Native Offload of Haskell Repa Programs to GPGPU

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 Graphics Processing Units (GPGPUs) are becoming increasingly commonplace, and demand novel programming models to account for their radically different architecture. For the most part, existing approaches to programming GPGPUs within a high-level programming language choose to embed a domain specific language (DSL) within a host metalanguage and implement a compiler mapping programs written within said DSL to code in low-level languages such as OpenCL or CUDA. We question this design choice, and argue that by directly implementing a GPGPU offload primitive as part of a general-purpose language compiler, we gain access to a substantial number of existing optimization techniques without having to reimplement them in a DSL compiler. In this paper we describe the structure of our prototypical treatment of this research direction, demonstrating the applicability of our approach by showing how to bridge between APIs by extending the Repa library of Haskell with an offload primitive, and detailing an experimental implementation of our approach within the Intel Labs Haskell Research Compiler. We also provide a detailed study of a set of nine benchmarks, by compiling them to both GPU and two distinct CPUs and comparing their performance.

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 graphics processing units (GPGPU) are becoming more and more accessible to ordinary consumers. 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 [8] and CUDA [14] are two such popular frameworks that come with restricted C-based languages which address this need, and hardware vendors typically provide the necessary toolchain to compile OpenCL or CUDA programs to their devices. Programming for a combination of mixed processing units, such as CPUs, GPUs, and FPGAs (amongst others) is often referred to as *heterogeneous computing*. Whilst CUDA is only available for NVIDIA GPUs, OpenCL is an open standard providing support for programming devices from multiple hardware vendors.

Heterogeneous computing is a complex area of research due to the existence of and need to accommodate multiple different programming models. Despite being designed to target heterogeneous platforms, OpenCL is still a low-level programming language that requires programmers to write a range of boilerplate code addressing topics such as device setup and manual memory management. In the past few years, however, a number of programming frameworks have emerged for higher-level languages with the goal of supporting GPU programming (and heterogeneous computing in general) *without* requiring familiarity with low-level languages such as OpenCL or CUDA. 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, it translates the embedded code into OpenCL or CUDA and then compiles and offloads that code to the GPU. Accelerate [4] and Obsidian [19] are two examples of this approach towards programming GPUs using the functional language Haskell. Delite [3] is another such approach allowing not only the compilation and execution of DSLs in a heterogeneous setting, but also a flexible technique for building DSLs using a multi-staged tool chain built on top of the Scala language.

If we view OpenCL as providing a hardware abstraction over data-parallel hardware architectures, then programming GPUs through DSLs raises the level of abstraction even higher by hiding the hardware interface almost entirely. The obvious benefit of taking this approach is that programmers can continue to write programs using higher-level constructs and operators, and the DSL compiler shoulders the responsibility of translating them to target GPUs. Additionally, these DSLs are carefully crafted around a limited set of constructs and operators such that only those programs suitable for GPU execution are expressible. This kind of self-imposed design choice not only ensures program runtime safety, but also saves DSL implementers from the task of writing a full-blown general-purpose compiler targeting GPUs, which is often infeasible given hardware limitations.

On the other hand, the DSL approach has drawbacks of its own. To understand why, consider two similar parallel programming libraries in Haskell, Repa [7] and Accelerate [4]. Repa is a Haskell library enabling high-performance array computation on multi-core CPUs, and Accelerate is an embedded array language targeting GPUs. These two share similar high-level APIs regarding multi-dimensional and shape polymorphic parallel arrays (no coincidence—the same team designed and implemented both), yet differ in ways beyond those associated with 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 many of the existing features that are already available in the host language compiler in order to handle ordinary Haskell expressions, function composition and so on. The first versions of Accelerate indeed had performance issues due to missing general optimizations, which are not GPU specific [13].
- The DSL approach needs a special mechanism to interface with foreign code since they cannot directly use the FFI features already available to the host language [5].
- Data transfers between the host program and DSL program must be explicitly handled at runtime, as they operate in separate address spaces. With the increasing dominance of integrated GPU and recent introduction of Shared Virtual Memory (SVM) to OpenCL 2.0 standards, this becomes an unnecessary burden.

To the best of our knowledge, this alternative approach of compiling a restricted subset of the host language itself to OpenCL or CUDA and thereafter to GPU has not been tried, at least not for any functional programming languages. With this in mind, we have implemented a proof of concept system for compiling a restricted subset of Haskell to Intel's integrated GPU. Our system is as yet incomplete, with a number of short-cuts in place to quickly prototype our concept and produce preliminary results. Despite this, we strongly believe that the results are interesting and show that this alternative approach is viable. The majority of the shortcomings in the work we present here are engineering related, requiring more time than effort to address; however, some are more serious, and as such we discuss them below. One point in particular is worth emphasizing—we are targeting an *integrated* GPU. A number of CPU vendors have in recent years started making processors with both a CPU and a GPU on the same die, and these processors usually share at least the last-level cache and often also share a coherence domain, and can be considered to have a shared memory system. This is in contrast to discrete GPUs, which typically communicate (incoherently) with the CPU across a limited interconnect, and as such should be treated as if they possess a *separate* memory system. This difference greatly simplifies compiling to GPU, and should be borne in mind throughout.

With those caveats, 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) [11] backend.
3. We integrate our solution with the Concord compiler (a C++ based heterogeneous computing framework for integrated GPUs that compiles to OpenCL) [2], and the latest Intel OpenCL SDK to take advantage of the new SVM hardware features available on the Intel Broadwell platform.
4. We demonstrate the effectiveness of this native offloading 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 [10].

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 manifested (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 manifested only when they are 'forced', 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 propose extending 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—which we will call the *kernel instance*—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 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. Our intention (not fully implemented) is to detect and reject at compile time any computation that ill-suited for compilation to GPU. To further understand the issues involved, we first discuss OpenCL and, more generally, the GPU computing paradigm.

2.2 Overview of OpenCL

OpenCL [8] 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, call 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 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 that we call 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 amongst the work items of a given work group. This local memory is limited is 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. The ubiquity of 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 in a heterogeneous programming model.

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 and offloading 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). Putting to the side device setup and error handling issues for the time being, we can express 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, since the function we offload could be arbitrarily constructed. 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 should 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 behaviour, 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.

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 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 ()
```

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 manifested. The actual computation 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 function that computes 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. To counter this, we specify the overall work-range to have the same value as the destination array, and

defer the dispatching of jobs to available compute units to the underlining OpenCL computation.

In a similar manner we implement `loadG` for other array representations (such as Cursored, Partitioned, etc). 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. A Heterogeneous Backend for HRC

We implement `offload#` as part of the Intel Labs Haskell Research Compiler backend. HRC uses GHC as a front-end and intercepts the external Core before performing whole-program optimization. The HRC also implements a multitude of optimization passes based on a strict SSA-style internal representation called MIL, and generates code in an extension of C called *Pillar* [1], which is then transformed to standard C code and compiled with either the Intel C compiler or GCC. 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 [11, 15], 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 called the "Haskell Gap" using a set of Repa benchmark programs compiled by HRC for both the Xeon CPU and Xeon-Phi Co-processor [16]. We re-use the same set of benchmarks in this study to compare performances.

Because HRC uses GHC as its frontend, we first have to modify GHC to add 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 the HRC compiler as input. Since HRC is a whole-program compiler, when compiling a source program it will also pull in Core code from all libraries identified by a dependency graph, and amongst them is our modified Repa library where we can find the definition of `computeG` in terms of `offload#`.

The overall HRC compilation pipeline works as follows. GHC Core is first translated into a strict IR based on administrative normal form (ANormStrict), then run through an optimizer before being closure-converted to MIL. After the program is converted to MIL, it goes through a number of control-flow and data-flow based optimizations. What makes MIL unique is that it combines a low-level CFG-based block structure with a high-level object-based memory model. Eventually, the MIL code is translated into Pillar and passed on to the Pillar-to-C converter, before being compiled – and linked – by a C compiler.

There are several issues we must consider in order to both implement `offload#` in HRC and support it at runtime:

1. We need to implement runtime support for setting up OpenCL devices, calling OpenCL kernels, and integrating SVM into the garbage collector.
2. 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 could be arbitrarily composed, which can become a problem for the code generator, i.e., we cannot generate code from it if we do not know where the code is at compile time.
3. If we *are* able to locate the function body that is passed to `offload#`, it could contain arbitrary code that allocates memory, evaluates thunks, makes calls to other functions, and so on. Therefore we either have to implement runtime support for doing these things on GPU, or enforce restrictions on the kind of code that we expect to convert to OpenCL.

To deal with this first issue, we choose to integrate with 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.

The latter two issues are related to program safety and correctness, and we must assess them carefully when taking the native offload approach. We present the set of design choices and concessions we have made in Table 1, and discuss their implications below:

1. We do not attempt to offload arbitrary Haskell functions, but rather only computations that are composed of functions and combinators drawn from the Repa library. The authors of the Repa library have already taken great effort to instruct GHC to aggressively inline and fuse all array functions. We note that whilst we explicitly preclude the calling of non-recursive function calls within the kernel in this presentation of our work, implementing a pass to inline their definitions is a technically simple matter. As a result, more often than not what we get by compiling Repa programs that use the `computeG` API is a known static function passed to `offload#`. As we shall see in Section 5, this is true for all benchmarks tested, without any modifications to their source.
2. HRC is good at turning tail-recursive calls – and even mutual ones – into local loops, so typical Repa computations in the offload kernel are not recursive. Calls to other functions are possible, but currently we do not transitively translate all functions that are called, but instead leave them to the compiler to resolve. For the most part, functions concerning operations such as arithmetic are easily resolved by the Concord/OpenCL compiler.
3. Unlike Accelerate or other DSL solutions, Repa is a native Haskell library and thus must adhere to its call-by-need semantics. Although most Repa computations are not lazy by design, strictness analysis alone does not handle all cases, and sometimes one must still modify the program to make sure no residual thunk evaluations remain inside the offload kernel. Static analysis is enough to catch these and report them at compile time, however.
4. One early design choice was to make use of SVM to ease memory management, but the interaction of the offloaded kernel with the garbage collector can still prove difficult. For instance, only memory allocated in the SVM region is visible to the kernel, but it is very difficult to know beforehand if a piece of memory will be accessed by the kernel when allocating it. Therefore, we choose to avoid this problem entirely by instructing the garbage collector to *always* allocate in the SVM region. Unfortunately, the amount of data which we can store in SVM is restricted by the specification of OpenCL.

Property	Scope	Checked at	Note
Kernel function pointer	Only static functions	Compile time	This is usually the case for Repa programs
Non-recursive function calls	Not supported	Compile time	Can be inlined into the kernel definition
Recursion	Not supported	Compile time	Tail-calls are already turned into loops
Foreign calls	Limited	Compile and link time	Only safe foreign calls to OpenCL functions are allowed
Shared memory	Full support		By making GC always allocate in SVM space
Garbage collection	Limited	Run time	SVM has a maximum heap size of 400MB
Memory allocation	Not supported	Compile time	No GC runtime for OpenCL devices
Thunk-eval	Not supported	Compile time	Strictness annotations to source code may be required

Table 1. Design choices: implementation of offload

5. Always allocating inside SVM does not solve all memory issues, however, as dynamic heap allocation inside the kernel presents another challenge. We must either callback into the main program on CPU, or implement part of the garbage collector as GPU kernels. Since such allocations are not data-parallel to begin with, we choose to simply rule out *all* kernel allocations, and report them as errors at compile time.
6. Due to the relaxed memory consistency guarantee during GPU kernel execution, we must temporarily halt garbage collection whilst the GPU is interacting with memory.

With these decisions, it is then straightforward to implement a first iteration of 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 static 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. Finally, 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 representation, heavy use of the `INLINE` pragma and GHC rules to aid in fusing 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 the 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). On the other hand, 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, but at the same time cannot make good use of the optimizations already implemented in the Haskell compiler, and as such the approach is reduced to a matter of taste. In this section we explore ways to both help users write Repa programs appropriate for GPU offloading, and aid compilers in better optimizing the generated code.

4.1 Strictness

As discussed previously, the HRC performs static checks to ensure that kernel functions contain no 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 [10]:

```
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, which is essentially the lambda expression passed to `R.map`.

To alleviate this problem, Lippmeier, et al [10] 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;
   }
   ...
}
```

Suppose this kernel is offloaded to a number of PEs for execution. As all PEs run exactly the same instruction, if some PEs enter branch `A` (i.e. condition `C` is satisfied), other PEs *cannot* enter branch `B` simultaneously, as to do so would violate the SIMD model. These latter PEs must now either wait, or enter `A` but discard the effect of running it. As a consequence, all compute PEs execute both branches, 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 size of code in both of them, with most of the instructions in each branches almost identical up to alpha-renaming of variables. This is not a problem unique to GPU offloading, but also an issue previously encountered when implementing SIMD vectorization for CPUs into HRC.

To reduce the impact of situations such as these, we implement an algorithm that attempts to merge conditional branches using a new conditional move (`CMOV`) primitive introduced to 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 is strict we require that `a`, `b` and `c` are either constants or variables, the values of which 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. For simplicity, we impose the following pre-conditions for the branch merge optimization:

- The branch is a binary switch over a boolean variable. Binary switches over non-boolean values can be easily converted into boolean ones.
- Both branches contain only one instruction block, and these blocks 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.

The actual merge function takes the two blocks of each branch as input, and produces a single output block as the result. If the algorithm opts to 'bail out' prior to completion, no output is produced and the input code stays unchanged. The general algorithm can be described as follows:

1. Pop one instruction from the head of each block, and check if they are alpha-equivalent by looking up a dictionary of equivalent variable names.
2. If these instructions are equivalent, we produce a new instruction in the output block after alpha-renaming, and loop back to Step 1.
3. If these instructions differ only in their arguments, we identify each pair of arguments that are different, and assert their equivalence by making a new variable and map both arguments to it in the equivalence dictionary. Then we produce a CMOV instruction for each equivalent variable to initialize them based on the condition variable. As a result, the two instructions are now equivalent, and we loop back to Step 2.
4. If the instructions are indeed different, we count this as a mismatch. We check if one of them can be appended to the end of the output block by checking whether it has no side-effects. If this is the case, the instruction is safe to run regardless of control flow.
5. If the check in Step 4 fails, we bail out of the algorithm. If it succeeds, we push the side-effect free instruction to the end of output block, push the other instruction back to the top of its input block, and loop back to Step 1.

The above algorithm terminates successfully if we process all instructions from both input blocks. We also set a threshold of how many mismatches are allowed, and bail out if this threshold is exceeded. This is a heuristic to ensure that we don't over-merge branches that are indeed substantianlly different from each other. Ideally, we should check instruction equivalence modulo instruction re-ordering when permitted by side-effect constraints, but we choose not to implement this and adhere to the existing instruction order in the input blocks. The actual equivalence check is more complicated than we describe here due to the variety of instruction types, but we omit the details for simplicity.

4.3 Better Index Calculation

The Repa library internally represents all arrays as a single continuous memory block. In order to read from a multi-dimensional array, we first convert a N-dimensional index into a row-major linear index, and read the data from the underlining memory block using this linear index. Usually this is not a problem, but if we recall the definition of `loadG` in Section 2, we assume that the range argument passed to the `offload#` primitive is the same as the linear size of the underlining array block, and the kernel function also expects a linear index as its first parameter. Consequently, to obtain the value to be written to the output array, we first convert the linear index to the appropriate shape specified by the abstract representation using `fromIndex`. An issue arises if the linear index is used to both read from and write to multi-dimensional arrays of the same size, as it is first converted to a multi-dimensional one and then immediately converted back.

The problem is evident when we compile the `map2` example seen in Section 2 using HRC. The OpenCL kernel code is given below:

```
kernel void map2(global const int * restrict a,
                 global       int * restrict b)
{
   int idx0 = get_global_id(0):
   int i = idx0 / 5;
   int j = idx0 % 10;
   int idx1 = i * 5 + j;
   int x = a[idx1] * 4;
   b[idx0] = x * x;
}
```

We can immediately tell that `idx1` is equal to `idx0`. Furthermore, there are 4 redundant arithmetic operations within a kernel that only comprises 2 multiplications. We might expect a quality OpenCL compiler to be capable of reasoning about the equality of the two indices and optimize away redundant arithmetic operations. In practice, we still find it beneficial to add a *divMod* optimization to HRC as part of the MIL optimizer, as being able to derive index equality is a powerful result that may trigger optimizations that may not be viable otherwise. In Section 5 we will inspect some benchmarks that demonstrate the effectiveness of this simple optimization.

4.4 Cache Locality

A topic related to multi-dimensional array representations is how to best make use of cache locality, avoiding memory latencies incurred by cache misses. 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 [9]. This proves to be quite effective in practice.

Cache locality optimization in general applies to both CPUs and GPUs, however there are also GPU specific techniques. For example, we know that PEs are grouped into compute units on device, and that PEs in the unit can share local memory. Therefore, instead of using a linear global ID, it is best to address PEs using a scheme that is close to their physical structure, and arrange memory access pattern in the kernel so that all PEs internal to the same unit work on a contiguous memory region, and two units work on different memory regions. Unfortunately we cannot make use of this kind of cache locality optimization because our `offload#` primitive assumes a range linear in its type, and once a multi-dimensional index is collapsed to a linear one, its structural information is lost.

5. Benchmark Results

We measure the performance of native offloading to GPU using a set of benchmarks written using the Haskell Repa library. The majority of these benchmarks originate from the "Haskell Gap" study [16], 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 2, wherein the iteration count refers to the number of iterations of the kernel function per program run. This iteration count serves to amortize the cost of compilation and initiating GPU offload, as some of the benchmark kernels take milliseconds to complete. We refer interested readers to the Haskell Gap 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	Hyperthread	Peak Perf.
HD4600 (GPU)	20	1.3GHz	N.A.	432 GFLOPs
Core i7-4770	4	3.4GHz	Yes	435 GFLOPs
Xeon E5-4650	32	2,7GHz	No	2970 GFLOPs

It must be noted that the above peak GFLOPs for each processor are calculated based on their hardware specifications, and hence shall 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 conumption through GPU offloading.

For the Core i7, we run each benchmark with 1, 4 and 8 OS threads with hyper-threading enabled. For the Xeon, 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 vectorizations on CPUs, as all but one 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 [17].

The relative kernel performance is given in Figure 1 and Figure 2, 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 the ascending order of their max CPU performances, 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 $4.7\times$ means it's $4.7\times$ faster than the baseline.

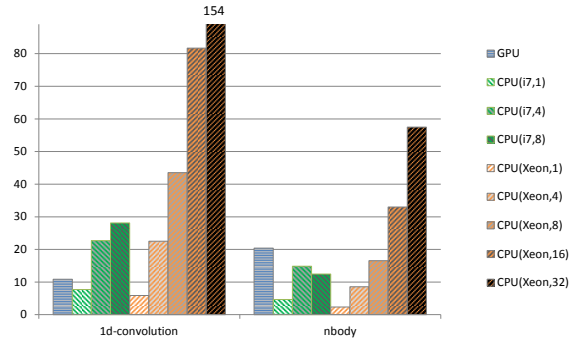

Figure 1. Kernel Speedups Relative to Core i7 (bigger is better)

Figure 1 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 $8\times$. 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 ($154\times$) significantly outperforms GPU ($11\times$).

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

Figure 2 shows the rest 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 Core i7 at $0.75\times$ 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 ($21\times$), which significantly outperforms Core i7 (max at $4\times$), and is close to the performance of 32-core Xeon ($21\times$).

blackscholes This benchmark is memory bound, and is limited by the available memory bandwidth, which explains the significant speedup brought by 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 I/O saturation. GPU performance is good at $19\times$, comparable to the Core i7 ($20\times$) and the Xeon ($19\times$), which is another indication of memory bandwidth bound application.

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

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 2. Benchmarks and their parameters

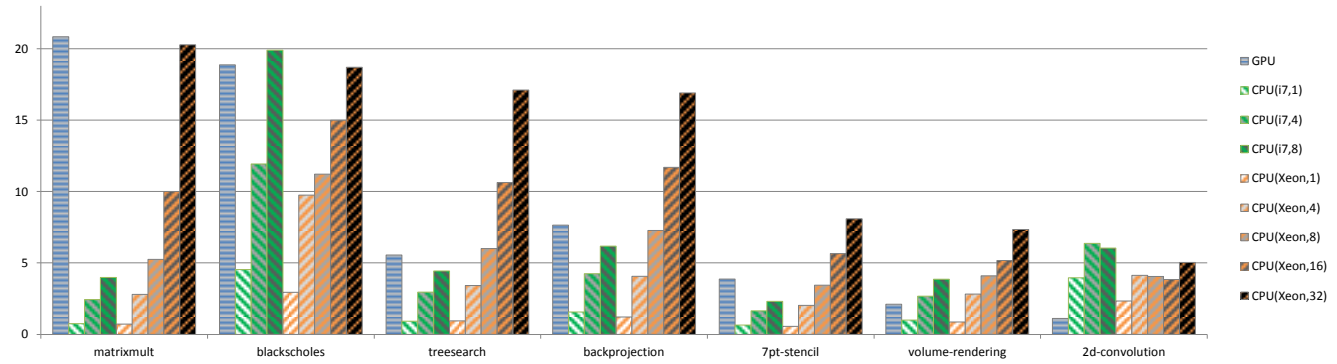

Figure 2. Kernel Speedups Relative to Core i7 (bigger is better)

HRC. CPU vectorization is rather ineffective as a result: 0.9× on Core i7 for single-thread, and 1.3× on Xeon (not shown in the figure). Likewise, its GPU performance is not great either (5.5×). It scales linearly on 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 Xeon with a peek 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 stencil. 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 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 2 are without vectorization. It scales linearly on 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 GPU. 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 too. We summarize by giving the geometric means of the best relative performance 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, which is left to future work.

5.2 Performance Factors

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

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 o 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 use 20x20 local memory to cache inputs
ocl-linear	A OpenCL implementation ported from the generated kernel of haskell-1

Table 3. OpenCL and Haskell Benchmarks for 2D Convolution

- **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 will no longer help with the speed, and sometimes may result in slowdowns. This is evident for the blackscholes and 2d-convolution benchmark.
- 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 (e.g., nbody, 1d-convolution), but it is not always possible. Grouping data together for sequential access (e.g., AOS to SOA conversion as in blackscholes), and cache blocking are two frequently used techniques (treesearch, 2d-convolution) to help memory locality.
- **Over-subscribing with hyper-threads** will help applications to improve performance by amortizing the cost of memory I/O due to cache misses (e.g., matrix-mult and blackscholes), but will not help those that are already computation intensive (e.g., nbody).
- **SIMD vectorization** is an effective means to gain performance when the inner loop is regular and can be vectorized (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 (matrix-mult, backprojection, 7pt-stencil) suffer from it, future hardware (including the current generation of Xeon Phi) will no longer have this deficiency. Besides, algorithm change can help to turn strided loads into sequential ones, as demonstrated by the modifications to Repa we did for 2D stencils [16].
- **Branch-avoidance** helps SIMD vectorization and gaining performance on both CPU and GPU. The CMOV optimization we implemented in HRC helps to enable vectorization for programs that have conditionals in its hot loop (blackscholes), although its effectiveness is limited when branching cannot be avoided (volume-rendering) and when branching cost outweighs computation cost (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 change to implement advanced cache blocking, and others require explicit strictness annotation or INLINE/NOINLINE pragmas in order to achieve desirable low-level compiled outputs. 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 technique could be effectively applied given 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 the "Haskell Gap" study, we believe it makes a very good comparison between the following:

1. Idiomatic Haskell program compiled by an optimizing compiler that targets GPGPUs;
2. the best-performing low-level program 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 an OpenCL Accelerate 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 graphics cards.

Furthermore, due to our limited resources, we were unable to port all benchmarks to OpenCL and hand optimize them for best performance. Therefore we choose to focus on a single benchmark, 2D convolution, which is one of the worst performing benchmarks on GPU. We obtained a sequence of hand-tuned OpenCL programs for 2D convolution from [18], modified to work on the same inputs, 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 3. We have 6 OpenCL programs ranging from naive to optimized ones, and 2 Haskell programs, which are actually the same source program that we have considered before, but with different Repa library implementations. When compiled by HRC, both program would produce an OpenCL kernel with inner stencil loop completely unrolled, and all stencil values already specialized as constants. The haskell-row benchmark here is the same 2d-convolution benchmark presented previously in Figure 2.

One important difference between Haskell and OpenCL implementations is that all OpenCL kernels (except ocl-linear) use a 2D index consisting of both X and Y coordinates, while the Haskell kernels use a linear index required by the `offload#` primitive. All OpenCL kernels compute only one output pixel. The ocl-linear program is produced by hand-porting the C kernel code from compil-

ing haskell-1 with HRC to OpenCL, and hence it also uses a linear index.

Another difference is that our OpenCL implementations do not handle border conditions at all, while the Haskell ones do.

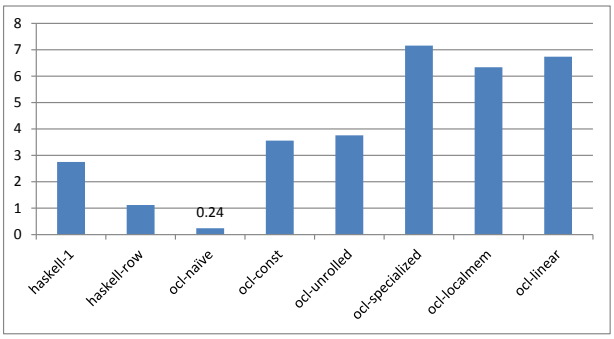

Figure 3. 2D Convolution Kernel Speedups Relative to Core i7 (bigger is better)

Figure 3 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 benchmarked running a single thread on Core i7 with vectorization turned off. This helps to compare OpenCL performance on GPU with Haskell performance on CPU. All benchmarks are run for 100 iterations on an input image of 3200x4000 pixels. We make the following observations on this set of results:

- The best performing OpenCL program (ocl-specialized) is at 7.16×, which is slightly higher than the best performing Haskell program on Core i7 (6.37× at 4 threads). This further confirms that this benchmark is memory bound.
- By declaring 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 should try to take advantage of. By further eliminating memory reads of stencil value, ocl-specialized (7.16×) is able to double the performance.
- Explicitly doing a cache blocking using local memory doesn't seem to give much advantage. The overhead of filling a 20x20 cache (with a local group of 16x16) and synchronization at the end of cache-fill actually gives a slight slowdown, as indicated by ocl-localmem (6.34×) and ocl-specialized (7.16×), This is contrary to the original report by Reda [18]. We suspect this is due to hardware architecture differences.
- There is virtually no difference between the kernels of ocl-linear (6.74×) and of haskell-1 (2.75×), and yet there is more than twice a performance gap. Besides border handling, there are possibly other non-negligible overheads in the Haskell implementation. Having to allocate a new array in between the convolution iterations 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×) also come as a surprise. Lippmeier and Keller carefully designed the cursored representation of Repa arrays for parallel execution, where adjacent reads from source image can be shared when computing 4 output pixels at a time. When we batch compute 4 output pixels with a 5x5 stencil, it only requires 5x(5+4-1)=40 memory loads (as in haskell-row). Compared to 5x5x4=100 memory loads when each output pixel is computed in isolation (as in haskell-1), it was presented as a good optimization technique for CPUs. However, the same doesn't seem to apply to GPUs. As a result of the batched load and inlining, haskell-row produces a very long kernel code. We suspect the added overhead is due to overly unrolled loop body. Again, further analysis is required to better understand this.

As a conclusion, there apparently is great opportunity for compiler and library writers to borrow some of the low-level GPU and OpenCL optimization techniques. As a complement to the discussion in Section 5.2, 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 low-level toolchain and hardware specifics before one can begin to understand them. We consider this as part of our future work.

6. Related and Future Work

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 2, and moreover other DSL based frameworks for GPGPU programming (such as Obsidian [19] and Nikola [12]) only target CUDA. As a result, a direct performance comparison upon the same hardware is not possible. Obsidian's DSL uses a lower-lever abstraction that exposes more details of the hardware hierarchy by only targeting one-dimensional arrays of limited size. At the high level, Nikola is similar to Accelerate, with more targeted optimizations only supporting first-order array functions, and also makes use of meta-programming to allow DSL programs to be compiled statically when the host program is being compiled, avoiding the overhead of having to compile them at runtime. More generally, Gaster and Morris [6] implement a direct embedding of OpenCL in GHC, offering a way to program GPGPUs in a high-level language for applications outside the domain of data-parallel array programming.

Furthermore, we intend to study irregular workloads, as there is nothing constraining us from using the `offload#` primitive to compile programs beyond those written using the Repa library. In particular, the Concord compiler [2] which we use in this work has been *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 support and thunk-evaluation 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 in the Repa library of Haskell to GPGPUs 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 required, and by implementing a GPU backend in the Haskell Research Compiler, most programs written using Repa can be compiled down to a strict kernel function comprising straight loop code, which is ideal for execution on GPGPUs. We demonstrate the feasibility of the native offload approach by presenting a detailed analysis of nine benchmarks contrasting the performance of 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. . URL `http://doi.acm.org/10.1145/2544137.2544165`.

[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. . URL `http://dx.doi.org/10.1109/PACT.2011.15`.

[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] B. R. Gaster and J. G. Morris. Embedding OpenCL in GHC haskell. 2013.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, Mar. 2008. ISSN 1542-7730. . URL `http://doi.acm.org/10.1145/1365490.1365500`.

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

[16] 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.

[17] 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.

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

[19] 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.