Harnessing the Driving Force of Dependencies

Eva Burrows *

Department of Informatics, University of Bergen, Norway
March 15, 2010

Parallelism Going Mainstream

Computational devices are rapidly evolving into massively parallel systems. Multicore processors are standard, and high performance processors such as the Cell BE processor [6] or graphics processing units (GPUs) featuring hundreds of on-chip processors (e.g. [10]) are all developed to accumulate processing power. They make parallelism commonplace, not only the privilege of expensive high-end platforms. However, classical parallel programming paradigms cannot readily exploit these highly parallel systems. In addition, each hardware architecture comes along with a new programming model and/or application programming interface (API). This makes the writing of portable, efficient parallel code difficult. As the number of processors per chip is expected to double every other year or so, entering parallel processing into the mass market, software needs to be parallelized and ported in an efficient way to massively parallel, possibly heterogeneous, architectures.

Related Work

One way of transforming manycore power into real application performance – as being adopted by most hardware vendors – is to provide libraries, APIs or some ready-to-use software toolbox that help developers to annotate legacy code with parallel constructs where possible (e.g. [11, 1, 2]), or to offer some low-level cross-platform standardization (i.e. OpenCL [8]) which targets both multicore CPUs and/or GPUs. While these approaches are undoubtedly for the benefit of practicing program developers, they do not constitute a unified parallel programming model. Computer science is in search for more radical solutions. It investigates the possibility of a new high-level parallel programming paradigm that can easily adapt to the era of commonly available parallel computing devices as well as to the increasingly more accessible realm of high-performance computing facilities.

One of the major issues in parallelizing applications is to deal with the underlying inherent dependency structure of the program. Miranker and Winkler [9] suggested that program data dependency graphs can abstract how parts of a computation depend on data supplied by other parts. This served as

*http://www.ii.uib.no/~eva/

a basis for parallelizing compilers ([12]), and proved the idea of embedding a program's communication structure into the hardware topology a reasonable approach. However, automatic dependence analysis is difficult for the general cases, and as a result parallelizing compilers cannot make the most of the underlying dependencies.

Data Dependency – A Programmable Interface

The constructive recursive approach developed by Haveraaen [7] allows the modular separation of computation from dependency, such that both are programmable independently from each other. This also entails a separation between local dependencies of a function, and the global communication pattern of the whole computation. The uniqueness of this approach consists in the fact that dependencies are captured by algebraic abstractions – Data Dependency Algebras (DDAs) – and turned into explicit, first class, programmable entities in the program code, so that a parallelizing compiler can harness directly the implicit driving force of dependencies. In a recent work [5] we pointed out how this approach can serve as a basis for a hardware independent parallel programming model.

My Research Directives

I explore the role DDAs can play in parallel computing from the on-chip parallelism of microprocessors via GPUs, up to parallel machine networks. I investigate algebraic properties that allow the combination and nesting of various DDAs to be used by a DDA-enabled compiler to generate efficient code. In addition, my work includes the design of new language features with various computational mechanisms targeting different hardware architectures (e.g. sequential, CUDA, MPI, FPGA placements, etc) from the same DDA-based program code. These are going to be implemented in the framework of the Magnolia programming language [3, 4], which itself is under development. Magnolia allows the definition of concepts to specify the interface and behaviour of abstract data types which are well-suited to express DDA concepts. I am working on the design of the DDA related backend features of Magnolia.

Intermediate Results

DDA abstractions allow us to modify at a high-level the way computations are mapped onto a given (parallel) hardware. They give us full control over data locality, and spatial placements of computations. The latter led us to the idea that DDAs are suitable abstractions for defining FPGA placements, which is going to be investigated in the near future.

DDAs can capture the space-time communication topology of modern parallel hardware, as it has been shown in particular for the CUDA model of NVIDIA's GPUs.

Practical experiments show that once the dependency pattern of a computation is defined in a separate module, this can be reused over various platforms. This results in high portability, but at the cost of slightly less optimal running times compared to fine-tuned platform specific program codes. A DDA-enabled compiler with a proper optimizing mechanism in place however is likely to alleviate this problem.

References

[1] Intel Parallel Studio. `http://www.intel.com/go/parallel`, 2009.

[2] Intel's Ct. `http://software.intel.com/en-us/data-parallel/`, 2009.

[3] A. Bagge and M. Haveraaen. Interfacing concepts. In T. Ekman and J. Vinju, editors, *Proceedings of the ninth Workshop on Language Decsriptions Tools and Applications LDTA 2009*, pages 238–252, 2009.

[4] A. H. Bagge. *Constructs & Concepts, Language Design for Flexibility and Reliability*. PhD thesis, Department of Informatics, University of Bergen, Norway, October 2009.

[5] E. Burrows and M. Haveraaen. A hardware independent parallel programming model. *Journal of Logic and Algebraic Programming*, 78:519–538, 2009.

[6] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementation – A performance view. *IBM Journal of Research and Development*, 51(5):559–572, 2007.

[7] M. Haveraaen. Efficient parallelisation of recursive problems using constructive recursion (research note). In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 758–761, London, UK, 2000. Springer-Verlag.

[8] Khronos OpenCL Working Group. *The OpenCL Specification. Version 1.0*, 2009.

[9] W. L. Miranker and A. Winkler. Spacetime representations of computational structures. *Computing*, 32(2):93–114, 1984.

[10] NVIDIA. *CUDA Programming Guide. Version 3.0*, 2010.

[11] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007.

[12] M. Wolfe, editor. *High Performance Compilers for Parallel Computing*. Addison Wesley; Reading, Mass., 1996.