Data Layout Inference for Code Vectorisation

Artjoms Šinkarovs, Sven-Bodo Scholz
School of Mathematica and Computer Sciences
Heriot-Watt University
Edinburgh, United Kingdom
a.sinkarovs@macs.hw.ac.uk

***Abstract*—SIMD instructions of modern CPUs are crucially important for the performance of compute-intensive algorithms. Auto-vectorisation often fails due to an unfortunate choice of data layout by the programmer. This paper proposes a data layout inference for auto-vectorisation which identifies layout transformations that convert SIMD-unfavorable layouts of data structures into favorable ones. We present a type system for layout transformations and we sketch an inference algorithm for it. Finally, we present some initial performance figures for the impact of the inferred layout transformations. They show that non-intuitive layouts that are inferred through our system can have a vast performance impact on compute intensive programs.**

I. Introduction

In the last decade vectorisation became an important research topic again, as most of the modern CPUs grant vectorisation capabilities by means of SIMD instructions [10]. Classical research into auto-vectorisation focuses on the optimisation of loop nestings [8]. Data-independent operations within such loop nestings are identified and the loop-nestings as well as the order of operations within the loop nestings are reorganised to match pre-defined vectorisation patterns, typically sequences of identical arithmetic operations within loops. For vectorisation to be effective, such subsequent operations need to work on data that are adjacent in memory. Otherwise, loading/storing overheads in most cases outweigh any possible performance gains from using SIMD operations. Furthermore, vectorisation only yields a substantial benefit if it can be applied within loop nestings, preferably within the innermost loops. As a consequence, classical auto-vectorisation fails to deliver substantial performance improvements whenever loop nestings cannot be re-arranged to match the layout of the data structures that are being computed on.

In this paper, we propose a novel approach towards program vectorisation: rather than focusing on a reorganisation of loop nestings we suggest a reorganisation of data layouts to enable vectorisations. Key to this approach is a program analysis for inferring suitable data layouts. Based on this inference, a subsequent program transformation followed by classical auto-vectorisation achieve the overall goal.

Data layout inference comes with several challenges: Most importantly, we have to make sure that any new layout can be accommodated by means of a semantics preserving program transformation. Languages such as C give guarantees on how data are being stored in memory which we need to observe. Interfaces of modules need to be preserved, as well as potential effects that result from the sharing of data or code.

Another core challenge lies in the identification of suitable data layouts. The theoretically very large number of possible layouts for any given array needs to be narrowed down to a manageable size. At the same time we need to ensure that all those layouts that enable very good vectorised performance stay in this set. Further challenges arise from the differences in the executing hardware. They impact directly the way code can or should be vectorised and, consequently, they also impact the choice of data layouts.

In this paper, we focus on the challenge of identifying suitable layout combinations that enable auto-vectorisation. We propose a type inference, that identifies data layouts suitable for vectorisation. A functional core language serves as the basis for our formalisation. It constitutes a strip-down version of the programming language SaC which we use as a vehicle for our experiments. In SaC, all the data structures are n-dimensional arrays, data parallel loop-nests are expressed using an explicit syntactical construction, and memory management is fully implicit.

The key idea of our approach is a layout inference that identifies ideal layouts (wrt SIMD vectorisation) for each individual loop nesting and then employs representational changes whenever necessary. We provide a solution to the separate compilation problem by utilising the overloading capabilities of SaC. This enables code adaptations at the calling site without making representational changes inevitable.

We use the N-body problem as a case study throughout the paper. It nicely demonstrates the difficulties when attempting the classical approach to vectorisation and it also shows the effectiveness of our proposed approach. Finally, we present some initial performance measurements that show substantial speedups even in the presence of multi-threaded executions.

The paper is structured as follows: In the next section, we introduce our core language, a stripped-down version of SaC. Section III introduces our running example in that language and it discusses the key ideas of our approach at it. Section IV provides a formal presentation of our type system and Section V sketches a possible algorithm for inferring the layout types and applies it to the N-body example before Section VI shows the effect of these transformations on the execution times of our running examples. Related work is presented in Section VII before Section VIII concludes.