depths larger than 3 become not practical and not considered in this benchmark testing.

From the test results we can see that in most cases the precision of prediction of the winner implementations increases with the depth of the dispatch tree. This is expected because, as open subspaces can be partly closed by exploring deeper levels, the precision increases. This trade-off is exposed to the users.

We also can see that for a relatively short training time, we get a reasonable prediction precision in total which means pruning closed subspaces works and the assumption that we can treat all points in a closed subspace equally holds for those benchmarks. Another evidence for the assumption is the comparison between two version of test results, one which perform closed subspace pruning and one which does not. We get almost the same results from the two set of tests on all benchmarks we use, thus it is safe not to explore closed space in the training phase.

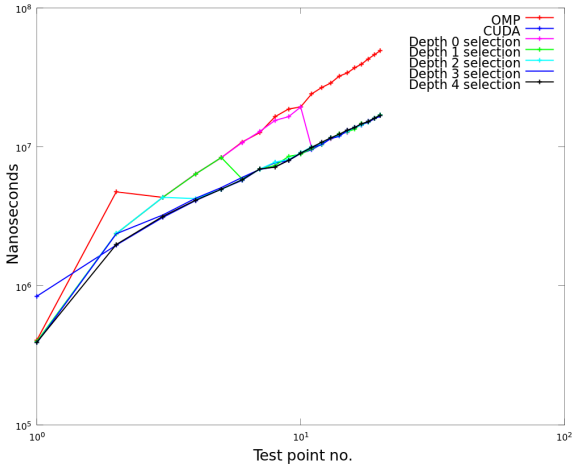

Fig. 4. Performance with maximum depths 0 to 4 for the backpropagation benchmark on Fermi.

The time overhead for run-time selection is acceptable, on the level of microseconds. Since we only explore a shallow depth of a dispatch tree, the number of nodes generated is small, too, so the memory overhead is acceptable as well.

As for the relation between precision and performance, we can illustrate it in Figure 4 for backpropagation. Comparing constant invocation of the OpenMP implementation variant with dynamic selection among all available variants, we see that for subspaces where the OpenMP variant wins, the performance of all variants only differs by a few microseconds; for the subspace where OpenMP does not win, we gain performance. The performance gained might be remarkable if some variant scaling badly is constantly invoked. From the figures we can also see that wrong decisions for points within open subspaces often happen near transition points between different winners, and often the performance difference of implementation variants at points near transition points is low, thus a wrong decision does not yield a performance penalty as large as in other points in the subspace.

In general, our approach can pick the best implementation variant for most of the cases for the different platforms.

We observed an anomaly for exploration of subspace in matrix-matrix multiplication on Fermi. When the depth increases from 0 to 1, with more training time, the precision drops. One possible explanation is that when splitting some space where the winner on one of the corners is shared by a minority of the other corners, the Euclidean distance criterion will cause a majority of points to be predicted wrongly, which with the coarser dispatch tree are predicted correctly. Continuing to refine that subspace may make the precision increase again; however, continuing the exploration for matrix-matrix multiplication on such large space is so time-consuming that we have to postpone further investigation of this problem to future work.

Table 4. Test results for 4 benchmarks on Cora

td	tt (s)	Precision (%)	ato (μs)	nn	tt (s)	Precision (%)	ato (μs)	nn
Matrix-matrix multiplication on Cora, 343 test points								
	pruning closed space				no pruning for closed space			
0	67	48	17.6	1	63	48	18.3	1
1	634	49	22.5	9	621	49	22.2	9
2	5115	67	26.4	73	5009	68	26.2	73
Sorting on Cora, 110 test points								
	pruning closed space				no pruning for closed space			
0	162	34	5.3	1	159	35	5.5	1
1	714	62	7.1	5	710	62	8.5	5
2	1747	80	8.6	17	2809	78	8.6	21
Back propagation on Cora, 20 test points								
	pruning closed space				no pruning for closed space			
0	3	55	11.9	1	3	60	13	1
1	4	85	12.6	3	4	90	14.7	3
2	4	95	16.1	5	5	95	14	7
3	4	100	13.4	7	7	95	15.2	15
Path finder on Cora, 200 test points								
	pruning closed space				no pruning for closed space			
0	21	39	12.5	1	21	39	12.1	1
1	97	67	14.5	5	92	67	16.1	5
2	219	82	15.6	17	400	82	16.2	21
3	367	95	15.9	45	1511	95	18.1	85

5 Related work

Techniques for automated performance tuning have been considered extensively in previous work; they are applied e.g. in generators of optimized domain-specific libraries (such as basic linear algebra [25,14,21], reduction [26], sorting [15,21] or signal transforms [8,20,17,7]), iterative compilation frameworks (e.g. [16]), or for the optimized composition of general program units [11,13,2,1,24], e.g. the components in our case.

Automated performance tuning usually involves three fundamental preparatory tasks: (1) search through the space of context property values, (2) generation of training data and measurements on the target system, (3) learning a decision function / rule (e.g. for best variant selection, decomposition, or settings for tunable parameters), or alternatively (3a) learning a predictor for performance and then (3b) decide / optimize based on that predictor among the remaining options. In our approach, these three tasks are tightly coupled to limit the amount of measurement time and representation size required, while most other approaches decouple at least two of these tasks.

Search, measurements and learning can each be performed off-line (i.e., at deployment time or compile time) or on-line (i.e., at run time), or as a combination of both. In our approach, all tasks are done off-line at component deployment time, while all are performed at runtime in the StarPU runtime system by continuously recording measurements from the running program and using these data for future decisions [2].

Kessler and Löwe [11] propose a methodology for optimized composition of grey-box components. The component provider offers additional knowledge such as time functions for performance prediction, which might include data obtained

from microbenchmarking, measuring, direct prediction or hybrid prediction. Predictions are made for a regularly sampled (dense) space of context instances, including composition of prediction functions for recursive components in a dynamic programming algorithm. Based on those predictions, a dispatch table and dynamic selection code are generated and injected into the components for runtime selection. The dispatch tables can be a-posteriori compressed using various machine learning techniques such as decision tree, decision graph, Bayesian classifier and SVM, where the decision tree was empirially found to be most effective [5]. In contrast, our current work does the compression a-priori, thus avoiding excessive prediction or measurements.

PetaBricks [1] provides a framework with language and compiler support for exposing implementation variant choices. It also contains an off-line autotuning system which starts to test with a small input size and doubles the size of the input on each later iteration. They assume that optimal choices for smaller subproblems are independent of the larger problem, so they construct new composition candidates incrementally from small input sizes to larger ones. The algorithmic choices are made off-line in the output of the compiler.

Elastic computing [24] lets the programmer transparently utilize the heterogeneous computing resources by providing a library of elastic functions. The autotuner trains itself from measurements (which are not further specified) and then uses a linear regression model for predicting performance of untested input values.

Grewe and O'Boyle [9] suggested an approach for statically choosing the best mapping between tasks and unit types (CPU, GPU). Static features such as numbers of float operations, are extracted from a set of programs, and and scheduling decisions are fed to a SVM classifer. Then at compile time, the decision for distribution of work load on different kinds of processors is made.

ABCLibScript[10] is a directive system that provides autotuning functionality on numerical computations within the FIBER framework. The choice of performance-related parameters, such as unrolling depth and block length, is specified for training execution. A performance model is also specified by the users, and generated together with training results. At run-time, best code regions are selected.

Danylenko et al. [5] compares 4 different machine-learning approaches, Decision Trees, Decision Diagrams, Naive Bayes and SVM on sorting benchmark in the field of context-aware composition for a-posteriori compression of the dispatch function. Results show Decision Diagram performs better in scalability, and almost the same in prediction accuracy and decision overhead comparing with other 3 approaches.

Singer and Veloso [19] applied a back-propagation neural netork for performance prediction in the field of signal processing. Results show that choices of different combination of features affect remarkably the prediction precisions.

[22] presents an unsupervised learning approach (fuzzy clustering algorithm) for a machine learning based compiler. Significant reduction in the training cost is achieved by grouping training programs into clusters using 'ratio of assembly

instructions to the total program instructions' as a feature vector. After clustering, they carried training executions on one (randomly selected) representative from each cluster, recording the best execution configuration for each of the selected programs. This is an alternative approach to reduce training time.

In [23], Wang and O'Boyle developed two predictor functions (data-sensitive and data-insensitive) to predict the best OpenMP execution configuration (number of OpenMP threads, scheduling policy) for an OpenMP program on a given architecture. They use two machine learning algorithms (Artificial Neural Network and Support Vector Machine) and train them using code, data and runtime features extracted via source to source instrumentation.

Our approach can be considered as an adaptive variant of decision tree learning. Decision tree learning, often based on C4.5 [18] or similar tools, is also used in many other approaches, e.g. in [20,21,26,7]. A direct comparison of our learning algorithm with C4.5 and other learning methods is planned for future work.

6 Conclusions and Future Work

We have developed an adaptive off-line training algorithm and dispatch tree representation that allows to pick the best implementation variants for most of the cases on different GPU-based heterogeneous machines, hence it improves performance portability. Our method allows to reduce training time and enables the user to trade off prediction precision, runtime overhead and training time.

Our approach for pruning closed space is based on the assumption that, if corners of a space show a common winner, all points in the space would have the same winner, which holds in most of our benchmark applications. The assumption needs to be further investigated with more applications, and refined prediction methods for open spaces should be developed. Note that, in cases where the user knows that the assumption does not hold, a better accuracy could then be enforced by also refining closed space within the given depth limit, at the expense of a larger dispatch tree and longer training time.

Further improvements of our method are possible and will be considered in future work. For instance, timeouts for individual measurements (applicable on CPUs) and aborting variants under measurement that exceed the current winner of a training point can save more training time. Also, the user may accept a tolerance such that even suboptimal variants not slower than the winner by that tolerance could also be considered winners in order to close spaces earlier.

Acknowledgments This work was partly funded by EU FP7 project PEPPHER (www.peppher.eu) and by SeRC. We also thank the Scientific Computing group at the University of Vienna, Austria, for letting us use their GPU server *Cora* on which some of the measurements reported in this paper were taken.

References

1. Jason Ansel, Cy P. Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman P. Amarasinghe. PetaBricks: A language and compiler for algo-

rithmic choice. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009*, pages 38–49. ACM, 2009.

2. Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic calibration of performance models on heterogeneous multicore architectures. In *Euro-Par Workshops 2009 (HPPC 2009)*, volume 6043 of *Lecture Notes in Computer Science*, pages 56–65. Springer, August 2009.
3. Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.
4. Siegfried Benkner, Sabri Pllana, Jesper Larsson Träff, Philippas Tsigas, Uwe Dolinsky, Cedric Augonnet, Beverly Bachmayer, Christoph Kessler, David Moloney, and Vitaly Osipov. PEPPHER: Efficient and productive usage of hybrid computing systems. *IEEE Micro*, 31(5):28–41, 2011.
5. Antonina Danylenko, Christoph Kessler, and Welf Löwe. Comparing machine learning approaches for context-aware composition. In *Proc. 10th Int. Conference on Software Composition (SC-2011), Zürich, Switzerland*, volume 6703 of *Lecture Notes in Computer Science*, pages 18–33. Springer, June 2011.
6. Usman Dastgeer, Lu Li, and Christoph Kessler. Performance-aware dynamic composition of applications for heterogeneous multicore systems with the PEPPHER composition tool. In *Proc. 16th Int. Workshop on Compilers for Parallel Computers (CPC'12), Padova, Italy*, January 2012.
7. Frédéric de Mesmay, Yevgen Voronenko, and Markus Püschel. Offline library adaptation using automatically generated heuristics. In *Int. Parallel and Distr. Processing Symp. (IPDPS'10)*, pages 1–10, 2010.
8. Matteo Frigo and Steven G. Johnsson. Fftw: An adaptive software architecture for the FFT. In *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, May 1998.
9. Dominik Grewe and Michael F. P. O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *Proc. 20th int. conf. on Compiler construction*, CC'11/ETAPS'11, pages 286–305. Springer-Verlag, 2011.
10. Takahiro Katagiri, Kenji Kise, Hiroki Honda, and Toshitsugu Yuba. Abclibscript: a directive to support specification of an auto-tuning facility for numerical software. *Parallel Computing*, 32(1):92–112, 2006.
11. Christoph W. Kessler and Welf Löwe. A framework for performance-aware composition of explicitly parallel components. In *Parallel Computing: Architectures, Algorithms and Applications, (ParCo 2007)*, volume 15 of *Advances in Parallel Computing*, pages 227–234. IOS Press, 2007.
12. Christoph W. Kessler and Welf Löwe. Optimized composition of performance-aware parallel components. In *Proc. 15th Int. Workshop on Compilers for Parallel Computers (CPC-2010)*, July 2010.
13. Christoph W. Kessler and Welf Löwe. Optimized composition of performance-aware parallel components. *Concurrency and Computation: Practice and Experience*, 24(5):481–498, April 2012. Published online in Wiley Online Library, DOI: 10.1002/cpe.1844, sep. 2011.
14. Xiaoming Li and Maria J. Garzarán. Optimizing matrix multiplication with a classifier learning system. In *Proc. workshop on Languages and Compilers for Parallel Computing (LCPC'05)*, pages 121–135, 2005.

15. Xiaoming Li, Maria J. Garzarán, and David Padua. A dynamically tuned sorting library. In *Proc. ACM Symp. on Code Generation and Optimization (CGO'04)*, pages 111–124, 2004.
16. Eunjung Park, Sameer Kulkarni, and John Cavazos. An evaluation of different modeling techniques for iterative compilation. In *Proc. Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES'11)*, October 2011.
17. Markus Püschel, Jose M. F. Moura, Jeremy R. Johnson, David Padua, Manuela M. Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicolas Rizzolo. Spiral: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2), February 2005.
18. J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
19. Bryan Singer and Manuela Veloso. Learning to predict performance from formula modeling and training data. In *Proc. 17th Int. Conf. on Machine Learning*, pages 887–894, 2000.
20. Bryan Singer and Manuela Veloso. Learning to construct fast signal processing implementations. *Journal of Machine Learning Research*, 3:887–919, December 2002.
21. Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 277–288. ACM, 2005.
22. John Thomson, Michael F. P. O'Boyle, Grigori Fursin, and Björn Franke. Reducing training time in a one-shot machine learning-based compiler. In Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li, editors, *LCPC*, volume 5898 of *Lecture Notes in Computer Science*, pages 399–407. Springer, 2009.
23. Zheng Wang and Michael F.P. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. *SIGPLAN Not.*, 44(4):75–84, February 2009.
24. John Robert Wernsing and Greg Stitt. Elastic computing: A framework for transparent, portable, and adaptive multi-core heterogeneous computing. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems (LCTES)*, pages 115–124. ACM, 2010.
25. R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.
26. Hao Yu and Lawrence Rauchwerger. An adaptive algorithm selection framework for reduction parallelization. *IEEE Trans. on Par. and Distr. Syst.*, 17(10):1084–1096, October 2006.