Optimising locality of reference in Futhark

Nicolai Nebel Jrgensen

January 8, 2017

1 Introduction

In the field of compiler and programming languages, there have recently been a trend of moving towards higher levels of optimisation concerning parallelism and accelerators such as GPUs. Since single-core CPUs are approaching the physical limits of computation speed, it becomes more attractive to exploit instruction-level parallelism. Even more so, multi-core CPUs are commonplace today and most desktop computers also contains a GPU.

GPUs in particular are components capable of highly parallel computations. The chips were designed to be able to process image data and graphics computations rapidly, but advanced compilers can use the architecture of the chip to speed up all manner of data-intensive computations. This can for many types of programs significantly speed up evalutaion.

Even so, with the speedup gained from using specialised hardware, it remains important for compilers to ensure proper program structure. This is captured in the concept locality of reference. A program with good locality of reference will have efficient interactions with memory, removing a bottleneck in the program. An optimised program may still arrive at such a problem, if it fails to account for its memory layout.

This report will discuss the concept of locality of reference as it relates to GPU code and provide an algorithm for optimising memory access. We will discuss the algorithm in the context of the Futhark programming language. We will also consider the difference between solving a problem for a single GPU computation, a kernel, and when solving for several kernels in sequence.

2 Background

2.1 GPU Architecture

2.2 Computation model of a GPU

2.3 Locality of reference on a GPU

Let us now consider the central concept in the report: Locality of reference on a GPU. Locality of reference refers to the idea of data being "close" in a program. What is meant by close depends on the memory model of the architecture running the program. A CPU stores recently used memory in a cache that is smaller but has much lower latency than the larger latency stored in RAM blocks. "Close" data in the context of a CPU means data that gives few cache misses.

There are, in general, two ways for data to be "close" in a program, temporal locality and spatial locality. Temporal locality is the concept of data being accessed frequently at some timespan and not at all or infrequently at other times. For example, a variable in a loop will be accessed for every iteration and then fall out of scope, it is temporally limited. On a GPU, the interpretation isn't quite as straightforward. Recall from the earlier sections that a GPU

executes in a SIMD for clusters of threads. These threads differ in exactly one variable, their thread index. If a variable doesn't depend on the thread index, even if it refers to a large array being iterated over, these threads will have the same copy of the variable. That is, for any given instruction step, the threads will agree on the variable. We can then say that it is temporally local to the cluster.

The other way we talk about locality of reference is spatial locality. When designing programs, the data they operate on are not unrelated. Often, large amounts of data are related closely to each other, such as when organised in simple arrays or other similar data structures. This data will most often be accessed together through loops or recursion. Having this data fragmented across the memory space will lead to cache misses on a CPU and result in slower performance. On a GPU, the concept of spatial locality is slightly more restrictive. Recall the memory model of the GPU, that a cluster of threads recieve data from the same bus and share memory in the same space. The bus is wide and capable of returning multiple values in a single lookup, but only if the values are close to each other in memory. If the memory addresses the bus needs are sequential, then the GPU has efficient memory performance. In this case, we say that the memory s coalesced.

As an example, consider a two dimensional array in memory. The rows of the array are laid out sequentially, with values side by side. If a program iterates over the columns of this array, the individual values will have a stride between them equal to the length of each row. In this case, the GPU will have to make a lookup for eachvalue individually. If the program instead iterated over a row, then the GPU could take multiple values in a single lookup. In this case access is coalesced.

2.4 Futhark

Futhark is a functional language in development that focuses on producing highly performant GPU code. Many of these optimisations concern themselves with running parallel computations on the GPU instead of the CPU. For example, many machine learning algorithms make iterations over large matrices. These computations, which are not graphical in nature, still runs faster if executed in parallel. Futhark achieves this by heavily analysing and optimising at compile time.

A simple Futhark program that squares the first n numbers looks like this:

```
fun main(n:i32):[]i32 =
  let xs = iota(n)
  let as = map (fn (x:i32):i32 =>
    x*x
    ) xs
  in as
```

Furthermore, Futhark uses a construct called Second Order Array Combinators (SOACs) to maximise the amount of parallelism the compiler is able to

take advantage of. Futhark is able to analyse and combine SOACs at compile time. There are five of them, being "map", "filter", "reduce", "partition" and "scan". In this report we will mostly focus on just "map", which behaves like the SML and Haskell function with the same name.

Let us consider how locality of reference applies in a Futhark program. Take the following example:

```
TODO: insert driving example here
```

Futhark is able to combine the nested maps to a single kernel. This kernel represents the different maps as iterations over variables. In this case, the array accesses aren't efficient.

```
TODO: walk through each array access here.
```

3 The algorithm

3.1 Intuition

Let us consider the program from the last section to understand the goal of the algorithm. The program doesn't have good locality, because the operations of array indexing will cause the GPU to stride over memory chunks to access individual values.

TODO: do explanation of program here, put high level optimised program in

3.2 Representation

3.2.1 Access patterns as sets of restrictions

When thinking about finding the optimal memory layout in kernel code, it is tempting to simply find the optimal representation for each memory access in sequence and use that. Intuitively this makes sense, since after all, were trying to be optimal.

However, two considerations must be remembered. Firstly, since a strategy that makes no transposition of an array is preferable to a strategy that requires it, a nave strategy will just forget about choices that are locally suboptimal. This becomes a problem because of the second consideration: In the greater scheme of a kernel several transactions may occur, and a locally suboptimal choice may cause later transactions to be way cheaper.

For example, the following access pattern is important to catch in the algorithm:

```
For (parallel) p1,p2,p3:
  A[p2,p3,p1];
  A[p1,p2,p3];
  A[p2,p1,p3];
```

In this case, a naive algorithm will find the first access, and conclude that p1 should be the innermost (thread variant) variable, since this will produce optimised access. However, because there are two accesses later that are only coalesced if p3 is the thread variant variable, such an algorithm would be incorrect.

The proposed algorithm instead considers the problem from a bottom-up perspective by representing a strategy as a list of all possible ways to make coalesced access. This way we capture the unintuitive early choice and keep it in memory for later consideration.

3.3 Composition

Now that we have our representation of locality of reference in a Futhark program, we can look at combining them. Above we argued that an optimisation for the program can be represented as a pair of values, ‘(Maybe Variable, [Variable], Transposes)‘, that defines a restriction upon a set of accesses. The first two elements define an interchange strategy and the last defines a transpose strategy.

Given two strategies ‘(in1, out1, trans1)‘ and ‘(in2, out2, trans2)‘ we combine them to a new strategy in the following way:

First, we handle interchanges. in1 and in2 must combine to zero or one variables. If in1 and in2 require different variables to be innermost, then the combination will fail. Otherwise, if either of in1 or in2 contains a variable, the new strategy will have that as well. If both in1 and in2 contain nothing, the combined strategy will have nothing as well.

out1 and out2 contain the variables each restriction requires to not be innermost. The combined restriction will contain the union of these. If this union causes the set to contain all kernel variables, then the combination will fail. Similarly, if the union contains the variable required to be innermost by the above operation, then the combination fails.

Then, we handle transposes. Each restriction has a mapping of array transposes. Each strategy must constrain every array to maximally one transpose. If they don't, we can't transpose an array to coalece accesses that fits the new interchange constraints, and thus the combination fails.

In this manner, the constraints are preserved and represented as a new strategy.