Optimising locality of reference in Futhark

Nicolai Nebel Jrgensen

January 11, 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

In order to understand how an optimisation of locality on a GPU can be achieved, we must first look the GPU itself. GPUs are processor chips traditionally designed and used for high-speed graphics. Recently, GPUs have expanded their field and are used in all sorts of other computations. This is called general purpose computing on GPU, or just GPGPU for short. In this project, we are interested in GPGPU and not just graphics processing.

A GPU is a processing unit just like a CPU, but it is designed for a different set of tasks. A GPU focuses on high parallelism in cores, while the CPU has higher speed on individual cores. For example, the recent NVIDIA GeForce GTX 1050 GPU contains 640 cores for processing, far more than a CPU will have. The tradeoff is that these cores are slower individually than a CPU core.

The cores on a GPU chip are not independent. They are organised together so they can be utilised efficiently in units called warps. Warps are clusters of 32 cores that execute together in Single Instruction Multiple Data fashion. That means that inside a warp, each thread will execute the exact same instruction

in parallel, but they will not necessarily share data. It is the responsibility of the warp to load memory. This means that when a warp performs a memory transaction, all cores will wait for the memory, effectively stalling the entire computation.

A warp acts as a stream processor. It recieves a stream of data, which it then distributes among its cores. The cores then apply the necessary operations on the data in the stream in parallel. This is the most basic idea of a GPU, that speedups can be achieved though distribution.

2.2 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.

2.3 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:

```
fun main(xs: [n][m]i32, muls:[k]i32, bias:[k][m][n]i32):[][][]i32 =
  map (fn (i:i32):[m]i32 =>
    map (fn (j:i32):i32 =>
      map (fn (k:i32):[n][m]i32 =>
        xs[j,i] * muls[k] + bias[k,i,j]
      ) (iota(k))
    ) (iota(m))
  ) (iota(n))
```

This, admittedly a bit contrived, example takes a 2D array `xs` and expands it out to a 3D array by mulitplying each element with elements in `muls` and adding a `bias`. Futhark is able to combine the nested maps to a single kernel. Thw kernel represents the different maps as iterations over variables. The variable `k` will be the variable distributed over GPU cores.

NOTE: Add rough kernel transformation here.

In this case, the array accesses aren't efficient.

`xs[j,i]` is currently fine, since both `i` and `j` will be the same across cores. That means this access is currently invariant and the data can be shared.

`muls[k]` is also fine, since the thread variant `k` is the innermost (and only) variable in the access. That means the access is currently coalesced.

`bias[k,i,j]` is the problem. `k` is variant between the cores but the access has it in the outermost access. That is, there is a large stride between values as `k` varies. This causes bad memory performance.

3 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.

Let us look at the example from the previous section. How would an optimised version of the program look like? The most obvious optimisation would in this case be to simply transpose the bias array, so that k is now the innermost dimension. We can add an annotation to `bias` to reflect that.

```
fun main(xs: [n][m]i32, muls:[k]i32, bias:[k][m][n]i32):[][][]i32 =
  map (fn (i:i32):[m]i32 =>
    map (fn (j:i32):i32 =>
      map (fn (k:i32):[n][m]i32 =>
        xs[j,i] * muls[k] + bias_transposed[j,i,k]
      ) (iota(k))
    ) (iota(m))
  ) (iota(n))
```

Note that the transposition of the other indexes doesn't matter with respect to locality. We only care about the innermost index. Another way that is a bit less obvious would be to use the interchange strategy and make `j` the thread variant index and then transpose the `xs` array to have `xs_transposed[i,j]`. This gives the following program:

```
fun main(xs: [n][m]i32, muls:[k]i32, bias:[k][m][n]i32):[][][]i32 =
  map (fn (i:i32):[m][k]i32 =>
    map (fn (k:i32):[m]i32 =>
      map (fn (j:i32):i32 =>
        xs_transposed[i,j] * muls[k] + bias[k,i,j]
      ) (iota(m))
    ) (iota(k))
  ) (iota(n))
```

Why might this be a useful optimisation? When `xs` is smaller than `bias`, the transposition is cheaper. Interchanging variables in the kernel space is a compile time optimisation that is free, so nothing is gained or lost there. For now, keep in mind that we want to capture strategies that are not immediately obvious.

3.2 Representation

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.

In the algorithm, the following representation (in Haskell notation) is used:

```
Strategy = { interchangeIn  :: Maybe Variable
           , interchangeOut :: [Variable]
           , transposes     :: Map Variable Integer
           }
```

There are three fields, two relating to interchanging kernel variables, on relating to array transposition.

Let us first consider the interchanges. The first field, `interchangeIn` is an option type that either has a variable or nothing. If it has a variable, it denotes the restriction that the variable must be the thread variant of the kernel. If it

contains nothing, then the choice is open. The second field, `interchangeOut`, is complementary to the first. It is a list of variables that cannot be thread variant for the strategy to be optimal. There are two restrictions imposed on these fields: No one variable can be present in both fields, as this would imply conflicting requirements, and `interchangeOut` may not contain all the kernel variables. Violating these restrictions would lead to the strategy being impossible.

Now let us look at the transposes. They are represented by the field `transposes` which is a map from variable names to integers. The variables in question here are array names, and the integers correspond to which array index should be the innermost one in order to create coalesced access. Any transposition that puts this index innermost should be coalesced after applying the strategy.

As an example, the strategy that captures the first optimal strategy would look like this:

```
{ interchangeIn = Just 'k'
, interchangeOut = ['i','j']
, transposes = ( 'bias' -> 2 )
}
```

Keeping `k` innermost keeps the access to `muls` coalesced. Making sure `i` and `j` are not innermost keeps the access to `xs` invariant between cores. Finally, the transposition makes the access to `bias` coalesced. Note that the indexes go from inner to outer, so a transpose with index 0 is a no-op.

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.

Let us now look at an example of combining two strategies. Consider these two strategies:

```
{ interchangeIn = Just ‘k‘
, interchangeOut = []
, transposes = ()
}
```

and

```
{ interchangeIn = Nothing
, interchangeOut = ['i', 'j']
, transposes = ('bias' -> 2)
}
```

First, we combine the `interchangeIn`. The first strategy has `Just 'k'` and the second has `Nothing`. The combination of these two is then `Just 'k'`. This keeps the restriction set by the first strategy, since the second doesn't care.

Then we combine `interchangeOut`. Set union gets us `['i','j']`. Now we check that `k` isn't part of the combined `interchangeOut`. It isn't, so the combination is still correct. Neither is every variable `i,j,k` in the combined set.

Lastly, we add the transposes together. In this case, it just becomes the map from the second strategy, since the first is empty. The combined strategy is then

```
{ interchangeIn = Just 'k'
, interchangeOut = ['i','j']
, transposes = ( 'bias' -> 2 )
}
```

Which we saw in the last section.

3.4 Combined algorithm

Now we have the tools to produce solutions to our original problem: Given a kernel, find an optimised form with good locality. We have the representation of a solution, and we have the means to combine solutions to (sometimes) form new ones. To put these pieces together, we just need a starting point. Once we have the initial solution for a simple problem, we can combine to produce a solution for the more complex one we are targeting.

The base case for our solution must be a single array access. Given a kernel space and a single array access, how can we generate possible solutions? There are two ways an access can be efficient, it is invariant or it is coalesced.

To create invariant access, we place the restriction that no variable that the access is variant to may be thread variant in the kernel.

To create coalesced access, we require that a specific variable is thread variant in the kernel and, if necessary, transpose the array so that its access becomes coalesced.

Let us look at a concrete example: A kernel with thread indexes `p1, p2` and `p3`. An array access of the form `A[p3,p1]`.

To handle the first case, we notice that the access is only variant to `p1` and `p3`. That means we simply place `p1` and `p3` in the out parameter of the restriction and don't transpose the array.

To handle coalesced access, we need to generate more than one access. For every variable that is variant in the access, we generate a restriction. For example, for `p1` in the example, we require it to be innermost and we don't need to transpose the array. We still store the null-transposition though, so future combines can't change it. For `p3` we also need to transpose the array, so the access will be like `a_transpose[p1,p3]`.

For every access in a kernel, we will then generate a set of possible restrictions. We can take two of these sets and combine every pair to generate a new set of restrictions, using our combine operation.

The algorithm is then as follows:

For every access in a kernel, generate the set of restrictions that solve the problem.

Combine all the sets of restrictions into a single set

From the resulting set, choose the best set of restrictions and apply them on the kernel, producing a new kernel.

Let us now show an example of generating the restrictions from a single access. Let us look at the access `xs[j,i]` from the example.

Firstly, we notice that the access is invariant to `k`. That means we can generate a restriction that ensures invariant access by making sure `i,j` are both not thread variant variables. The strategy for this would be `{ Nothing, ['i','j'], () }`.

Next, we can have both `i` and `j` to be innermost variables. If we force a variable to be thread variant, we must also make the transposition of the array such that access is coalesced. The restriction generated for making `i` thread variant is `{ Just 'i', [], ( 'xs' -> 0 ) }` and the one generated for `j` is `{ Just 'j', [], ( 'xs' -> 1 ) }`. Notice that we still carry the null-transposition with the strategy. That is because we fix the array to not be transposed and we need to save that information for combining later.

3.5 Complexity

Let us now consider the complexity of the algorithm. There are three steps in the algorithm, let us find the complexity of each.

The first step iterates over every access in the kernel. For every access in the kernel, it generates a number of restrictions. Recall that we generate one restriction that corresponds to an interchange strategy and one restriction corresponding to each dimension of the array index. This step then has complexity $O(A * I)$, where A is the number of accesses and I is the number of dimensions in the array access. If we assume that the number of accesses is larger than the number of dimensions except in very small kernels, we have complexity $O(A)$.

The second step of the process is a bit more complex. It can be viewed as a non-deterministic computation where at each step we make every possible combination and output the list of results. This combination takes $O(R^2)$ time. If the result also is $O(R^2)$, then the entire step would run in $O(R^n)$, which would be terrible. Here, we are helped by the fact that our combination may fail. Consider how a restriction combines. If the field `interchangeIn` contains a variable, it will only combine succesfully with restrictions that either carry `Nothing` or the same variable in the field. Since we maximally generate two such restrictions per access, we will only make two succesful combinations. We will also at most carry one restriction that has `Nothing` in `interchangeIn`, because of the way we defined our combination. This restriction may combine with every restriction in the new set.

The initial number of restrictions is $O(D)$ where D is the number of dimensions in the kernel space, and each step produces a new list with at most 3 times as many restrictions. There are n steps of combinations, which each take $(D * number_of_restrictions_accumulated)^2$ time, we arrive at the following complexity:

$$O((3 * D * N)^2 * N) = O(D^2 * N^3)$$

4 Intra-Kernel Optimisation

The algorithm generates a set of solutions for us to freely pick from. We want to pick the one that will result in the fastest programs. When looking at a simple kernel, we can easily find the best solution. Swapping around the dimensions of a kernel space is free at runtime, because we only change the thread variant index but keep memory representation the same. Transposing an array is not free, since we need to make a copy of the array and move the elements around at runtime.

Therefore, in order to pick the best solution given a single kernel we simply pick the restriction that requires us to transpose the least amount of arrays.

We need to note that this is not correct in all cases. For example, arrays have different sizes, so given an equal amount of transposes, we should pick the restrictions on the smallest arrays. It might also be that we can avoid

transposing an array by simply creating it in the proper configuration when it is computed. That would also eliminate a transpose.

Let's look at the example program a final time. What new form for the kernel will the algorithm find? Recall that the kernel generated by Futhark roughly looks like this: NOTE: Make this the kernel representation instead.

```
fun main(xs: [n][m]i32, muls:[k]i32, bias:[k][m][n]i32):[][][]i32 =
  map (fn (i:i32):[m]i32 =>
    map (fn (j:i32):i32 =>
      map (fn (k:i32):[n][m]i32 =>
        xs[j,i] * muls[k] + bias[k,i,j]
      ) (iota(k))
    ) (iota(m))
  ) (iota(n))
```

The algrithm looks at the three accesses and generates all possible restrictions for each one. It then combines them together to generate a set of solutions for the entire kernel operation. We now consider the best solution will be, according to our heuristic. We can convince ourselves that the algorithm will generate no solution requiring no transpositions. If `k` is thread variant then `bias` will be transposed. Likewise with `i`. If `j` is thread variant, then `xs` will be transposed. Therefore, the solution found will have one transposition, which could be either of `xs` or `bias`. If we arbitrarily pick the one that transposes `xs` and interchanges to have `j` as the thread variant index, we get the following kernel:

```
NOTE: Add transformed kernel with \verb‘xs‘ transposed.
```

5 Inter-kernel optimisation

A more complex and interesting problem is to consider the same optimisation we just made, but this time optimise for several different kernels in sequence. We consider this, because an optimisation that is optimal for one kernel might be wasteful on a larger scale. A motivating example will help here:

```
Transpose array so index 1 is innermost.
Kernel 1 (i,j,k):
  Interchange so that i is the thread variant variable.
  (...)
Transpose array so index 2 is innermost.
Kernel 2 (i,j):
  Interchange so that j is the thread variant variable.
  (...)
```

Suppose that there for both kernels existed a solution, such that index 2 would be innermost in both kernels. Then we can choose that solution in kernel 1 and save one transposition. This would be a better solution than just picking

blindly kernel by kernel.

We have not come up with a clever way of picking an optimal set of solutions to this problem, but it is possible that a simple brute force is good enough. Such a solution would be of exponential time complexity, which should cause us to consider it carefully and cautiously. It depends on how many kernels a function will have in real, complex programs and how many dimensions these kernel spaces will have. Recall that the number of solutions a kernel can have is $O(D)$ where V is the number of dimensions in the kernel space. The inter-kernel brute force approach is then $O(D^n)$. If n tends to be very small, then we can brute-force while keeping compilation times low.