Compilers for Heterogeneous Computing

Philip Ginsbach and Michael O'Boyle
School of Informatics, University of Edinburgh

Heterogeneous computing can be seen as a natural extension of parallel computing. Where traditional multi-threaded software runs on several identical processing cores, heterogeneous computing allows the processing cores to be structurally different.

This adds many additional challenges that have to be resolved for software to utilise heterogeneous hardware most efficiently. The segmentation of a single program into multiple threads remains an important task. Furthermore however, the scheduling of the tasks onto the different cores becomes a much more sophisticated problem that can no longer be easily resolved by a run time scheduler. This is because the cores are suited for different kinds of workloads, may use different ISAs and distinct address spaces.

Efficiently distributing code in these complicated environments generally requires domain specific knowledge and as of today compilers play only a minor role in the decision making. There is however extensive library support for numeric computations on the most prevalent heterogeneous combination of CPU cores with GPGPU accelerators. These libraries cover many important domains such as linear algebra, Fourier analysis, computer vision, and machine learning.

Our research intends to transfer the domain specific knowledge that is captured in these well optimised library functions to the compiler. The underlying idea is that if compilers were able to automatically identify specific computational patterns such as linear algebra, stencil computations, reduction operations etc. they could automatically substitute them by library calls or other well optimised code snippets that use domain specific knowledge for efficient computations on heterogeneous systems.

Preliminary work on a benchmark study showed that even a limited set of only three computational patterns (linear algebra, stencil computations, general reduction operations) achieves 60% coverage of the bottlenecks of two well-established benchmarks suites (NAS Parallel benchmarks, Parboil). Using optimised libraries such as Intel MKL and Halide as well as established techniques for parallelising reduction computations we achieved at least 20% speedup in 46% of those benchmarks running on an Intel i7-2860QM processor versus OpenMP baseline implementations. This reinforced our belief that providing compilers with recipes for heterogeneous hardware utilisation in a few dozen particularly prevalent bottlenecks will be entirely sufficient to achieve very good hardware utilisation in many use cases.

For compilers to recognise computational patterns in existing code, we developed a method to formalise computational patterns in terms of graph constraints on control flow, data flow and control dependence graphs. We paid particular attention to sufficient flexibility so that our recognition is robust enough to capture computational patterns across syntactically different implementations. We achieved good results on the above mentioned benchmark suites and are able to capture the previously identified computational patterns reliably using a prototype written for the LLVM compiler infrastructure.

In the near future we plan to extend our system to several more computational patterns and test it on more complex code bases as well as on additional benchmark collections. Furthermore we will implement and train probabilistic models that capture the behaviours of specific implementations of individual computational patterns. This will then allow our system to intelligently choose between competing options.