Semantics-Preserving Data Layout Transformations for Improved Vectorisation

Artjoms Šinkarovs
School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh
a.sinkarovs@macs.hw.ac.uk

Sven-Bodo Scholz
School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh
s.scholz@hw.ac.uk

Abstract

Data-Layouts that are favourable from an algorithmic perspective often are less suitable for vectorisation, i.e., for an effective use of modern processor's vector instructions. This paper presents work on a compiler driven approach towards automatically transforming data layouts into a form that is suitable for vectorisation. In particular, we present a program transformation for a first-order functional array programming language that systematically modifies they layouts of all data structures. At the same time, the transformation also adjusts the code that operates on these structures so that the overall computation remains unchanged. We define a correctness criterion for layout modifying program transformations and we show that our transformation abides to this criterion.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation, Compilers, Optimization; D.1.1 [*Functional Programming*]; C.1.2 [*Multiple Data Stream Architectures (Multiprocessors)*]: Single-instruction-stream, multiple-data-stream processors (SIMD)

Keywords Vectorisation, Type systems, Correctness, Program transformation

1. Introduction

Most programming languages directly relate type definitions and type declarations to one particular layout of the data in memory. For example, FORTRAN maps arrays in a column major order into memory, whereas C uses a row major scheme. The fields of records or objects are typically adjacent in memory and algebraic data types such as those in many functional languages typically are mapped to pointer connected graphs in memory.

In languages that support an explicit notion of memory and pointers there is very little that can be done about this tight coupling between types and their corresponding mapping into memory. In contrast, language that completely abstract away from the notion of memory, such as purely functional languages, allow for almost arbitrary mappings of data into memory. While this opportunity may be less relevant in the context of algebraic data types, it definitely can have a huge impact on the performance of programs that operate on arrays.

It is well known from the optimisation of compute intensive applications in FORTRAN or C, that a reorganisation of data accesses can vastly improve the overall runtime [1, 8, 12] Improvements typically do not only stem from better cache locality but also from improved chances of the utilisation of vector instructions, i.e., auto-vectorisation. However, data dependencies in programs and a fixed data-layout often constrain what can be achieved. The high-performance computing literature provides many cases where manual re-writes for enforcing different memory layouts are crucial for achieving a sufficient level of performance [7, 9].

Here, a purely functional setting offers a unique opportunity. The implicitness of the memory management allows the compiler to adjust a programmer-specified layout in a way that improves locality and that enables better vectorisation opportunities. In [13] we have demonstrated for a small functional core language how a type system can be used to ensure consistent layout adjustments, we sketched an inference algorithm and we demonstrated the potential of the approach by looking at the N-body problem and an implementation in SAC. In this paper, we present a formal layout-transformation scheme that takes a program annotated with layout-types and transforms the program on the source level so that a straight-forward mapping into a pre-defined memory mapping achieves the desired layout transformation. This approach allows layout transformations to be implemented as a high-level AST to AST translation without requiring any adjustments on the low-level code generation. Besides the type-driven code translation scheme we also provide a correctness proof of our transformation. It is based on a correctness notion that captures the essence of data-layout independence by looking at equality as being factored by index-space transformations of function arguments and function results.

The rest of the paper is organised as follows: the next section formalises the programming language we use to describe our transformation, briefly introduces the layout type system and formalises correctness criteria. Section 3 introduces the transformation itself providing informal explanation across the lines. Section 4 gives a formal proof of the correctness of our transformation according to the criteria formulated earlier. Then follows a brief example of application of the described technique using matrix multiplication as a case study. Section 6 discusses related work and we conclude with Section 7.

2. Setting

We use a first order functional core language as basis for our formalisms. The language is a stripped-down version of SAC [2] very similar to the core language introduced in [3]. In essence, the language constitutes an applied λ-calculus with a C flavour when it

FHPC '13, September 23, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2381-9/13/09…$15.00.
http://dx.doi.org/10.1145/2502323.2502332