Native Offload of Haskell Repa Programs to Integrated GPUs

Hai Liu Laurence E. Day Neal Glew Todd A. Anderson Rajkishore Barik

Intel Labs

{hai.liu,todd.a.anderson,rajkishore.barik}@intel.com led@cs.nott.ac.uk aglew@acm.org

Abstract

In light of recent hardware advances, general-purpose computing on graphics processing units (GPGPU) is becoming increasingly commonplace, and needs novel programming models due to GPUs' radically different architecture. For the most part, existing approaches to programming GPUs within a high-level programming language choose to embed a domain-specific language (DSL) within a host metalanguage and then implement a compiler that maps programs written within that DSL to code in low-level languages such as OpenCL or CUDA. An alternative, underexplored, approach is to compile a restricted subset of the host language itself directly down to OpenCL/CUDA. We believe more research should be done to compare these two approaches and their relative merits. As a step in this direction, we implemented a quick proof of concept of the alternative approach. Specifically, we extend the Repa library with a `computeG` function to offload a computation to the GPU. As long as the requested computation meets certain restrictions, we compile it to OpenCL 2.0 using the recently added feature for shared virtual memory. We can successfully run nine benchmarks on an Intel integrated GPU. We obtain the expected performance from the GPU on six of those benchmarks, and are close to the expected performance on two more. In this paper, we describe an offload primitive for Haskell, how to extend Repa to use it, how to implement that primitive in the Intel Labs Haskell Research Compiler, and evaluate the approach on nine benchmarks, comparing to two different CPUs, and for one benchmark to hand-written OpenCL code.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Compilers

Keywords GPU Programming; Heterogeneous Programming; Haskell

1. Introduction

With recent advances in computing hardware, general-purpose computing on graphics processing units (GPGPU) is becoming a common practice. Despite being labeled as general purpose, modern GPUs differ significantly from regular CPUs architecturally, giving rise to specific programming models and languages designed to simplify the programming of such devices. OpenCL [10] and CUDA [16] are two such popular frameworks that come with restricted C-based languages that address this need, and hardware vendors typically provide the necessary toolchain to compile OpenCL or CUDA programs to their devices.

OpenCL and CUDA are low-level programming languages and require programmers to write a range of boilerplate code to setup devices and manually manage memory. Obtaining reasonable performance usually requires programmer knowledge of some aspects of the compute resources and the memory hierarchy of the GPU.

In the past few years, however, several frameworks for GPU programming have emerged for high-level languages, which *do not* require familiarity with low-level details and languages. A popular approach is to embed a domain-specific language (DSL) within a host language. When execution of the host program on the CPU encounters an embedded program, the DSL compiler translates the embedded code into OpenCL or CUDA and then compiles and offloads that code to the GPU. Accelerate [4] and Obsidian [22] are two examples of this approach, using the functional language Haskell. Delite [3] is another such approach allowing not only the compilation and execution of DSLs on GPUs, but also a flexible technique for building DSLs using a multi-staged toolchain built on top of the Scala language.

If we view OpenCL as providing a hardware abstraction over data-parallel hardware architectures, then DSLs raise the level of abstraction even higher by hiding the hardware interface almost entirely. The obvious benefit is productivity due to higher-level constructs and operators. Additionally, the DSLs are carefully crafted so that only programs suitable for GPUs are expressible. This self-imposed design choice saves DSL implementers from writing a full-blown general-purpose compiler targeting GPUs, which is often infeasible given hardware limitations. Furthermore, DSLs naturally separate data in the host language from data inside the DSL, providing a hook to address the following aspect of GPUs. Specifically, discrete GPUs typically communicate incoherently with the CPU across a limited interconnect, and should be treated like distributed-memory machines. Thus, carefully choosing when to move data between CPU and GPU is important to achieving good performance.

On the other hand, the DSL approach has drawbacks of its own. To understand why, consider two similar libraries for parallel programming in Haskell, Repa [9] and Accelerate [4]. Repa is a Haskell library for high-performance array computation on multi-core CPUs, and Accelerate is an embedded array language targeting GPUs. They share similar high-level APIs for multi-dimensional and shape-polymorphic parallel arrays (no coincidence—the same team designed and implemented both), yet differ in ways beyond those due to differing hardware architectures:

- Repa programs are statically type-checked and compiled, whilst those written using Accelerate are only compiled at runtime.
- The DSL compiler for Accelerate has to re-implement optimizations already implemented by the host-language compiler.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FHPC'14, September 04, 2014, Gothenburg, Sweden.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3040-4/14/09…$15.00.
http://dx.doi.org/10.1145/2636228.2636236

The first versions of Accelerate indeed had performance issues due to missing general optimizations, which are not GPU specific [15].

- The DSL approach needs a special mechanism to interface with foreign code since DSL programs cannot directly use the FFI features already available to the host language [5].

An alternative approach for high-level languages, which we call native offload, is to compile a restricted subset of the host language itself to OpenCL or CUDA. To the best of our knowledge, this alternative approach has not been tried for any functional programming languages. If the restrictions are carefully selected, this approach should obtain many of the benefits of the DSL approach, while retaining all the facilities of the host-language.

With this in mind, we have implemented a proof of concept system for compiling a restricted subset of Haskell to Intel's integrated GPU, and we strongly believe that the results are interesting and show that this alternative approach is viable. Whilst we have employed a number of short-cuts for quick prototyping, the majority of these are engineering related, requiring more time and effort to address. One point in particular is worth emphasizing—we are targeting an *integrated* GPU. A number of companies have already been shipping processors with both a CPU and a GPU on the same die that can share the same memory system (e.g., Intel Haswell™, and AMD Kaveri™). The integrated GPUs from Intel also share the last level cache between the CPU and the GPU. Thus, the issues with data movement between CPU and GPU are largely mitigated. This difference simplifies compiling to GPU, and should be borne in mind throughout.

With these caveats in mind, we make the following specific contributions:

1. We introduce a new `computeG` combinator to the Repa library that offloads a parallel array computation to GPU.
2. We give a prototype implementation of compiling native Haskell functions to OpenCL as part of the Intel Labs Haskell Research Compiler (HRC) [13].
3. We integrate our solution with the Concord compiler (a C++ based heterogeneous computing framework for integrated GPUs that compiles to OpenCL) [2].
4. We demonstrate the effectiveness of the native-offload approach by comparing a set of Haskell benchmarks on both GPUs and CPUs.

2. Offloading Repa Array Computation

2.1 Overview of Repa

The Repa library represents the state-of-the-art for data-parallel computing with arrays in Haskell. It allows computation over high-rank arrays to be expressed in a type-safe manner, and at the same time enables implicit parallel execution on multi-core CPUs with aggressive array fusion guided by indexed types [12].

Consider the following Repa program `map2`:

```
import Data.Array.Repa as R

a :: Array U DIM2 Int
a = R.fromListUnboxed (Z :. 5 :. 10) [0..49]

b :: Array D DIM2 Int
b = R.map (^2) (R.map (*4) a)

c :: IO (Array U DIM2 Int)
c = R.computeP b
```

In the above program, both `a` and `b` are two-dimensional arrays, where `a` is fully manifest (type-indexed by `U`), and `b` represents a 'delayed' computation (type-indexed by `D`) based on the input array `a`. To fully compute `b` (i.e., to turn the delayed array `b` into a manifest array `c`), one can use either the `computeS` or the `computeP` combinators:

```
computeS :: (Shape sh, Unbox e)⇒
            Array D sh e → Array U sh e

computeP :: (Shape sh, Unbox e, Monad m)⇒
            Array D sh e → m (Array U sh e)
```

The difference between the above two functions is that `computeP` evaluates its input array in parallel by distributing the work across a set of worker threads (which can be specified by RTS option `-N` at runtime), whilst `computeS` evaluates sequentially. Besides delayed computations, Repa supports several other kinds of representations of typical array computations, but the gist remains the same: Repa array computations are made manifest only when we 'force' the delayed computation to evaluate to a concrete form (i.e., force an array of thunks to an array of values), and users do not have to specify anything other than replacing `computeS` with `computeP` to run the computation in parallel.

Following the same design philosophy, we extend the Repa API with a `computeG` combinator with the same type signature as `computeP` and with the intention that instead of spawning worker CPU threads, it offloads the actual computation to the GPU:

```
computeG :: (Shape sh, Unbox e, Monad m)⇒
            Array D sh e → m (Array U sh e)
```

In theory, with the `computeG` function added to the Repa library, it should be possible to run *any* Repa computation on the GPU, because semantically `computeG` is equivalent to both `computeS` and `computeP`. In practice, however, it is difficult to do this without compromise because of the differing underlying hardware architectures of GPUs and CPUs. Instead, we will restrict the computations that `computeG` will run by rejecting at compile time any computation that is ill-suited to run on GPU. To further understand the issues involved, we first discuss OpenCL and, more generally, the GPU computing paradigm.

2.2 Overview of OpenCL

OpenCL [10] is an open standard for cross-platform, parallel programming of modern processors, including CPUs and GPUs, digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and others. OpenCL includes a C99 based language, called OpenCL C, for writing *kernels* that are executed on OpenCL-compliant devices, and a set of APIs for controlling such devices. Although both task-based and data-based parallelism are supported by OpenCL, the latter still is the dominating programming model as it is well matched to the hardware characteristics of most OpenCL devices. Devices like GPUs usually consist of one or more compute units, each of which consists of a number of processing elements (PEs) and local memory. The most important OpenCL API call is to request execution of a *kernel instance* on a device. A kernel instance consists of a kernel, its arguments, and an *NDRange*. The latter is a contiguous rectilinear index set of integers in one, two, or three dimensions. For each index in the space, the GPU will execute the kernel on the given arguments and the index, this execution is called a *work item*. This kind of computation falls into the general category called single instruction multiple data (SIMD). NDRanges are additionally divided into *work groups*, contiguous sub-blocks of the index space of a user-requested size. Kernels have access to a local memory that is shared among the work items of a given work group. This local memory is limited in size, but usually much faster to access than global memory. It is often important to exploit locality and copy data to local memory for repeated processing.

OpenCL 2.0 is the latest iteration of the OpenCL standard, and among the many features and enhancements is the introduction of SVM (Shared Virtual Memory), which allows programs to directly share pointer-containing data structures between CPU and GPU. Integrated GPUs—those built on the same die as the CPU and therefore sharing the same physical memory and coherency domain—has significantly lowered the cost of offloading work from CPU to GPU, and made SVM an important feature to have.

An OpenCL application typically consists of a main program that runs on the host, and one or more kernels which run on OpenCL devices. The main program queries the system for a list of devices, sets them up appropriately, requests compilation of the kernels, creates kernel instances, and enqueues those kernel instances for execution on OpenCL devices. The OpenCL C language has a number of restrictions that limit the kind of programs that one can write. Most implementations provide a just-in-time compiler capable of compiling the kernel at runtime.

To illustrate, below is an example of an OpenCL kernel:

```
kernel void map2(global const int * restrict a,
                 global       int * restrict b)
{
    int idx = get_global_id(0):
    int x = a[idx] * 4;
    b[idx] = x * x;
}
```

The above kernel performs the same computation as the Repa program given in Section 2.1. The main program must allocate both arrays, initialize the source array `a` and then build and enqueue the kernel instance for execution with an appropriate range (i.e. the size of both arrays). Ignoring device setup and error handling issues for the time being, we can generalize this notion of data parallel execution via the following (Haskell) type signature:

```
offload# :: Int → (Int → State# s → State# s)
                → State# s → State# s
```

The `offload#` function takes two arguments: the range of the task to run on the external device, and the actual function, mapping from a valid index (within the range) to a stateful computation with the potential to read from and write to the main memory. Implicitly, we assume a form of memory coherence between CPU and GPU. As a consequence, we will make use of the SVM feature introduced in OpenCL 2.0. Furthermore, there are limitations on what offloaded functions can actually *do*—for instance, they must adhere to the same set of restrictions on the kernel functions as those for OpenCL kernels. In contrast with the DSL approach taken by Accelerate and others, the type of `offload#` gives little guarantee on its runtime behavior, and so any restrictions must be enforced either at compile time or runtime. As such, we specify `offload#` as a primitive in the implementation and not as part of the general API. In Section 3, we will discuss the set of restrictions again in greater detail, and moreover how they are enforced.

2.3 Implementing computeG

The `offload#` primitive represents the line we draw between high-level abstractions that can be implemented in a library (such as `computeG`) and the lower-level implementation details that are best handled by a Haskell compiler such as GHC and HRC.

Internal to Repa, `computeP` is implemented by forking a set of native threads (called a `Gang`), and dispatching chunked computations of the target array as individual tasks to each thread. The implementation of `computeG` is similar in the sense that it also relies on individually computing each chunks, but does so by offloading work to an external device rather than by spawning CPU threads.

The actual mechanism of chunk division varies according to the representation of the target array, and is implemented by declaring instances of the `Load` class:

```
class (Source r1 e, Shape sh)⇒ Load r1 sh e
    where
  -- | Fill an entire array sequentially.
  loadS   :: Target r2 e⇒
             Array r1 sh e → MVec r2 e → IO ()
  -- | Fill an entire array in parallel.
  loadP   :: Target r2 e⇒
             Array r1 sh e → MVec r2 e → IO ()
  -- | Fill an entire array in parallel via
    offload.
  loadG   :: Target r2 e⇒
             Array r1 sh e → MVec r2 e → IO ()
```

Without diving into the details of Repa's internal implementation, let us just say that this `Load` class abstracts the computation of all elements defined by an array (of type `Array`) and writes them to a manifest target of type `MVec`, a mutable vector representation. Here we add a new `loadG` method with the same type signature as the sequential `loadS` and parallel `loadP`. We then implement `loadG` for each array representation. For example, delayed array computations are offloaded as follows:

```
import GHC.IO (IO(...))

instance Shape sh⇒ Load D sh e where
  loadG (ADelayed sh getElem) mvec
   = mvec ‘deepSeqMVec‘ (IO dispatch >>
                            touchMVec mvec)
   where
     dispatch s =
       case size sh of
        I# n → (# offload# n f s, () #)
       where
         f i s =
           case w of
             IO m → case m s of
               (# s', _ #) → s'
           where
             w = unsafeWriteMVec mvec (I# i)
               ((getElem . fromIndex sh) (I# i))
```

In the above, we first compute the range `n` of the computation to offload, which is the size of the array to be made manifest. The actual function `f` which we offload maps each index `i` to an array-write operation using the utility function `unsafeWriteMVec`, which takes an array, an index, and a computation of the value of the element to write into the array at that index. Unlike the `computeP` implementation, we need not chunk the target array into suitable sizes beforehand, because in general the thread model is a poor fit for GPU devices. Instead, we specify the overall work-range to be the size of the destination array, and defer the dispatching of jobs to available compute units to the underlying OpenCL computation.

In a similar manner we implement `loadG` for other array representations (including Cursored, which share index computations between array elements, and Partitioned, which allow different element-indexing functions to be defined for distinct partitions of an array). With a relatively small set of changes, we produced a Repa library that implements `computeG` via the `offload#` primitive. Our goal is to be able to replace any calls to `computeP` with calls to `computeG`, and therefore convert a data parallel Repa program from CPU execution to GPU execution. We must highlight at this point that when using the `computeG` combinator, no assumptions can be made regarding the order of execution, and moreover floating point determinism cannot be guaranteed.

3. Implementing the Offload Primitive

Implementing `offload#` has a number of issues. To solve these issues we made two main choices: first we decided to implement in HRC and utilize Concord, and second we restrict functions that can be passed to `offload#` to ones that we can compile to GPU code. We explain these choices, the issues they address, and the way they solve these issues next.

3.1 Implementing in HRC

HRC is an alternative backend for GHC. It uses GHC as a front-end and intercepts the external Core after GHC does its optimization of Core code. HRC also implements a multitude of optimization passes based on a strict SSA-style internal representation called MIL. Specifically it converts to a strict IR, optimizes that, converts that to MIL, and does a number of control-flow and data-flow based optimizations. What makes MIL unique is that it combines a low-level CFG-based representation with a high-level object-based memory model. Of particular note is the contification optimization [7], which, when appropriate, converts mutually tail-recursive functions into loops, and optimizations to flatten data structures into other data structures or local variables, thus often eliminating allocation of temporary data structures. Eventually, the MIL code is translated into an extension of C called *Pillar* [1], which is passed on to the Pillar-to-C converter, before being compiled and linked by a C compiler (we use Intel's C compiler).

We choose HRC over GHC to implement an alpha prototype of this work for a number of reasons:

1. HRC's existing backend already generates C-like code, which makes it easy to re-use the same compilation pipeline to target OpenCL.
2. HRC performs a number of loop-based and representation-style optimizations [13, 17], and is able to produce straight loop code for many Repa programs. This is a good fit for OpenCL because OpenCL does not allow recursive function calls.
3. We previously conducted a performance study using a set of Repa benchmark programs compiled by HRC for both the Xeon CPU and the Xeon-Phi coprocessor [18]. We re-use the same set of benchmarks in this study to compare performance.

Because HRC uses GHC as its frontend, we modified GHC, adding the `offload#` primitive by declaring its type and giving it an empty implementation. We need not implement `offload#` in GHC, as we only need GHC to compile the modified Repa library down to Core code, which is then passed to HRC as input. Since HRC is a whole-program compiler, when compiling a source program it will also pull in Core code from all libraries used by the program, and among them is our modified Repa library where we can find the definition of `computeG` in terms of `offload#`.

3.2 Utilizing Concord

We need to implement runtime support for setting up OpenCL devices, calling OpenCL kernels, and integrating SVM into the garbage collector. On the last point, code before the offload will allocate and build data structures on the CPU that the offloaded code may want to access. We want to use the SVM mechanism to pass pointers from the CPU code to the GPU code that the latter can just use to access the data structures. To achieve this, any such pointers must be in the space covered by the SVM mechanism.

To address all these issues we use Concord [2], a heterogeneous C/C++ programming framework for processors with integrated GPUs with SVM support. Concord implements SVM in software today making it suitable for processors like Ivy Bridge and Haswell from Intel. It enables existing multicore applications that use pointer-based traversals to take advantage of GPUs easily without having to marshal and un-marshal data.

Any memory that Concord code wants to share between CPU and GPU must be allocated with special SVM allocation operations. Since we do not know what parts of the garbage-collected heap might be used by GPU code, we simply place all of the heap within an SVM allocated area. Then any pointers into the heap can be used with the SVM mechanism and accessed on the GPU. Due to an OpenCL restriction, Concord limits SVM memory to 400 megabytes, which in turn limits our heaps to 400 MB. For now, this limit does restrict which programs we can offload. However, future implementations will use OpenCL's fine-grained SVM (which is not currently supported by hardware), and there is no limit on fine-grained SVM memory, so there is no long-term concern.

Also, we must prevent garbage collection while running GPU code, as the garbage collector needs to halt threads and then scan and possibly update their stacks. OpenCL and thus Concord currently provide no mechanism for achieving this. At worse, some CPU threads will be blocked waiting for garbage collection so that they can allocate, blocked until the GPU code completes. The programs we have run, however, do no interesting CPU execution while GPU code is running.

3.3 Restricted Functions Offloaded

The two arguments passed to `offload#` are the range and the function to offload. The range argument is easy to handle, but the function to offload comes from arbitrary Haskell code. In particular, when we compile the offload, we may not know what actual Haskell code is being called at runtime. Even if we are able to locate the code that is called, it could contain arbitrary code, including allocating memory, evaluating thunks, making calls to other functions, and so on. OpenCL and thus Concord do not support arbitrary code, so we either have to implement difficult encodings of Haskell constructs in terms of what is allowed, or restrict the Haskell code that we will compile to Concord code. We choose to restrict what we compile, and will issue and error and stop compilation if these restrictions are not met.

First, we require that HRC can determine which MIL function will be called by any call to `offload#`. If after all of GHC's and HRC's optimizations, this function cannot be determined, HRC will issue an error message and stop compilation. Our implementation of `computeG` passes a function to `offload#` that HRC can determine the MIL function for. In fact, since `offload#` should really only be used in the low-level parts of libraries providing high-level parallelism abstractions, it should be easy for those libraries to avoid this compiler error.

Second, we restrict what code is allowed in the function passed to `offload#`. Ideally, HRC would scan the MIL function passed to `offload#` and check for the use of any constructs that it cannot effectively compile to Concord code, and if any are found then HRC would issue an error message and stop compilation. We do not currently run this check pass; it would be easy to implement. Note though, this check is done on MIL code, which is far from Haskell source code. Any error messages might be difficult to interpret. Figuring out good ways to explain the error in terms of source would be good research, but we leave that to future work. Instead, We rely on the Concord compiler to report errors for anything that it does not understand or support. Violations to almost all of our restrictions are being caught this way. Next we describe the main restrictions.

Calling Other Functions Concord and OpenCL do not support indirect calls, as these either are not possible or perform badly on most GPUs. Theoretically speaking, it is possible to completely eliminate the use of runtime function pointers by a whole-program transformation called de-functionalization [6, 21], and hence make

every function call statically known. However, HRC does not currently implement this. Alternatively, HRC could restrict function calls to those where a single MIL function is known to be the one called. In this case, we can generate a direct call to that function. We do not currently implement this either, but it is simple engineering effort to traverse the call graph and generate code for all MIL functions that might be called on the GPU.

Recursive Functions Concord and OpenCL do not support recursive function calls, and it is neither easy nor likely to perform well to work around this restriction. Therefore, we should reject any recursive functions that might be directly or indirectly called by code executed on the GPU. Note that due to contification, this restriction is far less restrictive than it might first seem. Any algorithm that conceptually is non-recursive functions and loops can be easily written in Haskell in a way that HRC will contify into non-recursive functions and loops.

Calling Foreign Code Haskell has an FFI that can be used to call non-Haskell code. It is difficult, generally impossible, to make available on the GPU the same set of foreign code as is available on the CPU. We do allow foreign calls to OpenCL code or Concord code or to built-in OpenCL functions.

Allocation It is not impossible to write a memory allocator for GPU code, but it is not easy and performance is not likely to be good. Implementing garbage collection on the GPU is hard, and implementing garbage collection across the CPU and GPU both is even harder. Rather than attempting to tackle these challenges, we choose to disallow allocation in GPU code.

Laziness Unlike Accelerate or other DSL solutions, Repa is a native Haskell library and thus must adhere to Haskell's lazy semantics. In general, laziness requires forming thunks and later evaluating them. Forming thunks requires allocation, and evaluating them often involves unknown calls. Since we do not support allocation and unknown calls, we disallow thunks and thunk evaluation, meaning that the Haskell code, by time it gets to HRC must be strict.

As GHC and HRC have aggressive optimizations for avoiding laziness and executing in a strict manner when that is a correct optimization, and as Repa is implemented carefully to be mostly strict, typical Repa programs do end up being strict at HRC's MIL IR level. However, this is not always the case, but usually strictness annotations can correct matters.

Exceptions It is difficult to propagate exceptions generated in GPU code through that GPU code and back to the CPU. Therefore, we disallow exception throwing and catching constructs.

3.4 Implementation

With these choices, it is then straightforward to implement the actual `offload#` primitive as follows. Intercept all offload calls at the code generation stage *before* Pillar code is produced. Then, for each call, we first ensure the kernel is a known MIL function, and then examine the body of the kernel function to check for no-allocation and no-thunk-eval conformance before outputting them as separate source files to be compiled by Concord. Finally, we replace the `offload#` call in the main program with a Concord function that runs the kernels themselves. We ensure that all kernels are properly compiled by Concord, and linked with the main program together with the Concord runtime.

4. Optimization for Performance

The Repa library is known to produce high-quality code through a set of advanced optimizations including type-indexed representations, heavy use of the `INLINE` pragmas and GHC rules to help fuse array computations. The result is particularly effective considering that Repa is not a full-blown DSL compiler but rather a library, and as such has to rely on GHC to do the actual optimization. This kind of reliance can be fragile at times, as we cannot be sure if an optimization really has taken place unless we examine the generated Core code, or even lower, such as LLVM code via the GHC LLVM backend, and MIL code in HRC. In constrast, the DSL approach taken by Accelerate and Obsidian has direct control over which optimizations go into the DSL compiler, and how they are implemented and tuned. However, it cannot make good use of the optimizations already implemented in the Haskell compiler. Both approaches have different pros and cons, and in this section we explore ways to help users write Repa programs appropriate for GPU offloading, and ways to aid compilers to optimize the generated code.

4.1 Strictness

As discussed previously, HRC rejects kernel functions that contain thunk-eval instructions, which means that as a Repa library user, we must ensure that the computation passed to `computeG` is strict. This is often the case, but sometimes the situation is more complex than appears at first glance. To paraphrase an example given by Lippmeier, et al [12]:

```
diagonals ::    Array U DIM1 Int
             → Array U DIM2 Int
             → Array U DIM1 Int
diagonals xs ys = computeG
    (R.map (\i → ys `index` (DIM2 i i)) xs)
```

One would expect that the function `diagonals` completely evaluates the map function over the two input arrays `xs` and `ys` when it builds the output array, and thus is strict in both arguments. Unfortunately, this is not the case under lazy evaluation, as the array `ys` is not demanded at all if the length of `xs` is zero. Indeed, if we compile this program with our compiler, it will complain about a thunk evaluation in the kernel function that corresponds to the evaluation of `ys`.

To alleviate this problem, Lippmeier, et al [12] suggest users "add bang patterns to *all* array parameters for functions using the Repa library", and use `seq` when bang patterns are not sufficient. This might come only as a minor annoyance in practice, but it is important to know the difference, especially when DSLs like Accelerate impose a strict semantics while Haskell and Repa do not.

4.2 Branch Avoidance

GPUs are not good at executing branch code in general. Although the OpenCL compiler will not complain when you compile code with branches, runtime performance suffers. To illustrate, consider the following OpenCL kernel skeleton:

```
kernel void f(...)
{
   ...
   if (C) {
      A;
   } else {
      B;
   }
   ...
}
```

When this kernel is offloaded to a typical GPU, multiple hardware threads will execute its instructions in lockstep. Some of those hardware threads might evaluate `C` to true, and others might evaluate it to false. The first group will execute the instructions of `A` in lockstep while the second group sit there idle; then the first group

will sit idle while the second group executes the instructions of B. Effectively, all the hardware threads take time equal to the time to execute both branches, rather than just the one taken on that hardware thread, a significant waste of computing cycles.

If conditionals cannot be avoided at the source level, we should attempt to minimize the number of instructions in both branches. Unfortunately, this is not a factor that the user can control when compiling Haskell programs with GHC, as GHC performs many code transformations when optimizing a program, with a particular tendency to push code towards branch leaves in the hope that this will reveal more opportunities for optimization. Therefore, when examining the code generated for offloaded kernels, we often see branches with relatively large amounts of code in both of them, with many instructions common to both branches (after alpha-renaming variables to match them up). This issue is not a problem unique to GPU offloading, but was previously also encountered when implementing SIMD vectorization for CPUs in HRC.

To reduce the impact of situations such as these, we implemented an algorithm that attempts to merge conditional branches using a new conditional move (`CMOV`) primitive introduced into the MIL IR. A `CMOV` is semantically similar to the following C expression:

```
c ? a : b
```

That is to say, `c` represents a boolean value, and both `a` and `b` represent values of the same type. The expression evaluates to `a` when `c` is true, and to `b` otherwise. Unlike in a general C expression where only one branch needs to be evaluated, since MIL requires that `a`, `b` and `c` are either constants or variables, their values will already have been evaluated prior to reaching the conditional.

MIL has a CFG-based block structure, where conditional branches are represented by a case switch that can potentially jump to one of several destination blocks. We impose the following preconditions for the branch merge optimization:

- The conditional is a binary switch over a boolean variable. Binary switches over non-boolean values are easily converted into boolean ones.
- Both branches contain only one instruction block, i.e., they do not contain intermediate block transfers.
- Both branches must share a unique predecessor block, the block that performs the case switch.
- Both branches must share a unique successor block to ensure that there are no more branching transfers after the merge.

We illustrate the branch merging algorithm in Figure 1 as psuedo-code written in Standard ML. We show only the type signatures of many utility functions used, and leave out their detailed definitions due to the lack of space. The `mergeBranch` function takes the two blocks from each branch as input, and produces a single output block if the merge is successful, or `NONE` otherwise. The actual merging is done recursively by the `merge` function that tries to match up instructions from each block in a pairwise manner, while keeping the output block as part of a `state` being passed around. The `state` also keeps track of variable equivalences (in `vMap`), as well as equivalence assumptions (in `aMap`).

For each pair of instructions, if they are equivalent, the `match` function (not shown in Figure 1) adds a new instruction to the output block after alpha renaming. Equivalence means that both instructions may only differ in their arguments. The `vMap` keeps track of existing equivalent arguments, and if two arguments are not found in `vMap`, we must assert their equivalence by updating `aMap` and `vMap`. A new assertion also triggers adding a CMOV instruction to the output block, selecting one of the two arguments based on the boolean variable of the branch that we are merging.

```
(*
 type state = (equivMap, assumptMap, block)
 empty     : 'a Map
 threshold : int
 firstOf   : block → instr option
 nextInstr : instr → instr option
 >>= : 'a option → ('a → 'b option) → 'b option
 match : state * instr option * instr option
         → state option
 merge : state * int * instr option * instr option
         → (int * block) option
 mergeBranch : block * block → block option
*)
merge (state, mismatch, i1, i2) =
  let val (vMap, aMap, blk) = state
  in if mismatch > threshold then NONE
    else case (i1, i2)
      of (NONE, NONE)⇒ SOME (mismatch, List.rev blk)
       | _⇒
        let val i1' = i1 >>= nextInstr
            val i2' = i2 >>= nextInstr
        in case match (state, i1, i2)
          of SOME state⇒       (* we have a match *)
             merge (state, mismatch, i1', i2')
           | NONE⇒           (* we have a mismatch *)
            let fun mergeNext (u, v) = fn i⇒
                    merge ((vMap, aMap, i::blk),
                           mismatch + 1, u, v)
                val l = i1 >>= mergeNext (i1', i2)
                val r = i2 >>= mergeNext (i1, i2')
            in (* choose one with fewer mismatches *)
              case (l, r)
                of (SOME (m, _), SOME (n, _))⇒
                   if m < n then l else r
                 | (SOME _, NONE)⇒ l
                 | (NONE, SOME _)⇒ r
                 | (NONE, NONE)⇒ NONE
            end
        end
  end

mergeBranch (b1, b2) =
  let val state = (empty, empty, [])
  in case merge (state, 0, firstOf b1, firstOf b2)
    of SOME (_, blk)⇒ SOME blk
     | NONE⇒ NONE
  end
```

Figure 1. Branch merging algorithm in Standard ML

If two instructions fail to match, we increment a `mismatch` counter, choose to shift either one of the instructions to the output block, and continue to merge the remainder of both blocks. We do the merge recursively up to a certain mismatch `threshold`, and choose a result that eventually yields fewer mismatches. The `threshold` is a heuristic to ensure that we do not overly merge branches that are indeed substantially different from each other. Ideally, we should check instruction equivalence modulo instruction re-ordering when permitted by side-effect constraints, but for simplicity we choose not to implement this and adhere to the existing instruction order in the input blocks.

The actual equivalence check of two instructions is more complicated than we describe here due to the variety of instruction types, and also for correctness we must disallow mismatches that have side-effects. Such details are omitted in Figure 1.

4.3 Cache Locality

A topic related to multi-dimensional array representations is how to best make use of cache locality and avoid unnecessary cache misses and page faults. Repa uses a row-major representation for multi-dimensional arrays, so it is best to structure innermost loops along the rows, and use blocking techniques to structure the outer loops so as to maximize the use of the cache-line. For example, the

Repa library already implements explicit memory blocking in the load function for its cursored array representation. It also tries to make use of the Global Value Numbering (GVN) optimization that is available in GHC's LLVM backend by loop unrolling and relative index calculation [11]. HRC implements a range of arithmetic optimizations and has initializing-once immutable arrays, and effectively is able to achieve similar results. This proves to be quite useful at reducing the number of memory load instructions for stencil programs, which we will look at in greater detail in Section 5.3.

Cache locality optimization in general applies to both CPUs and GPUs, however there are also CPU- or GPU-specific techniques. For example, we know that execution resources on typical GPUs are grouped, and within groups share some local memory. Therefore, in addition to using a global ID, it may be beneficial to use group and local IDs and make use of local memory. Unfortunately we cannot make use of this kind of cache locality optimization because our `offload#` primitive assumes a linear range, which imposes a flat rather than hierarchical structure, and once a multi-dimensional index is collapsed into a linear one, its structural information is lost. One of the reason that we choose to support only linear ranges in `offload#` is because this is also what Concord currently provides. Whether changing to a hierarchical range could bring real cache locality benefit remains a research topic for future work.

5. Benchmark Results

We measure the performance of native offloading to GPU using a set of "embarrassingly parallel" benchmarks written using the Haskell Repa library. The majority of these benchmarks originate from our previous study [18], with three new benchmarks added: matrix multiplication, 7-point stencil, and 2D-to-3D back projection. We briefly describe the benchmarks and their runtime parameters in Table 1, wherein the iteration count refers to the number of iterations of the kernel computation per program run. This iteration count serves to amortize the cost of initiating GPU offload, as some of the benchmark kernels take milliseconds to complete. We refer interested readers to the previous paper for a more thorough study of these benchmarks on both the Xeon CPU and Xeon Phi co-processors.

5.1 GPU vs CPU performance

All benchmarks presented here are compiled using HRC with GHC 7.6.1 as its frontend, the Intel C/C++ Compiler version 13.1.3.198 as its CPU backend, and Intel OpenCL SDK 3.0 with the latest version of Concord as the GPU backend. We only measure time spent in kernel computations when running the benchmarks, excluding time spent preparing inputs and producing output. We run these benchmarks on the following hardware:

Processor	Cores	Clock	Hyper-thread	Peak Perf.
HD4600 (GPU)	20	1.3GHz	No	432 GFLOPs
Core i7-4770	4	3.4GHz	Yes	435 GFLOPs
Xeon E5-4650	32	2.7GHz	No	2970 GFLOPs

Note that the above peak GFLOPs for each processor are calculated based on their hardware specifications, and hence should only be considered as theoretical limits. Although the HD4600 GPU has about the same peak GFLOPs as Core i7-4770, its Cores are of a much simpler design. We do not expect to get better performance by offloading to HD4600, but for the same performance we do expect less energy consumption through GPU offloading.

For the Core i7 CPU, we run each benchmark with 1, 4, and 8 OS threads with hyper-threading enabled. For the Xeon CPU, we run each benchmark with 1, 4, 8, 16, and 32 OS threads *without* hyper-threading. We do not include numbers for GHC-compiled benchmarks in part due to GHC not supporting SIMD vectorization on CPUs, as all but one of the benchmarks we present contain a kernel that can be vectorized by HRC. We refer our readers to the work by Petersen et al for details on the HRC vectorizer [19].

The relative kernel performance is given in Figures 2 and 3, where all numbers are normalized to a baseline speed corresponding to the execution time of a Core i7 CPU running a single thread without vectorization (which we call the *baseline*). All benchmarks are ordered in descending order of their max CPU speedups, and split into two figures for clarity on their relative scale. In the remainder of this section, we will speak of the relative performance of a particular configuration for a given benchmark as a single number, for example 4.7× means it is 4.7× faster than the baseline.

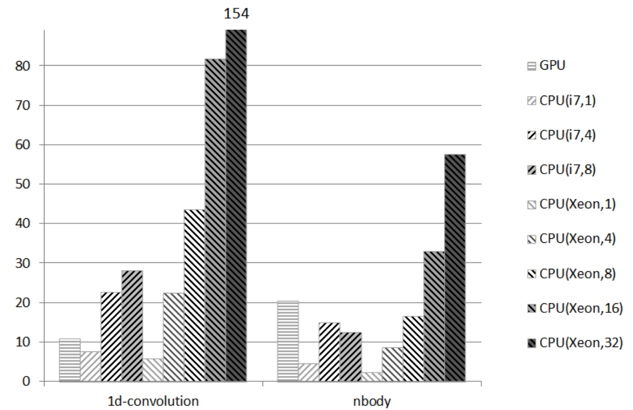

Figure 2. Kernel speedups relative to non-vectorized single-thread Core i7, Part 1/2 (bigger is better)

Figure 2 shows two high-performing benchmarks with very effective speedups on multi-core CPU. From left to right, we have:

1d-convolution This benchmark has a tight inner loop that iterates over the same input stencil array of 8192 elements, which can all fit into cache. 256-bit wide AVX2 vectorization gives very good speedup of almost 8× on CPU. Over-subscribing the Core i7 CPU with 8 hyper-threads also brings a little speedup. It scales linearly to the number of CPU cores, and at 32-core, Xeon (154×) significantly outperforms GPU (11×).

nbody This benchmark is computation intensive, since the entire input arrays all fit into cache, and over-subscribing the Core i7 CPU with 8 hyper-threads actually slows it down. Vectorization brings 4.7× speedup. It also scales linearly on CPU. Its GPU speedup is pretty good (20×), but still no match for 32-core Xeon (57×).

Figure 3 shows the remaining seven benchmarks. From left to right, we have:

matrix-mult This benchmark is both computation intensive and sensitive to cache misses, since the inner loop has to traverse both a row from one matrix, and a column from the other. The inner loop vectorizes on CPU, but the vectorization is ineffective due to strided loads being software-emulated instead of hardware-accelerated on AVX2 architecture. This actually gives a slow down on the Core i7 at 0.75× compared to non-vectorized version. Over-subscribing the Core i7 CPU with 8 hyper-threads boosts the performance quite significantly. Its GPU performance is good (21×), which significantly outperforms Core i7 (max at 4×), and is close to the performance of 32-core Xeon (21×).

Name	Parameter	iteration	Description
1d-convolution	3M pixels	10	1D convolution with 8192-point stencil
2d-convolution	3200×4000 pixels	100	2D convolution with a 5x5 stencil
7pt-stencil	256×256×160 pixels	100	3D convolution with 7-point stencil
backprojection	256×256×256 pixels	100	2D to 3D image projection
blackscholes	10M options	100	Black Scholes algorithm for put and call options
matrix-mult	2K×2K matrix	1	Matrix multiplication
nbody	200K bodies	1	Nbody simulation
treesearch	16-level tree, 20M inputs	50	Binary tree search
volume-rendering	1M input rays	1000	Volumetric rendering

Table 1. Haskell Repa benchmarks and their parameters

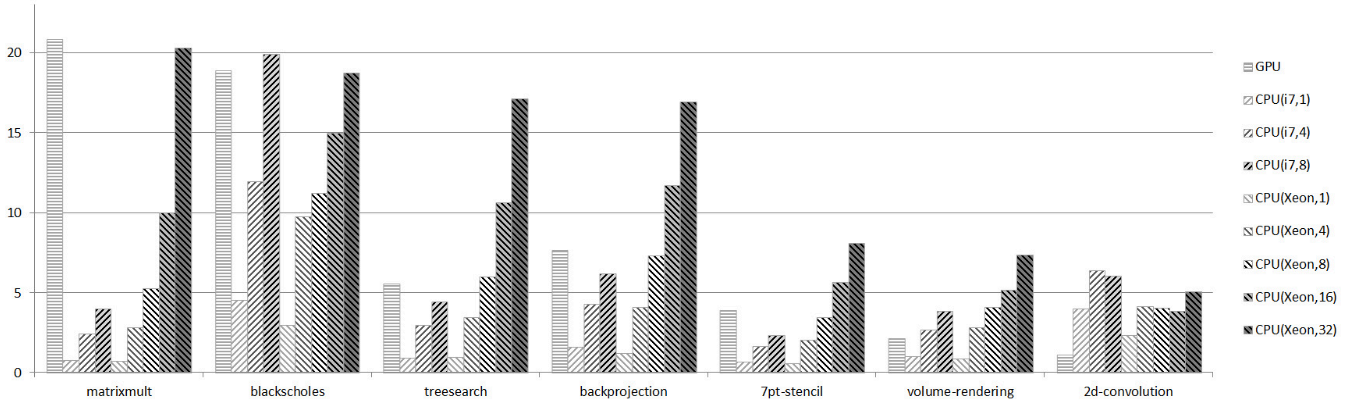

Figure 3. Kernel speedups relative to non-vectorized single-thread Core i7, Part 2/2 (bigger is better)

blackscholes This benchmark is memory bound, and is limited by the available memory bandwidth, which explains the significant speedup when over-subscribing the Core i7 CPU. As witnessed by the Xeon performance, it does not scale linearly to the number of cores due to memory bandwidth saturation. GPU performance is good at 19×, comparable to the Core i7 (20×) and the Xeon (19×).

treesearch This benchmark does little computation in its inner loop, and is sensitive to cache misses. It's also heavy on branches, which are translated to CMOV instruction on CPU by HRC. CPU vectorization is rather ineffective as a result: 0.9× on the Core i7 for single-thread, and 1.3× on the Xeon (not shown in the figure). Likewise, its GPU performance is not great either (5.5×). It scales linearly on the Xeon with a peak performance of 17× at 32-cores.

backprojection This benchmark is both computation intensive and sensitive to cache misses when indexing its input 2D array. It contains strided loads in the inner loop, which hampers vectorization on CPU (1.6× on single-thread Core i7). The GPU speedup is modest at 7.5×. It scales linearly on the Xeon with a peak performance of 17× at 32 cores.

7pt-stencil This benchmark is light on computation, and very sensitive to cache misses. The program is written as a naive traversal of its input 3D array because Repa does not yet provide domain-specific operators for 3D stencils. Vectorization is ineffective due to strided loads and cache misses, which brings a significant slowdown (0.65× on single-thread Core i7). Likewise, its GPU performance is poor (3.87×). It scales linearly on the Xeon with peak performance of 8× on 32-cores.

volume-rendering This benchmark has an irregular inner loop with two early loop exits, and HRC is unable to vectorize it. So for this one benchmark, the CPU numbers shown in Figure 3 are without vectorization. It scales linearly on the Xeon but the performance is not great (7× at 32 cores). Its GPU performance is also poor (2×).

2d-convolution This benchmark is memory bound, and sensitive to cache misses. Repa gives special treatment to 2D stencils using a cursored representation, which when compiled with HRC optimizations, is able to vectorize effectively on CPU (4× on single-thread Core i7). Its Xeon performance does not scale linearly, but appears to be limited by memory bandwidth. On the other hand, its GPU performance is very poor at only 1.1× speedup. We discuss this benchmark further in Section 5.3.

In all nine of the benchmarks we have studied, HD4600 either matches or beats Core i7 performance on six of them. It does reasonably well for for 1d-convolution, but fares poorly for the volume-rendering and 2d-convolution benchmarks. For volume-rendering, its performance is not bad given that the irregular kernel is more challenging for GPUs. In all benchmarks, the 32-core Xeon still out-performs the 20-core GPU, which is not surprising given that the Xeon is a server-grade CPU with a significantly higher theoretical peak performance, and it is a lot more expensive (and power hungry) too. We summarize by giving the geometric means of the best relative speedups of all benchmarks on the three architectures:

	HD4600 (GPU)	Core i7-4770	Xeon E5-4650
Geometric Mean	6.9	7.0	18.8

By showing that we can achieve expected performance on integrated GPU by offloading Haskell Repa programs, hopefully we have demonstrated the viability and promise of the native approach. With shared memory support already in place, our system can be further extended to combine both CPU and GPU to achieve even greater performance, something that we will consider in future work.

5.2 Performance Factors

Our benchmark study is focused on Haskell programs written using the Repa library. Despite all being generally categorized as data-parallel, these programs have different factors contributing or limiting their performances, and most of these factors are applicable to both CPUs and GPUs. We make the following important observations:

- **Thread-level or multi-core parallelism** usually brings linear scale-up for regular workloads under a shared memory model. This is true for most of our benchmarks, and especially effective for computation intensive ones such as 1d-convolution and nbody. A notable exception is that when the memory bandwidth is fully saturated, adding more threads or cores no longer helps, and sometimes may result in slowdowns. This is evident for the blackscholes and 2d-convolution benchmarks.
- Making good use of **memory locality** is crucial for applications to gain performance, Applications perform best when most its input data can fit into cache (as in nbody and 1d-convolution), but it is not always possible. Grouping data together for sequential access through AOS (Array of Struct) to SOA (Struct of Array) conversion (as in blackscholes), and cache blocking (as in treesearch and 2d-convolution) are two frequently used techniques to help memory locality.
- **Over-subscribing with hyper-threads** will help applications improve performance by amortizing the cost of memory I/O due to cache misses (as in matrix-mult and blackscholes), but will not help those that are already computation intensive (as in nbody).
- **SIMD vectorization** is an effective means to gain performance when the inner loop is regular and can be vectorized (as in 1d-convolution, nbody, blackscholes, and 2d-convolution). Although AVX2 (and older generation of SIMD hardware) currently does not support hardware-accelerated strided loads and some of our benchmarks (as in matrix-mult, backprojection, 7pt-stencil) suffer from it, future hardware (including the current generation of Xeon Phi) will no longer have this deficiency. Besides, algorithmic change can help to turn strided loads into sequential ones, as demonstrated by the modifications to Repa we did for 2D stencils [18].
- **Branch-avoidance** helps gaining performance on both CPU and GPU. The `CMOV` optimization we implemented in HRC enables SIMD vectorization for programs that have conditionals in their hot loops (as in blackscholes), although its effectiveness will be limited when branching cannot be avoided (as in volume-rendering) and when branching cost out-weighs computation cost (as in treesearch).

Most of the above mentioned techniques can and already have been implemented as part of an optimizing compiler and/or library. For example, HRC implements automatic SIMD vectorization. The Repa library makes it trivial to take advantage of thread-level (or multi-core level) parallelism, and its high-level interface enables automatic AOS-to-SOA conversion under-the-hood. It is also easy to implement cache blocking for Repa's abstract array representation (cursored, partitioned, etc.). Some applications such as treesearch require algorithmic changes to implement advanced cache blocking, and others require explicit strictness annotations or INLINE/NOINLINE pragmas in order to achieve desirable low-level compiled code. Hopefully we have shown that despite hardware differences these optimization techniques are applicable to the compilation of a data-parallel program written in a high-level language for both CPU and GPU targets, and when the performance is missing, which techniques could be effectively applied according to the characteristics of the application.

5.3 Haskell vs OpenCL Performance

No Haskell benchmarking is complete without comparing to native C performance, where 'C' symbolizes what is possible with low-level high-performance languages. Following the same spirit of our previous study, we believe it would be good to compare the following two categories:

1. idiomatic Haskell programs compiled by an optimizing compiler that targets integrated GPUs; and
2. the best-performing low-level programs written using either OpenCL or CUDA that targets the same hardware.

Ideally we would also like to compare Repa programs compiled using our native offload approach with Accelerate DSL programs compiled by its OpenCL backend for the same hardware. Unfortunately we were unable to complete this task at the time of writing due to the lack of a fully functioning OpenCL backend for Accelerate that targets Intel integrated GPUs.

Furthermore, due to our limited resources, we were unable to port all benchmarks to OpenCL and fine tune them for best performance. Therefore we choose to focus on a single benchmark, 2D convolution, which is one of the worst performing programs we have benchmarked on the GPU. We obtained a series of hand-tuned OpenCL programs for 2D convolution from [20], modified to work on the same inputs (e.g. only use one Float value per input pixel), and compiled by the same Intel OpenCL SDK 3.0 that HRC (via Concord) uses.

We summarize the set of 2D convolution benchmarks in Table 2. There are 6 OpenCL programs ranging from naive to optimized ones, and 2 Haskell programs that are actually source level identical, but with different Repa library implementations. When compiled by HRC, both programs produce an OpenCL kernel with its inner stencil loop completely unrolled, and with all stencil values specialized into the kernel code as constants. The haskell-row benchmark has the same implementation as the 2d-convolution benchmark that we previously presented in Figure 3.

An important difference between the Haskell and OpenCL benchmarks is that all OpenCL kernels (except ocl-linear) use a 2D index range consisting of both X and Y coordinates, whereas both of the Haskell kernels use a linear index as required by the `offload#` primitive. The ocl-linear benchmark is produced by hand-porting to OpenCL the kernel code obtained from compiling haskell-1 with HRC, and hence it uses the same linear index range that is used by the Haskell benchmarks.

One other difference is that the OpenCL implementations do not handle border conditions, while the Haskell ones do. This means the Haskell benchmarks are always going to have more overhead, despite that we have been cautious at modifying the Repa library to make sure that border values are computed on CPUs concurrently with the GPU kernel offload.

Figure 4 shows the relative performance of all 7 benchmarks for 2D convolution on HD4600, where the speedup is normalized to the same baseline we have considered previously, i.e., the Haskell 2d-convolution benchmark (same as haskell-row) running on a single thread on the Core i7 with vectorization turned off. This helps to compare the OpenCL performance on GPU with Haskell

Benchmark	Description
haskell-1	Haskell program with a kernel that computes only one output pixel
haskell-row	Haskell program with a kernel that computes an entire output row
ocl-naive	A native OpenCL implementation that reads 5x5 stencil from an array
ocl-const	Similar to ocl-naive, but specifies constant memory for stencil array
ocl-unrolled	Similar to naive-const, but with stencil loop unrolled
ocl-specialized	Similar to ocl-unrolled, but with stencil values specialized as constants
ocl-localmem	Similar to ocl-specialized, but uses a 20x20 local memory for cache blocking
ocl-linear	A OpenCL implementation ported from the generated kernel of haskell-1

Table 2. OpenCL and Haskell benchmarks for 2D convolution

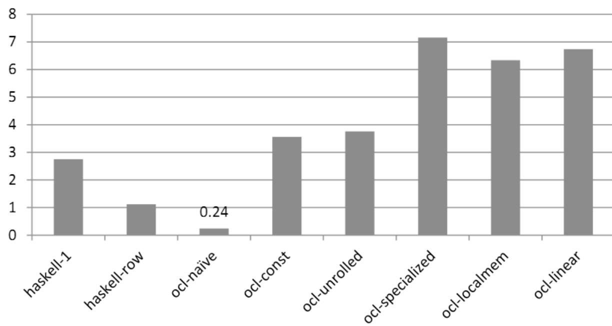

Figure 4. 2D convolution kernel speedups relative to Core i7 (bigger is better)

performance on CPU. All benchmarks compute 2D convolution repeatedly for 100 iterations over an input image of 3200x4000 pixels on a single run. We make the following observations based on the benchmark results:

- The best performing OpenCL program (ocl-specialized) is at 7.16×, which is only slightly higher than the best performing Haskell program on Core i7 (6.37× at 4 threads). This likely is another confirmation that this benchmark is memory bound.
- By declaring the input stencil array as constant memory, we immediately see a huge performance boost from ocl-naive (0.24×) to ocl-const (3.56×). This is the kind of low-level OpenCL optimization that compilers for high-level language can take advantage of. Eliminating memory reads of stencil values by specializing them (as in ocl-specialized, at 7.16×) doubles the performance further.
- Explicitly blocking using local memory does not seem to improve performance much. The overhead of filling a 20x20 cache (with a local group range of 16x16) and synchronizing at the end of the cache-fill actually gives a slight slowdown, when we compare ocl-localmem (6.34×) with ocl-specialized (7.16×). This is contrary to the original report by Reda [20]. We suspect this is due to hardware architecture differences.
- Despite ocl-linear having the same kernel source as haskell-1, it is more than twice as fast (ocl-linear at 6.74× vs haskell-1 at 2.75×). Besides border-condition handling, there are possibly other non-negligible overheads in the Haskell implementation. Having to allocate a new array to store the output image in between every convolution iteration could be one of them. Further analysis is required to better understand this.
- The performance difference between haskell-1 (2.75×) and haskell-row (1.12×) is also surprising. Lippmeier and Keller [11] carefully crafted the cursored representation and implementation of Repa arrays for parallel execution, where adjacent reads from source image can be shared during batch processing. For example, when we batch-compute 4 output pixels with a 5×5 stencil, only $5 \times (5+4-1) = 40$ memory loads are required (as in haskell-row), as compared to $5 \times 5 \times 4 = 100$ memory loads when each output pixel is computed in isolation (as in haskell-1). This was presented as a good optimization technique for CPUs. However, the same does not seem to apply to GPUs. As a result of the batched loads and unrolling and inlining, haskell-row is compiled to a very long kernel program. We suspect this overly unrolled loop body contributes to the added overhead. Again, further analysis is required to better understand this.

In conclusion, there are still great opportunities for compiler and library writers to borrow some of the low-level GPU and OpenCL optimization techniques to improve the compilation of a high-level language. Complementing the discussion in Section 5.2, with the 2D convolution benchmark study, we have shown that not all techniques for optimizing CPU programs are as effective when applied to GPU. Sometimes the performance discrepancy must be scrutinized on a case by case basis, and it requires deep knowledge of the low-level toolchain and hardware architecture before one can start to understand it. We consider this as part of our future work.

6. Related Work and Conclusion

Due to similarities between the APIs of Repa and Accelerate, we are interested in undertaking a detailed comparison between our native approach and the DSL approach taken by Accelerate. Unfortunately, Accelerate does not have a fully functional OpenCL backend to compile and run the benchmarks in Table 1, and moreover other DSL based frameworks for GPU programming (such as Obsidian [22] and Nikola [14]) only target CUDA. As a result, a direct performance comparison upon the same hardware is not possible. Obsidian's DSL uses a lower-level abstraction that exposes more hardware details by only targeting one-dimensional arrays of limited size. At the high level, Nikola is similar to Accelerate, but with more targeted optimizations only supporting first-order array functions. It also makes use of meta-programming to statically compile DSL programs when the host program is being compiled, avoiding the overhead of having to compile them at runtime. More generally, Gaster and Morris [8] implement a direct embedding of OpenCL in GHC, offering a way for low-level GPU programming in a high-level language for applications outside the domain of data-parallel array computation.

Furthermore, we intend to study irregular workloads, as there is nothing preventing us from using the `offload#` primitive to compile programs beyond those written using the Repa library, so long as the same set of limitations are respected. In particular, the Concord compiler [2] that we use in this work was *designed* to

target irregular workloads. By focusing only on programs written in Repa, we are not fully exercising the power of Concord. There are still many issues surrounding the native offload of arbitrary Haskell functions, however, especially considering the lack of garbage collector and thunk-evaluation support for GPU runtimes. It remains to be seen whether a compromise exists for capturing irregular GPU workloads by way of an abstraction between array-based data-parallel programming and the call-by-need semantics of Haskell.

In conclusion, this work presents a technique for directly offloading computations written using the Haskell Repa array library to integrated GPUs via OpenCL without requiring extensive API changes. We support the latest shared virtual memory model between the host and associated OpenCL devices, avoiding unnecessary data movement between them. The Repa library provides just the right kind of data-parallel abstraction our purposes, and by implementing a GPU backend in HRC, we can compile most programs written using Repa down to a strict kernel function comprising straight loop code, which is ideal for execution on GPUs. We demonstrated the feasibility of the native offload approach by presenting a detailed analysis of nine benchmarks contrasting the performance on GPU and two CPUs.

References

[1] T. A. Anderson, N. Glew, P. Guo, B. T. Lewis, W. Liu, Z. Liu, L. Petersen, M. Rajagopalan, J. M. Stichnoth, G. Wu, and D. Zhang. Pillar: A parallel implementation language. In *LCPC*, pages 141–155, 2007.

[2] R. Barik, R. Kaleem, D. Majeti, B. T. Lewis, T. Shpeisman, C. Hu, Y. Ni, and A.-R. Adl-Tabatabai. Efficient mapping of irregular C++ applications to integrated GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 33:33–33:43, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2670-4. .

[3] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4566-0. .

[4] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore GPUs. In M. Carro and J. H. Reppy, editors, *DAMP*, pages 3–14. ACM, 2011. ISBN 978-1-4503-0486-3.

[5] R. Clifton-Everest, T. L. McDonell, M. M. T. Chakravarty, and G. Keller. Embedding foreign code. In M. Flatt and H.-F. Guo, editors, *PADL*, volume 8324 of *Lecture Notes in Computer Science*, pages 136–151. Springer, 2014. ISBN 978-3-319-04131-5.

[6] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, pages 162–174, New York, NY, USA, 2001. ACM. ISBN 1-58113-388-X. .

[7] M. Fluet and S. Weeks. Contification using dominators. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 2–13, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0. .

[8] B. R. Gaster and J. G. Morris. Embedding OpenCL in GHC Haskell. In *MULTIPROG'13 Workshop On Programmability Issues For Heterogeneous Multicores*, 2013.

[9] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. L. P. Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In P. Hudak and S. Weirich, editors, *ICFP*, pages 261–272. ACM, 2010. ISBN 978-1-60558-794-3.

[10] Khronos Group. The OpenCL specification, version: 2.0, 2013. See `https://www.khronos.org/opencl/`.

[11] B. Lippmeier and G. Keller. Efficient parallel stencil convolution in Haskell. In K. Claessen, editor, *Haskell*, pages 59–70. ACM, 2011. ISBN 978-1-4503-0860-1.

[12] B. Lippmeier, M. M. T. Chakravarty, G. Keller, and S. L. P. Jones. Guiding parallel array fusion with indexed types. In J. Voigtländer, editor, *Haskell*, pages 25–36. ACM, 2012. ISBN 978-1-4503-1574-6.

[13] H. Liu, N. Glew, L. Petersen, and T. A. Anderson. The Intel Labs Haskell research compiler. In *Haskell Symposium*, pages 105–116, Boston, Massachusetts, USA, 2013. ACM. ISBN 978-1-4503-2383-3.

[14] G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. In J. Gibbons, editor, *Haskell*, pages 67–78. ACM, 2010. ISBN 978-1-4503-0252-4.

[15] T. L. McDonell, M. M. T. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional GPU programs. In G. Morrisett and T. Uustalu, editors, *ICFP*, pages 49–60. ACM, 2013. ISBN 978-1-4503-2326-0.

[16] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, Mar. 2008. ISSN 1542-7730. .

[17] L. Petersen and N. Glew. GC-safe interprocedural unboxing. In *Compiler Construction*, pages 165–184, Tallinn, Estonia, 2012. Springer-Verlag.

[18] L. Petersen, T. A. Anderson, H. Liu, and N. Glew. Measuring the Haskell gap. In *Post Symposium Submission to The 25th International Symposium on Implementation and Application of Functional Languages*, Aug. 2013.

[19] L. Petersen, D. Orchard, and N. Glew. Automatic SIMD vectorization for Haskell. In *ICFP*, pages 25–36, Boston, Massachusetts, USA, 2013. ACM. ISBN 978-1-4503-2326-0.

[20] K. Reda. A study of OpenCL image convolution optimization, April 2012. URL `http://www.evl.uic.edu/kreda/gpu/image-convolution`.

[21] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM. .

[22] J. Svensson, M. Sheeran, and K. Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In S.-B. Scholz and O. Chitil, editors, *IFL*, volume 5836 of *Lecture Notes in Computer Science*, pages 156–173. Springer, 2008. ISBN 978-3-642-24451-3.