Parallel-Pattern Aware Compiler Optimisations: Challenges and Opportunities

Zheng Wang[1] **and Hugh Leather**[2]

[1]Lancaster University
[2]University of Edinburgh

ABSTRACT

This report outlines our finding that existing compilers are not aware of the pattern semantics and thus miss massive optimisation opportunities.

Introduction

Programming heterogeneous multicore systems is extremely challenging, but increasingly important. Without a solution we cannot use these devices efficiently – energy and time will be wasted, and progress in program efficiency will slow or stop. Parallel patterns, where programmers write algorithmic intents that abstract away parallelisation, heterogeneity, and reliability concerns, offer a partial solution[1]. Non-expert programmers can now write parallel code, but to achieve good performance is still challenge.

One of the main reasons that existing toolchains fail to optimise parallel-pattern based programs is that the optimizing tool, such as a compiler, simply treats patterns as it would call to any other library. Optimisations it could do on the sequential version are prevented by intervening library calls and data flow analysis fails. The library cannot make optimisations since it cannot inspect the muscle functions or retranslate them. Nor does the library feed information back to the compiler to improve the code.

Example

Here we provide two simple examples showing what will happen when the compiler fails to optimise across pipeline boundaries using Intel TBB/ICC and Google Golang. Our implementation can be downloaded from `https://goo.gl/y7bBdN`. The results were obtained by running the examples on a server with a 40-core Intel Xeon E5-2650 CPU @ 2.30 GHz, 64 of RAM, running CentOS 7 with Linux kernel 3.10. We used Intel ICC v18.02 and TBB v4.4 (compiled with -O3), and go v1.1.0.

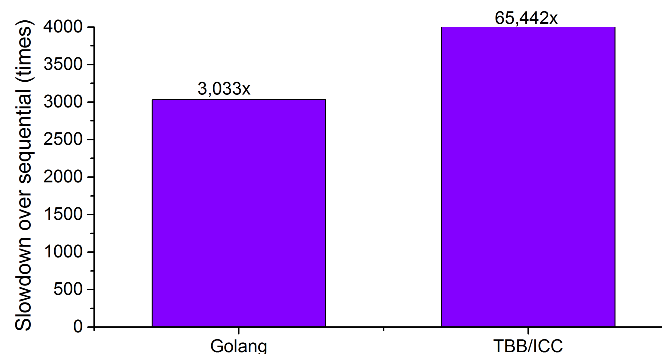

Figure 1. Experimental results showing Go and Intel TBB/ICC fails to analyze and optimise a simple pipeline program, leading to massive slowdown compared to a hand-optimised version.

In the examples, we create a pipeline of 100K stages. Each pipeline stage just adds one to its input. We compare the parallel version against a sequential version that just uses a simple for loop to carry out the same task. If the compiler is aware of the parallel patterns and can optimise across pattern boundaries, it could generate an optimised version to compute a single statement of 1 + 100000 = 100001.

However, the current implementation of Intel TBB/ICC and Golang fail to do so, leading to a slowdown of more than 3,000x over the sequential version as shown in the Figure 1.

Potential Opportunities

The issue arises from the fact that current compilers are oblivious to parallel patterns. A parallel construct encodes the sequential semantics of the program, but this is lost to the compiler. If the compiler knew the sequential semantics, it could e.g. do data-flow analysis and transformations across muscle functions, and dynamically merge pipeline elements that are too small to pay for their communication overhead. If we can do these, the primary barrier to adopting pattern-based programming would be torn down and heterogeneous multi-cores would be both easier to program and more efficient.

Aware of pattern semantics will permit us to perform optimisations like merging or splitting pipelines. One of the key challenges here is that the new set of pattern parameters, in combination of the existing optimisation options, will result in a massive optimisation space where finding the optimal parameters is non-trivial. Experience even with sequential programs has shown that the compiler optimisation space contains a huge number of possible options[2], which together define the *design space*. Within the design spaces will reside good solutions and bad – the task of finding the best compiler solutions is fundamentally infeasible in a traditional manual design process based on empirical know-how. A key enabling technology for tackling the parallel compilation problem is Machine Learning. Rather than hand-craft a set of optimisation heuristics based on compiler expert insight, learning techniques automatically determine how to apply optimisations based on statistical modelling and learning. This provide a rigorous methodology to search and extract structure that can be transferred and reused in unseen settings. Its great advantage is that it can adapt to changing platforms as it has no a priori. assumptions about their behaviour. There are many studies showing it outperforms human based approaches[3]. Recent work shows that it is effective in performing parellel code optimisation[4–16], task scheduling[17–20], model selection[21], etc. Therefore, machine learning can be an design methodology that provides a rigorous, automatic way to guide the search for the optimal compiler options for parallel pattern-based programs.

References

1. De Sensi, D., De Matteis, T., Torquati, M., Mencagli, G. & Danelutto, M. Bringing parallel patterns out of the corner: The p3 arsec benchmark suite. *ACM Trans. Archit. Code Optim.* **14**, 33:1–33:26, DOI: 10.1145/3132710 (2017).
2. Agakov, F. *et al.* Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, 295–305 (IEEE Computer Society, 2006).
3. Wang, Z. & O'Boyle, M. Machine learning in compiler optimisation. *Proc. IEEE* (2018).
4. Wang, Z. *et al.* Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM TACO* (2014).
5. Tournavitis, G. *et al.* Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI* (2009).
6. Wang, Z. & O'Boyle, M. F. Mapping parallelism to multi-cores: A machine learning based approach. In *PPoPP* (2009).
7. Wang, Z. & O'Boyle, M. F. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *PACT* (2010).
8. Grewe, D. *et al.* Portable mapping of data parallel programs to opencl for heterogeneous systems. In *CGO* (2013).
9. Wang, Z. & O'boyle, M. F. Using machine learning to partition streaming programs. *ACM TACO* (2013).
10. Wang, Z. *et al.* Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems. *ACM TACO* (2014).
11. Taylor, B. *et al.* Adaptive optimization for opencl programs on embedded heterogeneous systems. In *LCTES* (2017).
12. Ogilvie, W. F. *et al.* Fast automatic heuristic construction using active learning. In *LCPC* (2014).
13. Cummins, C. *et al.* End-to-end deep learning of optimization heuristics. In *PACT* (2017).
14. Ogilvie, W. F. *et al.* Minimizing the cost of iterative compilation with active learning. In *CGO* (2017).
15. Zhang, P. *et al.* Auto-tuning streamed applications on intel xeon phi. In *IPDPS* (2018).
16. Chen, S., Fang, J., Chen, D., Xu, C. & Wang, Z. Adaptive optimization of sparse matrix-vector multiplication on emerging many-core architectures. In *The 20th IEEE International Conference on High Performance Computing and Communications (HPCC)* (2018).
17. Grewe, D. *et al.* A workload-aware mapping approach for data-parallel programs. In *HiPEAC* (2011).

18. Emani, M. K. *et al.* Smart, adaptive mapping of parallelism in the presence of external workload. In *CGO* (2013).

19. Grewe, D. *et al.* Opencl task partitioning in the presence of gpu contention. In *LCPC* (2013).

20. Ren, J. *et al.* Optimise web browsing on heterogeneous mobile platforms: a machine learning based approach. In *INFOCOM* (2017).

21. Taylor, B. *et al.* Adaptive deep learning model selection on embedded systems. In *LCTES* (2018).