Keywords

Abstract interpretation, compilation, GPU, scratchpad memory, numerical abstract domains.

Location and supervision

The Ph.D will take place in LIP, Lyon, in partnership with IRISA, Rennes. Videoconference meetings will be organized every 2 to 4 weeks between LIP and IRISA. The thesis will be supervised by Matthieu Moy (60%, main supervisor), Laure Gonnord (20%) at LIP, and Sylvain Collange (20%) at IRISA.

Matthieu Moy and Laure Gonnord both belong to the new Inria joint team CASH, whose objective is to design various techniques for the optimised compilation of hardware and software. Laure Gonnord has a strong background in designing specialized static analyses for compilers. Matthieu Moy has a background in program analysis/verification and hardware architecture (in particular many-core processors).

Sylvain Collange belongs to the Inria joint team PACAP, Inria Rennes. His research objectives is to improve efficiency and the programmability of throughput-oriented processors through architecture changes and compiler techniques. He is a specialist of the GPU architecture. He has lead the Inria associate team Prospiel in the context of which he already collaborated with Laure Gonnord.

General Context

The gap between memory performance and compute performance of processor architectures keeps widening. On modern architectures, external memory accesses cost orders of magnitude more than computations. For instance, the Nvidia Geforce 1080 Ti GPU can perform 100 floating-point operations for each word read from external memory. As a consequence, reusing data through on-chip memories such as caches, scratchpads and register files is of paramount importance to optimize performance and minimize energy consumption.

However, little automation is available for these optimizations. Programmers are typically expected to place data manually in the various GPU memory spaces to obtain good performance on the GPU. Although new features such as Cooperative Groups, which enable direct inter-thread communication, hold the promise for highly-efficient data exchange between threads [4], they still have to see much use outside of hand-tuned library code [2].

In addition, optimizing for the GPU demands a programming style that would be counter-productive on multi-core targets, and vice-versa. For instance, GPU applications have to be written with fine-grained threading in mind. Consecutive threads should access contiguous locations, to make favorable access patterns for SIMD execution. However, general-purpose parallel applications written for multi-cores typically use coarser-grained threading. Different threads tend to access distant memory locations. These accesses are expensive to support efficiently in SIMD, and would thwart the benefits of GPU execution.

The objective of this thesis is to design and implement static analyses that automate the compilation of high-level code to optimized code for GPU targets, considering their specificities:

— in terms of programming model: all communications are expressed; but concurrent threads are grouped into “warps”, which execute the same common code on different data;
— in terms of memory usage: copies from and to the scratchpad memory are explicit and fine grain;
— in terms of specific instructions: with modern GPU, threads inside the same warp can communicate thanks to shuffle instructions that allow a direct register-to-register transfer, without accessing the shared memory.
As for analyses, the key component of memory optimizations is being able to infer precise data access patterns from array accesses; as well as precise information about register content.

**Phd content**

The goal of the PhD is to design static analyses of programs to infer properties that allow code transformation at least for the two following applications:
- the minimisation of memory transfers;
- the optimisation of the usage of *shuffle instructions* to communicate data between threads of the same wrap (group of threads).

The Ph.D will start with:
- Study GPU architectures, and the environments used to program them such as OpenCL and CUDA;
- From a set of programs provided to the PhD candidate, study interesting program transformations and their impact on performance;
- Design static analysis allowing the program transformations.

**Related work**  The work can rely on previous work on symbolic range analysis for pointers in the Ph.D of Maroua Maalej [6]. Compared to state of the art array analysis [7, 3], the precision can be relaxed since we do not need any information about the array content, but only information about addresses, hence array indices. We can also cite the work on the polyhedral abstract domain on binary code [1], where the difficulty is to precisely handle memory locations (binary code do not expose the notion of variable).

As for compiler optimisation techniques, related work include the polyhedral model's tiling transformations [8, 5], able to perform similar transformations. The goal here is to use abstract interpretation to get more widely applicable results.

**First step**  As a first step, the candidate will focus on programs manipulating 2-dimensional arrays, and programs running on GPU, where memory optimization is particularly crucial.

A straightforward range analysis is not sufficient for several reasons:

1. Addresses are rarely constant, but usually relative, e.g. to the start address of an array
2. For programs manipulating multi-dimensional arrays, a convex shape within the array is not convex in terms of address. For example a rectangle area within a 2D-array is a sequence of intervals. Using the usual convex hull operation on such shapes would loose important information.

As a consequence, we need symbolic analysis to address point 1), and intervals modulo a value to address point 2). The analysis will likely rely on classical abstract interpretation, with a new numerical domain adapted to 2D arrays. The numerical domain should be designed to be scalable, although the scalability constraint is not as strong for GPU compute kernels (which are usually relatively small) than for program-wide analysis.

The candidate will then validate his analysis on representative benchmarks, and propose an automation of the code transformation process.

The Ph.D requires a taste for both practical and theoretical research. Depending on the first results, and the preferences of the candidate, the balance between theory and practice can evolve on one side or the other (e.g. privilege static analysis, general code transformation, or close-to-hardware micro-optimizations).

**Références**


