Usman Dastgeer, Christoph Kessler. Conditional component composition for GPU-based systems. Accepted at MULTIPROG-2014 workshop at HiPEAC-2014, Vienna, Austria, Jan. 2014.

Conditional component composition for GPU-based systems

Usman Dastgeer and Christoph W. Kessler

IDA, Linköping University, 58183 Linköping, Sweden.
<firstname>.<lastname>@liu.se

Abstract. User-level components can expose multiple functionally equivalent implementations with different resource requirements and performance characteristics. A composition framework can then choose a suitable implementation for each component invocation guided by an objective function (execution time, energy etc.). In this paper, we describe the idea of *conditional composition* which enables the component writer to specify constraints on the selectability of a given component implementation based on information about the target system and component call properties. By incorporating such information, more informed and user-guided composition decisions can be made and thus more efficient code be generated, as shown with an example scenario for a GPU-based system.

1 Introduction

A software component consists of a description of a computational functionality along with one or more implementations (sometimes called *implementation variants* or simply *variants*) of that functionality. These implementations are considered functionally equivalent and can be used interchangeably when doing the computation. Implementations can come from various sources; from some standard library, from an expert programmer [1] or automatically generated by a tool by instantiating values for tunable parameters. The problem of selecting an appropriate implementation for each component invocation in the program is often referred to as the *component composition* (or *implementation selection*) problem. Normally it is guided by some optimization objective function such as execution time as we assume in this paper.

In GPU based systems supporting general purpose computations on GPU devices with CUDA/OpenCL, the implementations can be device/architecture specific. Considering this and the distributed memory address space in these systems, component composition is tightly coupled with resource allocation (current system workload) and data locality in these systems. Moreover, implementations can be written/optimized for certain execution scenarios and using them otherwise may result in a sub-optimal (or even worse, incorrect) execution. The implementation writers should be able to specify such constraints about the conditions on selectability of implementations they write. The composition decisions can

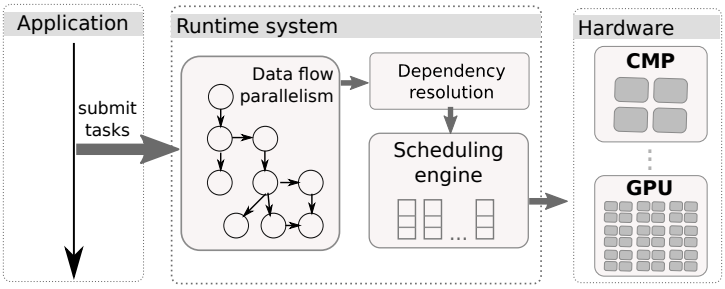


Fig. 1. Overview of an application execution using a task-based runtime system.

then be guided by such conditions/constraints (i.e., conditional composition), evaluated either before or during an actual program execution.

In this paper, we briefly discuss the component composition problem for GPU-based systems and how it can be carried out in multiple stages. We describe the idea of conditional composition and how it can provide flexible and powerful composition capabilities for the programmer in an intuitive manner. Specifically, we make the following contributions:

- We present the idea of conditional component composition for GPU-based systems and how it can be used to improve composition decisions made by the runtime system.
- We describe different information sources and how they can be queried in a portable and generic manner. Our composition tool prototype has been extended to provide a powerful interface for lookup of application and platform properties and to later use that information to affect composition decisions made by the runtime system.
- We present a case-study where the state of operand data alongside information about the underlying software platform is used to guide the composition decisions made by the runtime system.

The paper is organized as follows: Section 2 describes component composition whereas conditional composition is described in Section 3 followed by implementation details in Section 4. Section 5 describes the benefits of conditional composition with an example scenario followed by related work in Section 6; Section 7 concludes.

2 Component composition

Composition is selection of an implementation among potentially many functionally equivalent implementations for each component invocation to either minimize or maximize some objective function. For the discussion in this paper, the objective function to be optimized is execution time, but it could also be energy consumption etc. Normally, greedy composition decisions are made considering one component invocation at a time for practical reasons [8–12]; however, the

ideas described here are equally applicable for global composition where the objective function is maximized or minimized across multiple component calls[1].

In general, the composition decisions can be made statically and/or dynamically as described below.

2.1 Static (offline) composition

Static composition decisions are made offline (i.e., before the actual program execution starts). These decisions can be made e.g., at compilation time or by some preprocessor tool based on statically available information about the target system resources/topology. For example, if GPU is not available on the target system, all component implementations targeting the GPU can be disabled at compilation time. Pruning implementation choices offline can significantly reduce the dynamic decision making overhead. However, the information for making a decision at this time is often quite limited and preliminary. In few cases, where information about the problem instance (data sizes etc.) and performance models for those problem instance is available before execution, composition decisions can be made completely offline. However, in most cases, the composition decisions are made online during the actual program execution, based on performance models (e.g. decision tree) or dispatch tables [13] constructed either offline or made available from previous program executions.

2.2 Dynamic (online) composition

Dynamic composition decisions happen during the actual program execution. Internally, they can happen as frequently as once for each component invocation or just once during the whole program execution (at program initialization or first invocation).

The state of the art techniques for component composition on GPU-based systems (e.g., StarPU [12], UnMP [10], Nanos++ [11]) rely on dynamic scheduling where the composition decisions are made at runtime (i.e., at invocation) by considering performance characteristics of each implementation, along with operand data locality and current system workload. A common dynamic scheduling heuristic implemented by many such approaches is HEFT (Heterogeneous Earliest Finish Time) [14]. Although a greedy heuristic, it performs well in practice and is shown to work with multiple applications.

Most runtime approaches make the decision once per component invocation. At this time, information about the current operand data, system workload and operand data locality is available besides the information sources available from earlier stages. However, it is the stage most sensitive to the decision overhead as the overhead is incurred for each component invocation, i.e., the potential advantage of making better decisions can be superseded by the overhead of decision making. In many cases, the potential advantage of simultaneous executions

[1] In global composition, the optimal (with respect to the objective function) choice at an individual component invocation may result in an overall sub-optimal decision and vice-versa [16].

for independent computations as well as overlapping communication with computation amortizes or hides the decision overhead. Most runtime approaches implement dynamic composition as shown in Figure 1. Component invocations are submitted as runtime tasks with multiple implementations to the runtime systems. After data dependencies for a given task's operand data are satisfied, it is scheduled by some greedy scheduling heuristic such as HEFT [14].

3 Conditional composition

An implementation of a component can be specialized for some execution context such that using it in a different execution context may yield poor performance or even worse, can give incorrect results. By enabling component/implementation writers to specify constraints on selectability for the component implementations they write, more effective and powerful composition decisions can be made. These constraints can be specified based on the following information sources:

- *system resources and topology:* information about number/type/frequency of processing cores, NUMA organization, cache types/sizes, memory organization/sizes, interconnect type/capacity etc.
- *software platform:* information about availability of particular software library or a certain version of a library, device drivers and their versions, compilers and their versions etc.
- *system runtime characteristics:* information about current CPU/GPU utilization, application memory footprint, cache hit/miss ratio, effective bandwidth etc.
- *component call properties:* information about operand data sizes, actual contents of operand data (sparsity, sortedness) etc.

The first two information sources (i.e. information about system resources and software platform) are known, in many cases before the actual program execution starts. The information about the system runtime characteristics can be collected and made available by a runtime system during the actual program execution; either by monitoring system counters, or by employing application monitoring mechanisms. For example, the StarPU runtime system provides certain online performance monitoring mechanisms that can provide profiling information about individual tasks and registered operand data. Certain information about CPU and GPU workers can be queried such as the amount of time a worker spent in actual work compared to sleeping time etc. More powerful and high-level metrics about workers' availability can be built based on such information sources. Besides the system runtime characteristics, information about the actual component call is available for decision making at the component invocation time. This mainly includes access to the actual operand data which can be used to make the decisions. This does not require any specific mechanism in the runtime system and can easily be exposed for conditional composition.

The above information sources can be used by the component implementation writer for specifying hints or constraints on selectability of his implementations. However, conditional composition requires a mechanism for both specification and usage of that information, as described below.

Portable specification of selectability constraints: For systematic decision making in a portable manner, information about system resources and software platforms should be made available via a standard interface so that the component writer can write constraints for a component implementation that can work across different systems. Underneath that interface, the information about system resources and software platforms can be collected e.g., via system-/OS-specific routines or by doing micro-benchmarking. Libraries such as hwloc [6] implement a mechanism to look up such information across different system/OS configurations. One example of providing a generic interface for accessing information about the underlying hardware and software platform is the PEPPHER Platform Description Language (PDL) [15]. Similarly, the runtime system should expose a generic interface to lookup and "refer to" information about runtime system characteristics as well as component call properties.

Mechanisms to guide composition decisions by such constraints: A preprocessing tool can process the constraints specified by an implementation writer based on the information available offline (i.e., about system topology and software platform) to enable or disable selection of certain component implementations. Although processing such constraints at runtime is also possible, it is undesirable because of the runtime overhead. The constraints regarding runtime and call properties need to be resolved at the runtime, by a runtime system when doing the actual call invocation.

4 Design and Implementation

In our earlier work [7], we have implemented a composition tool (as a preprocessor) that can parse component implementations along their meta-data and can make static composition decisions while generating optimized code for dynamic composition with the StarPU [12] runtime system. Figure 2 shows how the developed prototype can be used to compose an application starting from the legacy code to the final code executable with the runtime system. Recently, we have implemented support for PDL [15] in the tool that allows modeling system properties in an XML form. The tool can parse information present in the PDL XML file and can serialize the information in the form of key-value properties. The code is then generated to load this information at program initialization time into internal data structures and provide a standard C++ API for efficient lookup of this information in the program source code. Besides other things, this PDL information can be used for conditional composition. For example, a CUDA implementation may require availability of at least 16 streaming multiprocessors for execution as specified below:

```
validIf(pdl::getIntProperty("numCudaSM") >= 16)
```

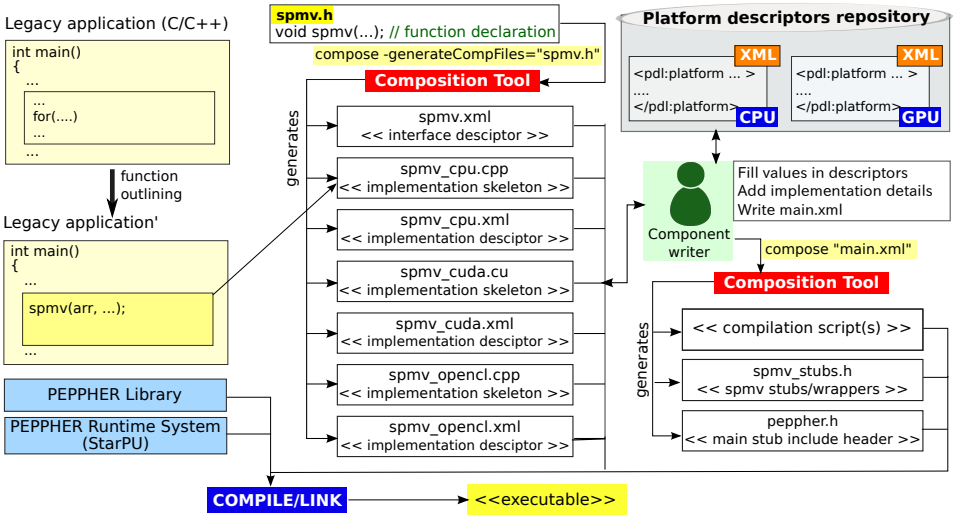


Fig. 2. Overview of the composition tool and how it could be used to compose an existing application. The tool can generate skeletons of XML descriptor files from a given function declaration in header file. The component writer can then fill in missing information in the generated descriptors' skeletons, and can provide actual component implementations including specification of conditions (if any) on their selection. In the end, the composition tool can generate code for execution with the StarPU runtime system.

Multiple constraints can be appended together using operators (AND, OR etc.) available in C/C++[2]. Furthermore, the tool can generate code for resolving constraints at runtime based on information about actual operand values. The tool currently allows the implementation writer to specify constraints on selection of an implementation based on system properties made available via a C++ API as well as information about the actual operand data. As shown in the next section with an example scenario, many interesting composition decisions can be made with the help of the conditional composition capability.

5 Evaluation

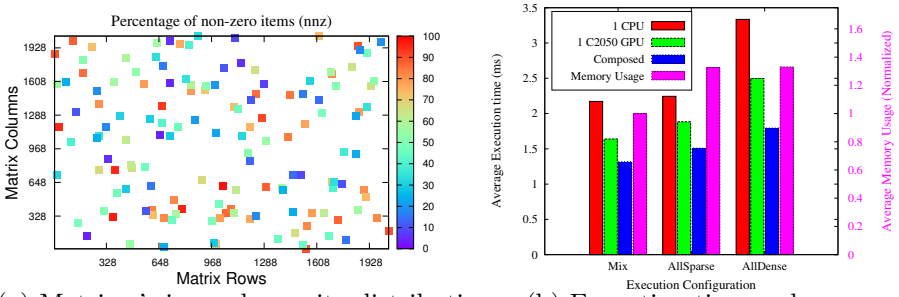
The conditional composition implemented in the current prototype enables conditional composition at runtime. To demonstrate conditional composition, we consider a matrix-vector multiplication (`mvmult`) computation with the following signature:

```
template <typename T>
void mvmult(GMatrix<T> &A, T *x, T *y);
```

[2] As XML restricts usage of certain special characters, we encode them. For example, we represent the logical AND operator && using `.AND.`

Table 1. Implementations for `mvmult` component.

Name	For	Valid If
mvmult_cpu_atlas	CPU	A.isDenseMatrix() .AND. pdl::getBoolProperty("atlasBlas")
mvmult_cpu_goto	CPU	A.isDenseMatrix() .AND. pdl::getBoolProperty("gotoBlas")
mvmult_cpu_csr	CPU	A.isSparseMatrix()
mvmult_cuda_cublas	CUDA	A.isDenseMatrix() .AND. pdl::getBoolProperty("cuBlas")
mvmult_cuda_cusparse	CUDA	A.isSparseMatrix() .AND. pdl::getBoolProperty("cuSparse")



(a) Matrices' size and sparsity distribution. (b) Execution time and memory usage.

Fig. 3. Conditional composition of `mvmult` computation. (a) shows distribution of matrices over number of rows, columns and non-zero elements whereas (b) shows execution time (ms), averaged over `mvmult` execution for all matrices. It shows execution on a CPU, a C2050 GPU and composed execution where either the CPU or GPU is used for each `mvmult` call (dynamic performance aware scheduling). Furthermore, it shows memory usage (normalised) for different execution configurations.

The input matrix operand is passed using a generic matrix container that can internally store the matrix data in either dense or CSR (Compressed Sparse Row) sparse format. The container is parameterized on which format to use for a given operand matrix. The `mvmult` component can thus do matrix-vector multiplication for matrices with different numbers of non-zero elements. However, depending upon whether the matrix is stored in sparse or dense format, different implementations can run faster. Table 1 lists the implementations that we have devised for this component. Most of them are simply created by wrapping optimised library functions available for this computation (e.g. AtlasBLAS and GotoBLAS for `sgemv` on CPU). The table also lists the constraints on selectability of each implementation. These constraints are based upon information about the software platform as well as information about storage of operand data.

For evaluation, we generate 200 matrices, randomized over the number of rows, columns and non-zero elements (`nnz`). Figure 3(a) shows the distribution

of randomly generated matrices based on number of rows, columns and non-zero elements. The execution times, as shown in Figure 3(b), are taken with the following three configurations:

1. *AllSparse* when all matrices are stored in sparse format so that implementations written for sparse `sgemv` get selected always.
2. *AllDense* when all matrices are stored in dense format so that implementations written for dense `sgemv` get selected always.
3. *Mix* when all matrices are stored in sparse format except those having more than 50% non-zero elements;[3] they are stored in dense format.

These different configurations are orthogonal to the actual component and its implementations as they are not modified in this process. However, these configurations do affect the selection of implementations used for `mvmult` computation. For example, when a matrix is stored in the sparse format, all implementations with `A.isDenseMatrix()` condition cannot be selected and vice versa.

As shown in the figure, the *Mix* format performs better, on average, than any other configuration considering both execution time and memory usage.

6 Related work

The notion of function interfaces in the Elastic Computing framework [5] provides a mechanism to specify simple constraints for each implementation based on operand data state (e.g. sorted or random array contents are specified as `sort_sorted` and `sort_random` respectively for a `sort` component). However, the specification mechanisms are naïve and cannot model more complex information sources. PetaBricks [3] allows the programmer to specify rules in a data-flow manner to specify mappings from inputs to outputs which allows it to explore multiple pathways while keeping the program execution consistent with the specified data-flow constraints. It relies on an auto-tuning compiler to actually make the decisions and find suitability of rules to the system- and problem-specific properties. Merge [4] provides predicate annotations to constrain selection of an implementation based on structure and size of input data as well as the target architecture type.

Our framework provides a more explicit and exhaustive set of information sources accessible for the component writer to specify such constraints. For example, we provide notions for runtime information about system workload and data locality that could be important when making the composition decision. Moreover, all these above mentioned frameworks (Elastic, PetaBricks, Merge) propose a new unified programming language/API for the component writer whereas we rely on existing well-established programming models (OpenMP, CUDA etc.) for component implementations.

There exists a large body of work in Grid computing environments about component models [17–19] and resource management [22–26]. Providing quality

[3] The 50% threshold is chosen arbitrarily in this case as focus is on demonstrating conditional selection rather than finding the optimum matrix sparsity threshold.

guarantees for executing jobs/applications is a tricky problem in grid computing considering the underlying complex, heterogeneous, dynamically fluctuating, conditionally available and geographically spread hardware (computation and storage) resources. Normally this is done by a Resource Management System (RMS) which do matchmaking between the application/job demands [21] and the capabilities advertised by the resources. The matchmaking is done by directly comparing values of attributes advertised by the resources with those required by different jobs. In [28], Tangmunarunkit et al. proposed a more complex matching technique using semantic web technologies.

The resource management in grid computing is done for different jobs which are normally independent of each other. In our case, composition decisions are made for component computations inside an application with arbitrary data and control dependency. Moreover, the decision in grid computing is mainly about choosing where to run a job considering its resources requirements whereas we deal with choosing a particular component implementation for execution, from a set of implementations, based on the constraints specified on their selectability.

7 Conclusion

The composition problem is becoming increasingly important as more and more computations are getting multiple implementations available to choose from; these implementations differ in their resource and platform requirements, context dependencies and performance behaviour. Conditional composition provides a powerful mechanism for a component writer to affect/control the decision making regarding selectability of a certain implementation in a given execution context. The proposed framework allows component writers to specify conditions on selectability of component implementations by providing standard APIs for accessing both static and dynamic information sources.

Acknowledgements: Partly funded by the EU FP7 projects PEPPHER (www.peppher.eu) and EXCESS (excess-project.eu) and by SeRC.

References

1. K. Asanovic et al. *A view of the parallel computing landscape.* Comm. ACM 52(10), pp 56–67, 2009.
2. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA. 1990.
3. J. Ansel et al. *PetaBricks: a language and compiler for algorithmic choice.* In Proc. ACM SIGPLAN conference on Programming language design and implementation (PLDI '09). ACM, New York, NY, USA, 2009.
4. M. D. Linderman et al. *Merge: A programming model for heterogeneous multi-core systems.* In Proc. 13th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, (ASPLOS 2008). ACM, 2008.
5. J. R. Wernsing and G. Stitt. *Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing.* SIGPLAN Not. Vol. 45(4), 2010.

6. F. Broquedis et al. *hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications.* Proc. 18th Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP), pp 180–186, 2010.
7. U. Dastgeer et al. *The PEPPHER Composition Tool: Per- formance-aware dynamic composition of applications for GPU-based systems.* Proc. Int. Worksh. on Multi-core Com- puting Systems (MuCoCoS'12), Salt Lake City, USA, 2012.
8. M. Kicherer et al. *Seamlessly portable applications: Mana- ging the diversity of modern heterogeneous systems.* ACM Trans. Archit. Code Optim., Vol. 8(4), pp 42:1–42:20, 2012.
9. M. Kicherer et al. *Cost-aware function migration in heterogeneous systems.* In Proc. Int. Conf. on High Perf. and Emb. Arch. and Comp. (HiPEAC'11). ACM, NY, USA, 2011.
10. A. Podobas, M. Brorsson and V. Vlassov. *Exploring heterogeneous scheduling using the task-centric programming model.* Int. Worksh. on Algorithms, Models and Tools for Parallel Computing on Heterog. Platforms (HeteroPAR'12), Euro-Par: Parallel Processing Workshops, 2012.
11. J. Planas et al. *Selection of Task Implementations in the Nanos++ Runtime.* PRACE WP53, 2013.
12. C. Augonnet et al. *StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures.* Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009, Vol. 23, pp 187–198, 2011.
13. C. W. Kessler and W. Löwe. *Optimized composition of performance-aware parallel components.* Concurr. and Comp.: Practice and Exp., Vol. 24(5), pp 481–498, 2012.
14. H. Topcuoglu et al. *Performance-Effective and Low-Com- plexity Task Scheduling for Heterogeneous Computing.* IEEE Trans. on Parallel and Dist. Sys., Vol. 13(3), 2002.
15. M. Sandrieser, S. Benkner and S. Pllana. *Using explicit platform descriptions to support programming of heterogeneous many- core systems.* Parallel Computing 38(1-2), pp 52–65, 2012.
16. U. Dastgeer and C. Kessler. *A Framework for Performance-aware Composition of Applications for GPU-based Systems.* 6th Int. Worksh. on Par. Prog. Models and Sys. Soft. for High-End Comp. (P2S2), ICPP, Lyon, 2013.
17. M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon and R. Bramley. *XCAT 2.0: A Component-Based Programming Model for Grid Web Services.* Technical report, 2002.
18. F. Bertrand and R. Bramley. *DCA: a distributed CCA framework based on MPI.* Proc. Ninth Int. Workshop on High-Level Parallel Programming Models and Supportive Environments, pp 80–89, 2004.
19. P. Caruso, G. Laccetti and Marco Lapegna. *A Performance Contract System in a Grid Enabling, Component Based Programming Environment.* Advances in Grid Computing (EGC), LNCS, vol. 3470, pp 982–992, 2005.
20. W. Smith, I Foster and V. Taylor. *Predicting application run times using historical information.* Proc. Of the IPPS/SPDP' 98 workshop on job scheduling strategies for parallel processing, 1998.
21. F.Vraalsen, R. Aydt, C. Mendes and D. Reed. *Performance contracts: predicting and monitoring application behaviour.* Proc. IEEE/ACM Second Int. Workshop on Grid Computing, LNCS, vol. 2242, pp 154–165, 2001.

22. M. Parashar, Z. Li, H. Liu, V. Matossian and C. Schmidt. *Enabling Autonomic Grid Applications: Requirements, Models and Infrastructure*. Self-star Properties in Complex Information Systems, LNCS, vol. 3460, pp 273–290, 2005.
23. M. Agarwal and M. Parashar. *Enabling autonomic compositions in grid environments*. Proc. Fourth Int. Workshop on Grid Computing, pp 34–41, 2003.
24. M. Parashar and J.C. Browne. *Conceptual and Implementation Models for the Grid*. Proc. of the IEEE, vol. 93 (3), pp 653–668, 2005.
25. K. Krauter, R. Buyya and M. Maheswaran. *A taxonomy and survey of grid resource management systems for distributed computing*. Softw. Pract. Exper. 32(2), pp 135–164, 2002.
26. Y.S. Kee, D. Logothetis, R. Huang and H. Casanova. *Chien, A., "Efficient resource description and high quality selection for virtual grids*. IEEE Int. Symposium on Cluster Computing and the Grid (CCGrid), pp 598–606, 2005.
27. R. Raman, M. Livny, and M. Solomon. *Matchmaking distributed resource management for high throughput computing*. Proc. Seventh IEEE Int. Symposium on High Performance Distributed Computing, 1998.
28. H. Tangmunarunkit, S. Decker and C. Kesselman. *Ontology-based resource matching in the grid - the grid meets the semantic web*. Proc. Second Int. Semantic Web Conference, pp 706–721, 2003.