Persistent Asynchronous Adaptive Specialisation for Data-Parallel Array Processing in SAC

Grelck, C.; Wiesinger, H.

Publication date
2013

Document Version
Final published version

Published in
CPC2013: 17th Workshop on Compilers for Parallel Computing: July 3-5, 2013, Lyon. Program

Persistent Asynchronous Adaptive Specialisation for Data-Parallel Array Processing in SAC

Clemens Grelck, Heinz Wiesinger
University of Amsterdam, Institute of Informatics
C.Grelck@uva.nl H.M.Wiesinger@student.uva.nl

SAC (Single Assignment C) is a purely functional (data-parallel) array programming language [6, 2]. As such, SAC puts multi-dimensional arrays into the focus. Such an array is characterized by a triple consisting of the *rank scalar* that defines the length of the *shape vector*. The elements of the shape vector define the extent of the array along each dimension, and the product of its elements determines the length of the *data vector*, which contains the array elements (in row-major unrolling).

SAC advocates shape- and rank-generic programming on multi-dimensional arrays, i.e. SAC supports functions that abstract from the concrete shapes (vector) and even from the concrete ranks of the arrays involved in some operation. Depending on the amount of compile time *structural* information we distinguish between three classes of arrays at runtime: For rank-generic arrays, all three properties (i.e. rank scalar, shape vector and data vector) are variable. For shape-generic arrays, the rank scalar is a compile time constant: The length of the shape vector is known in advance, but its elements are not. For non-generic arrays both rank scalar and shape vector are constants.

From a software engineering point of view it is (almost) always desirable to specify functions on the most general input type(s) to maximise code reuse. For example, a simple structural operation like rotation should be written in a rank-generic way, a naturally rank-specific function like an image filter in a shape-generic way. Consequently, the extensive SAC standard library is full of generic, mostly rank-generic functions.

However, genericity comes at a price. In comparison to non-generic code the runtime performance of equivalent operations is substantially lower for shape-generic code and again for rank-generic code [10]. The reasons are manifold and often operation-specific, but three categories can be identified nonetheless: Firstly, generic runtime representations of arrays need to be maintained, and generic code tends to be less efficient, e.g. no static nesting of loops can be generated to implement a rank-generic multidimensional array operation. Secondly, many of the SAC compiler's advanced optimisations [4, 5] are not as effective on generic code because certain properties that trigger program transformations cannot be inferred. Thirdly, in automatically parallelised code [3, 9, 1] many organisational decisions must be postponed until runtime and the ineffectiveness of optimisations inflicts frequent synchronisation barriers and superfluous communication.

In order to reconcile the desires for generic code and high runtime performance, the SAC compiler aggressively specialises rank-generic code into shape-generic code and shape-generic code into non-generic code. However, regardless of the effort put into compiler analyses for rank and shape specialisation, this approach is fruitless if the necessary information is not available at compile time as a matter of principle. For example, the corresponding data may be read from a file, or the SAC code may be called from external (non-SAC) code, to mention only two potential scenarios.

In order to reconcile software engineering principles for generality with performance demands we have developed an adaptive compilation framework [7, 8]. The idea is to postpone specialisation if necessary until runtime time, when all structural information is eventually available no matter what. A generic SAC function compiled for runtime specialisation leads to two functions in binary code: the original generic and presumably slow function definition and a small proxy function that is called instead by other code. When executed, the proxy function files a specialisation request consisting of the name of the function and the concrete shapes of the argument arrays before calling the generic implementation. Of course, proxy functions also check whether the desired specialisation has been built before, or whether an identical request is currently pending. In the former case, the proxy function dispatches to the previously specialised code, in the latter case to the generic code, but without filing another request. Concurrent with the running application, a specialisation controller (thread) takes care of specialisation requests. It runs the fully-fledged SAC compiler with some hidden command line arguments that describe the function to be specialised and the specialisation parameters in a way sufficient for the SAC compiler to re-instantiate the function's partially compiled intermediate code from the corresponding module, compile it with high optimisation level and generate a new dynamic library containing the specialised code and a new proxy function. The specialisation controller links the application with that library and replaces the proxy function in the running application.

In this paper we report on a series of extensions and generalisations of our previous work [7, 8]. One area of refinement pertains to the parallelisation of the specialisation process itself: the earlier specialised implementations of generic functions become available, the better for the performance of an application. We use more than one core for specialisation; more precisely, we dynamically adjust the number of cores reserved for adaptive code specialisation and thus taken away from the set of cores running the data-parallel application. Starting out with some default ratio, the expectation is that an application initially requires more specialisations while in many cases a fixed point is reached after some time or at least the need for specialisations reduces. Thus, we adapt the number of specialisation cores to the actual demand and leave as many cores as possible to the (implicitly) parallelised application. On the more technical side, we support specialisations originating from the data-parallel parts of an application (SAC employs an accelerator model of organising parallel execution) and try to combine related requests in the same compiler run for efficiency.

Another major area of refinement lies in the persistence of specialisations. In our previous work specialisations were confined to a single run of an application and automatically removed upon termination. We now aim at keeping specialisations alongside the original modules in a specialisation repository. Consequently, applications may benefit from the same specialisations across multiple invocations, and even completely unrelated applications may benefit from pre-specialised building blocks. This extension allows us to substantially reduce the potential overhead of runtime specialisation. Persistence, however, also creates a new range of research questions. For instance, specialisation repositories cannot grow ad infinitum. We employ statistical methods to decide when which specialisations may be displaced by others.

Our approach differs from just-in-time compilation of (Java-like) byte code in several aspects. In the latter hot spots of byte code are adapted to the platform they run on by generating native code at runtime while the execution platform was deliberately left open at compile time. This form of adaptation (conceptually) happens in a single step. In contrast, our approach adapts code not to its execution environment but to the data it operates on. This is an incremental process that may or may not reach a fixed point. The number of different array shapes that a generic operation could be confronted with is in principle unbounded, but in practice the number of different array shapes occurring in a concrete application is often fairly limited. Our approach is not specific to SAC, but can be carried over to any context of data-parallel array processing.

References

[1] M. Diogo and C. Grelck. Heterogenous computing without heterogeneous programming. In K. Hammond and H.W. Loidl, editors, *Trends in Functional Programming, 13th Symposium, TFP 2012, St.Andrews, UK*, volume 7829 of *Lecture Notes in Computer Science*. Springer, 2013. to appear.

[2] C. Grelck. Single Assignment C (SAC): high productivity meets high performance. In V. Zsók, Z. Horváth, and R. Plasmeijer, editors, *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary*, volume 7241 of *Lecture Notes in Computer Science*, pages 207–278. Springer, 2012.

[3] Clemens Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.

[4] Clemens Grelck and Sven-Bodo Scholz. SAC — From High-level Programming with Arrays to Efficient Parallel Execution. *Parallel Processing Letters*, 13(3):401–412, 2003.

[5] Clemens Grelck and Sven-Bodo Scholz. Merging compositions of array skeletons in SAC. *Journal of Parallel Computing*, 32(7+8):507–522, 2006.

[6] Clemens Grelck and Sven-Bodo Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[7] Clemens Grelck, Tim van Deurzen, Stephan Herhut, and Sven-Bodo Scholz. An Adaptive Compilation Framework for Generic Data-Parallel Array Programming. In *15th Workshop on Compilers for Parallel Computing (CPC'10)*. Vienna University of Technology, Vienna, Austria, 2010.

[8] Clemens Grelck, Tim van Deurzen, Stephan Herhut, and Sven-Bodo Scholz. Asynchronous Adaptive Optimisation for Generic Data-Parallel Array Programming. *Concurrency and Computation: Practice and Experience*, 24(5):499–516, 2012.

[9] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. Breaking the gpu programming barrier with the auto-parallelising SAC compiler. In *6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), Austin, USA*, pages 15–24. ACM Press, 2011.

[10] Dietmar Kreye. A Compilation Scheme for a Hierarchy of Array Types. In Thomas Arts and Markus Mohnen, editors, *Implementation of Functional Languages, 13th International Workshop (IFL'01), Stockholm, Sweden, Selected Papers*, volume 2312 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2002.