Abstract

General Purpose Graphics Processing Unit (GPGPU) programming enables substantial performance improvements in massively parallel problems, but requires a firm understanding of low level hardware concepts that are traditionally abstracted from an application layer programmer. Haskell, a purely functional programming language, is commonly referred to in the literature as "elegant", which has connotations with beautiful, modular and intuitive, but has no formal definition. Haskell's elegance potentially offers a solution to enable quicker and easier GPGPU programming. GPGPU programming in Haskell is a worthwhile exercise if this elegance can be preserved alongside performance benefits.

In the literature there is no formal definition for elegance, but the same concepts permeate discussions. Lazy evaluation, purity, type variables, higher-order functions, and elegant syntax (referred to as declaration-style programming) are found to be properties of elegance in Haskell. They form the working definition to assess the elegance of Haskell GPGPU programming. Accelerate and fractals are chosen as the suitable GPGPU library and problem area respectively under which to examine the question, before abstracting the results to Haskell GPGPU programming in general.

Working examples of generating the Mandelbrot Set in Haskell exist in Accelerate, demonstrating how to generate a static image, and a dynamic interactive GUI. The elegance of these is assessed, before iterating on the dynamic version to maximize the elegance under the working definition. The existing implementation is found to be high in purity, middling in declaration style & type variables, and low in laziness & higher order functions. The iteration introduces some improvements in type variables and declaration style, which remain limited by Accelerate. Laziness is unable to be introduced due to the execution of the program as a deep embedding in CUDA. Higher-order functions cannot be introduced to this specific implementation, but are found to be possible in general, and restricted in this implementation due to their link to laziness.

An evaluation of the results when applied to general GPGPU programming in Haskell show that Haskell GPGPU programming is elegant in terms of purity, syntax and higher-order functions, but not in terms of laziness and type polymorphism. An evaluation of research methods shows that type polymorphism was an invalid property under which to judge elegance. Under the iterated working definition, Haskell GPGPU programming is mostly elegant, aside from lack of lazy evaluation, which is seen to practically affect the elegance and modularity of code.