Haskell on the GPU

R.B. Rubbens
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
r.b.rubbens@student.utwente.nl

ABSTRACT

Moore's law is coming to an end: processing power is not doubling any more. To keep increasing processing power, Graphical Processing Units (GPUs) can be used. However, the programmatic complexity of these devices stands in the way of their adoption. A better and more parallel programming model is needed. To assess the viability of programming a GPU with a functional programming language, a minimal subset of Haskell that compiles to OpenCL was implemented. Using Haskell to program a GPU made some programs more clear and concise, but performance suffered. Especially the absence of global synchronization of the GPU and dynamic memory allocation made it harder to implement Haskell for the GPU. However, the prototype implementation proves that it is possible to use a high-level language to program the GPU.

Keywords

OpenCL, GPU, parallel execution, functional programming

1. INTRODUCTION

Over the past few years it has been becoming more and more clear: the magic doubling of processing power every two years (better known as "Moore's Law"[11]) as a result of the doubling of transistor density is slowly grinding to a halt. An actual doubling of transistor density has not happened in 2 years[2]. It is clear to the industry but also to academia that an alternative way of acquiring a speed-up in hardware and software is needed.

The past few years GPUs have gained popularity in the field of High Performance Computing (HPC). GPUs are chip boards containing several hundred smaller Central Processing Units (CPUs) that are running in parallel. A GPU can, if properly programmed, churn through gigabytes of data quickly. A CPU usually takes an order of magnitude longer because of its sequential nature. The relatively low cost of GPUs and their promise of processing power makes them attractive.

To program a GPU a low-level variant of C named OpenCL is often used. It is flexible and effective, but it is not a pleasant language to work with. The imperative nature of OpenCL does not aid its ability to deal with concurrent execution issues such as data races and deadlocks. The combinatorially complex problem of parallel execution and the low-level nature of OpenCL makes it clear that programming a GPU is hard. If a speedup corresponding to Moore's Law is ever to be achieved, something has to be done about this barrier of complexity.

A possible solution to this problem is to design a functional high-level language that compiles to OpenCL. A functional language can help combat concurrent programming issues, while a high-level language could aid the programmer by giving him more expressive power. To research the viability of such an approach, the following questions need to be answered:

1. What problems do experts experience when using OpenCL?
2. How can these problems be solved in a high-level functional language?
3. What are the benefits and drawbacks of programming a GPU with a functional high-level language compared to regular OpenCL?

To answer question one I will interview experts in the field of GPU programming to acquire problems and patterns that are recurring in GPU programming. Furthermore, relevant research on GPU programming languages and libraries will be studied. This will be helpful in determining what language features are effective and what language design choices to avoid. Combining the acquired expert knowledge with the literature study, Hywar, a small subset of the Haskell programming language targeting the OpenCL platform, will be implemented as a proof of concept of a high-level functional GPU programming language. From this implementation research question two and three can be answered.

In this paper background will be given on GPUs and the OpenCL platform. Similar efforts on reducing OpenCL complexity will be discussed, after which the method of interviewing experts and the implemented language will be explained. Then details will be given on the inner workings of the Hywar compiler. Lastly, the interview results and implementation details will be presented and discussed, followed by a conclusion.

2. BACKGROUND

2.1 Graphical Processing Units

GPUs have gained popularity over the past few years. At first they were only known for their ability to render 3D scenes and films with crisp quality. Since august 2009,

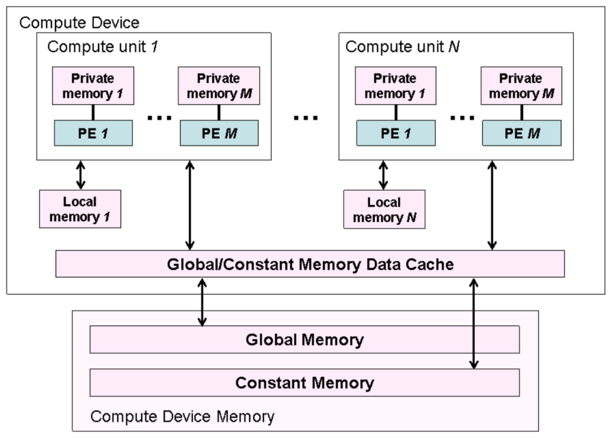

Figure 1. The OpenCL architecture as depicted in [8]. In practice a compute device is usually a GPU, a compute unit a workgroup, and a Processing Element (PE) an actual processor.

the release of the OpenCL 1.0 standard, an effort has been made to standardize the general purpose computing power offered by GPUs. This is not the first time efforts were made to use the computing power of GPUs for something else than rendering however. As early as 2005 and possibly even earlier GPUs were already used for general purpose computing, despite the absence of the OpenCL standard[3].

The many processors on a GPU make it a perfect fit for highly parallel tasks such as image processing and stencil operations. However, to fit so many processors on one chipboard, the processors have to be less powerful and fully-featured compared to actual AMD or Intel CPUs. This means that sequential algorithms and tasks are quite likely perform a lot worse when running on a GPU. The takeaway here is that “there is no such thing as a free lunch”: GPUs can only provide substantial speed-up if it is used in the proper context.

2.2 The OpenCL platform

OpenCL is a flexible language, allowing implementation of not only image processing but also complex algorithms such as bitcoin mining and scientific simulations.

OpenCL models GPUs in an abstract way. OpenCL assumes that a GPU consists of many individual “processors”, where each processor is executing the same “kernel”. A kernel is a program executed by a GPU. The amount of processors available is usually in the range of 100 to 500. An extra assumption OpenCL makes is that these processors can be divided in “workgroups” of arbitrary size, as long as the total amount of working processors is divisible by this workgroup size. The crux of the concept of workgroups is that while processors in the same workgroup can be synchronized (that is, a processor can stop execution until another processor in a workgroup has done something), processors in different workgroups cannot be synchronized. This constraint allows kernels to be executed on GPUs with varying amounts of processors, as long as the GPU supports the workgroup size. The architecture is visualised in figure 1.

There is one aspect in which OpenCL differs from many other programming languages: the need for a “host language”. In traditional languages the developer writes code and then instructs the compiler to compile the code. The developer is then left with an executable that is the program the developer has written in machine-executable form. For OpenCL this is not the case. In the context of OpenCL the developer writes a kernel, which is then compiled to GPU-executable form. The developer can not execute this GPU-executable form on its own. First code has to be written to load a compiled kernel on the GPU, load the arguments to the kernel, and then execute the kernel. This “kernel management” is all done in a host language. This kernel management is also the reason why GPU programmers often talk about two languages: OpenCL on one side (to program the kernel), and a host language (C, C++, Java, Python) on the other side (to manage the kernel).

However, OpenCL is not perfect. OpenCL is the closest a developer can get to the hardware without significant rise in complexity of tooling and usability. It only gives the programmer the very basic needs of a programming language. Recursion is an example of high-level feature not supported by OpenCL[8]. It is not a mandatory feature, but recursion can happen without the programmer intending it. Combined with the fact that the compiler does not warn for this, this can be a hard to debug problem. Safety is another concern. One example of this is that array accesses in OpenCL are not checked for boundaries. This makes OpenCL programs vulnerable to buffer overflows. Lastly, the usability of the OpenCL API is bad. For a small GPU program of 15 to 20 lines, the programmer needs between 100 and 150 lines of overhead code to run the GPU program and extract the results.

A possible cause of the complexity of OpenCL might be the roots of its programming model. While OpenCL represents the underlying architecture quite well, it is still inherently sequential. Some researchers argue that for the power of a parallel environment to be fully utilized a new programming model has to be created that does not depend on execution order[1]. When the order of instructions is taken away from the programming model a model that is more akin to mathematics is the result. With functional languages being akin to mathematics as well, functional languages should be researched as a possible solution to programming GPUs.

3. RELATED WORK

It is not the first time this problem is tackled. A traditional solution is to wrap OpenCL in a high level library or Domain Specific Language (DSL). Many libraries have been written to make programming GPUs from a host language more bearable (Boost.Compute, Data Parallel Haskell, accelerate, Thrust, and more). One problem these libraries solve well is the usability part. When a library is used, the OpenCL API is usually locked away behind a layer of abstraction. But these libraries are also not perfect. The abstraction layer over OpenCL can be too thin, resulting in a complex library interface. If that is not the case, the library is usually too high-level. This results in a loss in flexibility, and often requires complex marshalling of arrays to “GPU arrays” and other business.

The approach used in this research is another option: create a programming language that encapsulates the complexity of the target. Several languages have been developed that try to solve the problem that way: Futhark, RenderScript, and many more. Three of these stand out: SkelCL[14], Bacon[15], and Harlan[6]. Each of these languages improve upon OpenCL in their own way, but they also each have their own shortcomings.

SkelCL is a library and domain specific language that allows the programmer to easily interface with the GPU.

It simplifies the implementation of simple GPU programs with the introduction of so called ”skeletons”. This makes SkelCL a good basis from the point of view of usability. However, the language with which the skeletons are programmed is very spartan. It is useful for simple programs, but more complex programs still require OpenCL.

Bacon is a programming language that compiles to OpenCL. At the same time it also provides an API to interface with and execute the Bacon programs in C++. While Bacon improves OpenCL on both the language and usability front, it did not experience widespread adoption. One could argue that this proves that a high-level language targeting OpenCL is not the solution programmers are looking for. In my opinion, Bacon is just ahead of its time.

Harlan is a dialect of Scheme, geared towards interfacing with the GPU. In the spirit of Scheme it provides a minimalist programming language, extended with the “kernel” keyword to allow parallel execution on the GPU. While Harlan is a functional language, it runs in a restricted environment, making it not immediately obvious how the code can be interfaced with a C++ or Haskell program. With Scheme being well-known for being a language useful for teaching, Harlan scores very high on the accessibility scale.

These three languages are relevant because they improve upon OpenCL in a way that this research is also trying to: by providing an abstraction layer over OpenCL that allows the programmer to reason more abstractly and more easily. Especially Harlan is successful in this regard, as functional languages are often seen as the pinnacle of a way of programming in an abstract manner. However, usability is also important, and that is where SkelCL & Bacon prevail. This research builds upon the progress made in the Harlan language, while taking as much as possible from the usability of SkelCL & Bacon.

4. METHOD

To answer the research questions experts were interviewed and a programming language was implemented. In this section the reasoning behind interviewing will be explained, as well as the reasoning behind the choices for the programming language implementation.

4.1 Expert interviews

Experts have been interviewed to find the most important problems in the field of GPU programming. The main reasoning behind this is that experts first and foremost have had the proper time to actually do substantial amounts of GPU programming. This gave them the chance to experience the shortcomings of GPU programming first hand. Furthermore these experts are not only experts in the programming of GPUs, but they are also experts of their domain. They do not have unrealistic expectations of the capabilities of the GPU platform: if they say something is tedious and should be easier it is likely that this is a widespread problem in the field.

The experts were asked various questions about their experience with GPU programming, recurring programming patterns, and how time was spent on different programming activities (programming, debugging, optimization, etc.). The actual questions used to lead the interview are visible in appendix A.

4.2 The Hywar language specification

To determine the limitations and strengths of the OpenCL platform as a target for a functional language I have implemented a functional language: Hywar. For the most part it is not unlike Haskell. It has functions, variables, if-statements, and lists. There are a few caveats however.

4.2.1 General concerns

I have chosen Haskell to implement the compiler because Haskell is generally perceived to be a good environment to program a compiler in. This is mostly because Haskell has pattern matching and support for deep recursion, which is useful in compiler construction.

C++ was decided to be the host language target for the Hywar compiler. The reason for this being that while it is high-level (it has high-level data types such as lists and sets), it still interoperates with many programming languages. This ensures that the programs produced by the compiler are usable in many contexts and architectures.

It was determined that Hywar has to be based on Haskell. This has a few advantages. Firstly, the implemented programs will seem familiar to a large portion of the audience because Haskell is widely known. Secondly, my supervisor Jan Kuper already had a parsing framework for Haskell available, which greatly accelerated the process of implementation.

4.2.2 Implicit and explicit parallelism

In programming language design there are two ways to implement parallelism: implicit and explicit. Implicit means that in no way the developer indicates where or how parallelism takes place. The developer merely indicates which computations and functions should happen, and the compiler then analyses the code for possible opportunities for parallelism. Explicit means that the developer specifically says which code he wants to run in parallel. The developer can do this via specialized keywords, or by annotating regular code. Implicit parallelism is a more higher-level approach and restricts the developer less. It does however place a burden on the compiler to figure out what to parallelise. Explicit parallelism is more low-level, and places the burden on the developer to correctly indicate what to parallelise. In the context of compiling Haskell to OpenCL, both alternatives were considered.

The explicit way, as shown in listing 1, is very close to standard Haskell. In this case, the code generated for the main function would be compiled to OpenCL C and executed almost “in verbatim” on each processor of a GPU. The only difference between each individual execution is that `getGlobalId` returns a different ID for each distinct processor. Processor 0 gets global ID 0, processor 1 gets global ID 1, and so on. That way, each processor can search for a different prime. However, given this model, it is for example not possible to filter elements from an array. Extra custom notation is needed to signal this intent.

The implicit way, as shown in listing 2, is actually standard Haskell. While listing 2 is more concise and understandable than listing 1 the compiler not only needs to do more work (what part of this code has to be parallelised?) but the implementation of the filter operation is also left to the compiler. While in a functional context this is usually desired, it can be seen as a limitation.

For Hywar explicit syntax was chosen due to the complex data flow analysis needed for implicit parallelism. This also made implementation feasible within a short time frame. See listing 3 for an example of the chosen syntax.

4.2.3 Hywar program structure

A Hywar program is basically a Haskell `let` statement, as can be seen in listing 3. The statement first has a series

```
--@filter (\x -> x /= -1) main
main :: [Int] -> Int
main ps
    | isPrime prime = prime
    | otherwise = -1
    where
        prime = ps!!(getGlobalId 0)
```

Listing 1. Naive prime finding program in conceptual Haskell with explicit parallelism

```
main :: [Int] -> [Int]
main ps = filter isPrime ps
```

Listing 2. Naive prime finding program in Haskell with implicit parallelism

of variable definitions and types, followed by the keyword `in`, after which the actual "body" expression to be calculated is located. In this body expression only variables and functions defined in the let statement can be used.

The variable definitions are used for two purposes in Hywar. The first purpose is defining numerical constants or functions, such as `daysOfTheWeek = 7` or `multiplyByFive = \x -> x * 5` respectively. The second is to define inputs to be supplied at program execution. An example of this is in listing 3, where `ps :: [Int]` signifies that `ps` is a list of integers to be supplied at program execution.

The type system of Hywar is vastly simplified compared to Haskell: every variable is either a list of integers, an integer, or a function taking a certain amount of integers as arguments and returning an integer. Variables can only be lists if they are inputs to the GPU program (such as the `ps` in listing 3).

The body expression must be a function call. The following functions can be called: `filter`, `reduce`, `map`, `zipWith`, or `plow`. These are all similar to their cousins in Haskell, with a few exceptions. `filter` does the same as the `filter` in Haskell, except it might change the ordering of the results. `reduce` is similar to `foldl1`. `map` and `zipWith` do exactly the same as the Haskell versions. `plow` is the same as `scanl1` in Haskell.

5. HYWAR IMPLEMENTATION DETAILS

The Hywar compiler consists of four phases. These four phases turn a textual representation of a Hywar program into a C++ header and source file containing one function. This function has arguments in the same order as specified by the Hywar program, and when called executes the Hywar program. The four compiler phases are (in order of execution): lexing and parsing, tree transformation, OpenCL kernel generation, and C++ function generation.

The Hywar language implementation source is located at `https://github.com/bobismijnnaam/hywar`. It is licensed under the liberal open source MIT license.

See listing 4 and listing 5 for an example of generated OpenCL code from a Hywar program.

```
let
    ps :: [Int]
in
    filter isPrime ps
```

Listing 3. Naive prime finding program in Hywar

5.1 Lexing & Parsing

This phase turns a textual representation of a Hywar program into an expression tree representation. The lexing and parsing phase was implemented by Jan Kuper. The parser already parsed a big subset of Haskell, so it was trivial to make it parse Hywar, which is a tiny subset of Haskell. This whole phase is wrapped in the function `showExpr` in the file `Graphviz.hs`.

5.2 Tree transformation

The tree transformation phase transforms the Haskell expression tree generated by `showExpr` into an imperative expression tree that is easily convertible into OpenCL code. This is done in four separate steps, where each step produces a new tree that is slightly different from the old tree. They are the following (in order of execution): function application to function calls, lambda lifting, global variable typing, and local variable typing. This phase is executed by the function `doTaal` in the file `Compiler.hs`.

Function application to function calls.

In the Haskell expression tree function applications are nested. This means that if there is a function that takes two arguments, the second argument is applied to the function resulting from applying the first argument to a function. This makes sense in the functional paradigm, but in an imperative language there is no function application. Instead, functions are called: arguments are set somewhere in memory, and a jump is made to the function. Function application nodes in the Haskell expression tree are "flattened" to a function call to represent the underlying structure properly.

Lambda lifting.

In a Haskell expression tree a function can be passed to another function by means of a lambda. In OpenCL functions cannot be passed to other functions, so the Haskell expression tree cannot be directly translated to OpenCL[8]. I solve this by doing lambda lifting[7]. This "lifts" each lambda expression into each own global function definition. A reference to the global function definition is then hard-coded into the expression tree. Since now only the reference to the function definition is passed, the tree is convertible to OpenCL.

Global variable typing.

The global variable typing step assigns types to variables and function definitions.

Local variable typing.

Type information from the previous step is used here to assign types to functions calls and variables used in expressions.

5.3 OpenCL kernel generation

In this phase the kernel that will be executed by the GPU is generated. It starts out with a "boilerplate kernel" based on which body expression is used. To this boilerplate kernel are then added various functions that are defined by the user in the Hywar program. Lastly, this phase also leaves a few blanks in the kernel. These blanks will be filled in by the host language. These blanks are the kernel arguments and the lengths of various intermediate results. This approach is called "Just-In-Time specialization", pioneered for GPUs by the Bacon programming language[15].

```
let
    ps :: [Int],
    isEven = \x -> (mod x 2) == 0
in
     filter isEven ps
```

Listing 4. A Hywar program that filters all even numbers from a list.

```
int mod(int a, int b) {
    return a % b;
}

int isEven(int);
int lf_Cej8auY3yMX(int);

__constant int ps[{! ps_length }] = {!ps_contents};

int isEven(int v1) {
    return lf_Cej8auY3yMX(v1);
};

int lf_Cej8auY3yMX(int x) {
    return (mod((x), (2))) == (0);
}

kernel void GPUMonad(__global int *out, __global int *count) {
  if (isEven(ps[ get_global_id (0)])){
    out[atomic_inc(count)] = ps[ get_global_id (0)];
  }
}
```

Listing 5. An OpenCL kernel generated by the Hywar compiler. The `{!...}` segments are to be replaced with constants at run-time

5.4 C++ function generation

The last phase generates wrapper code in the host language that runs the kernel generated in the previous function. This wrapper code consist of a header and a source file in the C++ language and contains one function. When this function is called the blanks in the generated kernel are filled in, the kernel is compiled, the kernel arguments are loaded, the kernel is run, and the results are extracted and returned to the caller. The only dependencies of the generated function is that the C++ compiler links against the OpenCL SDK.

6. RESULTS

6.1 Expert interviews

Four experts were interviewed about their opinions of the programmability of graphics cards. The interviewed experts were (in no particular order): Kees Lemmens (University of Delft), Saeed Darabi (University of Twente), and Henk Mulder (University of Twente). One expert wished to remain anonymous. Three out of four experts preferred using CUDA to OpenCL in their day to day work. In summary, their responses can be divided in roughly three categories: language features, optimization, and usability versus performance.

Language features.

Language features are all features that should be provided by the language natively or via a standard library. Functional patterns such as reduce and map are apparent and useful in GPU programming. Developers already benefit from premade functional patterns packaged in libraries, so developers will benefit from a language with these patterns built-in. Automatic or low-effort scalability to more than one machine is a very desirable feature for a library or language to have. Examples of this are OpenMPI and Apache Hadoop. Functional features such as abstract data types (e.g. sum- and product types) and high-level data structures (e.g. maps, trees) are not used very often. And in the case that they are, the experts often prefer implementing them themselves for the sake of performance. The experts also had the experience that the absence of dynamic memory allocation is not a problem. In the cases that it is a problem, it can be solved by allocating memory before the kernel starts. Lastly, an extensive standard library for GPU programming was not deemed very useful. When the GPU feature landscape settles it might become a possibility.

Optimization.

The most notable and hard to avoid bottleneck in GPU programming is memory management. This encompasses the transfer of data to and from GPU memory. Automated solutions for this problem exists, but they are still very immature. These solutions are either too intrusive for the OS or they transfer too much useless data. Another problem of almost equal importance is the fluctuating landscape of GPU features and architecture. Features and architectures change often and quickly in GPU programming, which makes it very hard to optimize a program for more than one GPU. Different GPUs often have different features and strengths, so programs optimized for different GPUs look very different from each other. This makes a GPU program either slow and comprehensible or fast and incomprehensible.

Usability versus performance.

In the case of beginners getting a proper GPU programming environment up and running and debugging GPU programs costs the most time. In the case of experts optimization of a GPU program costs them the most time. Experts say that while having the choice between usability and performance would be nice they would always prefer performance. Hard to debug code is only a minor nuisance and perfectly acceptable. The complexity of GPU programming is a double-edged sword: while on one hand it is difficult to pull off, the reward of getting a GPU program right is worth it. On the whole experts mostly agree that the degree of complexity involved in a GPU program is higher than it should be. There definitely is room for a technology that would allow easier GPU programming in exchange for performance, if only to help beginners start out with GPU programming.

6.2 The Hywar language implementation

All in all the implementation of the Hywar specification was a success. As expected from the implementation process the following key insights were derived.

6.2.1 Performance

One of the more notable and distinct insights derived from the Hywar implementation is its performance. Even though it is a prototype built in relatively short time, the performance should still be taken into account. As depicted in figure 2 a Hywar implementation of the benchmark runs about a hundred times slower than a sequential C++ implementation of the benchmark. The benchmark consists of five example programs running on a variable amount of data, with each program using one of the body function calls (`zipWith`, `plow`, etc.).

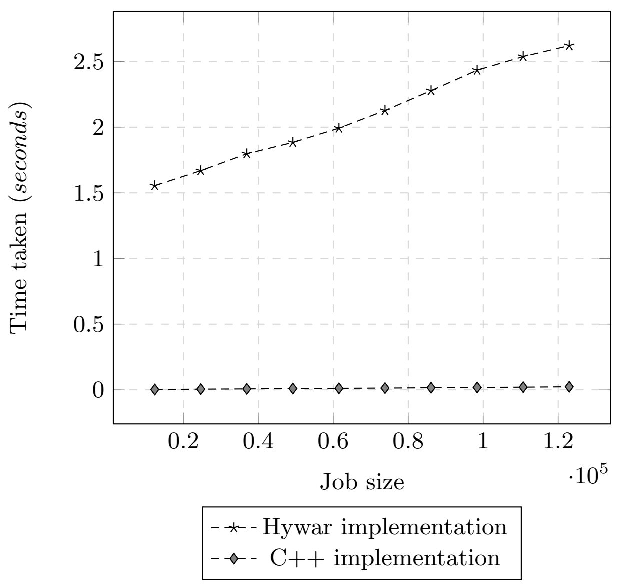

Figure 2. Runtime of the benchmarks made with Hywar (executed in parallel on the GPU) compared againsts benchmarks made in C++ (executed sequentially on the CPU).

Program	Hywar	OpenCL
filter (6003)	5	20
reduce (6004)	5	32
map (6005)	6	19
plow (6006)	5	32
zipWith (6007)	6	19

Table 1. Lines of code in Hywar compared to lines of code in OpenCL generated by the Hywar compiler. The numbers 6003-6007 correspond to the example programs in the benchmark.

The functions used in each program were very simple (checking if the number is a multiple of 2, adding two numbers together, etc.), while the input arrays grew towards the size of 100.000 elements.

6.2.2 Usability

An informal way of comparing two programming languages is looking at the lines of code used for a program implemented in both languages. This metric is shown for Hywar and OpenCL generated by Hywar in table 1.

6.2.3 Implementation difficulties

While the implementation of the Hywar compiler was successful, not every feature translated directly from the Hywar specification to an OpenCL implementation. The first problematic feature encountered that lead to the removal of the feature from Hywar was the List data type. While Lists form the foundation of many Haskell programs, their ability to be sized flexibly made it very hard to implement them in OpenCL. The removal of Lists made Hywar a lot less powerful than it could have been, but it did make implementation feasible in the time frame of this research.

```
kernel void GPUMonad(__global int *out, __global int *count) {
    if (isPrime(ps[ get_global_id (0)])) {
        out[atomic_inc(count)] = ps[ get_global_id (0)];
    }
}
```

Listing 6. An OpenCL kernel that filters all primes from a list. The definition of `isPrime` is omitted.

Another problem that lead to a small addition to the Hywar specification was the “stability” of `filter`: the guarantee that the order of filtered elements does not change. The Hywar implementation of filter uses an atomic increment to keep track of filtered elements. While this makes it possible to run the `filter` in parallel, it does allow the filter to be executed in any order. Retaining the original order of the elements would most likely require global synchronization. This is not available at the time of implementation, so the stability guarantee was removed.

Lastly, when reduce and plow were implemented similar problems with global synchronization were encountered. Since there was not enough time to solve this problem optimally the Hywar compiler cheats and supplies a partially sequential implementation. While the implementation of reduce is still theoretically faster than a sequential reduce, the implementation of plow is definitely slower than a sequential plow. This is because plow is implemented by first letting each workgroup do a plow on a portion of the input list, and then letting one processor do a global plow on the whole input list using the intermediary results. While this does employ each processor to justify calling it a GPU implementation, it is certainly slower than just doing a plow on the CPU.

7. DISCUSSION

7.1 Interviews

While the response of the experts was not entirely uniform, overall the idea that there is room for a high-level language targeting OpenCL was present. This language does not necessarily have to be functional; the experts were impartial to which paradigm it should be based on. Possibly future research could look at even more paradigms in the context of GPU programming besides imperative[15] and functional. Another insight the experts agreed upon was that in HPC there probably is not much use for such a language. In a domain where every ounce of performance has to be squeezed from the hardware even the overhead of a very good compiler is too much.

However, in a domain where a two or three times speed-up instead of a forty times speed-up is enough the language can still be effective. An application that comes to mind is quick prototyping of GPU-accelerated programs. The easy and quick implementation of an algorithm in Hywar can give an outline of the possible performance gains that can be achieved with a pure OpenCL implementation. Portable efficiency can also be approached with a high-level language, something that is non-existent currently. A compiler can optimize programs for specific GPUs and employ “rule of thumbs” (workgroup sizes, picking the right data types, etc.) that are hard for the developer to keep track of. While the experts believe that at the moment the GPU landscape is too chaotic to keep track of all these rule of thumbs, an effort can still be made. Research could be conducted to categorize characteristics of various GPUs. Above all I believe that as the GPU landscape will stabilize this benefit will keep on growing.

7.2 The Hywar language implementation

Lists.

One of the major problems when implementing Hywar was the inability for functions to produce flexibly sized lists.

This seems like a hard problem to tackle with just static analysis, especially with the presence of functions like `filter`. Earlier research has evaluated a Kernel Memory Allocator (KMA), proving that it is possible to have some form of dynamic memory allocation on the GPU[12]. A drawback about this KMA however was that it was slow. In Hywar, lists mostly appear in a per-thread context. Therefore I think that the KMA should be re-evaluated in the context of local or maybe even private memory. While there is not much private and local memory available (4kb and 64kb respectively, [10, 9]), they are a lot faster than global memory. This could make the KMA useful outside debugging contexts.

Body expressions.

Another major problem was the implementation of the body expressions `filter`, `reduce`, and `plow`, in the absence of global synchronization. Research is already being done on that matter. For example, using StreamScan[13] a plow function could be implemented that is actually faster and more efficient than its sequential implementation. Furthermore, if more global synchronization primitives become available research has to be done if these can be used to implement the body expression more efficiently, such as a parallel `filter` described in [5]. Research can also be done on the feasibility of using multiple kernel calls for global synchronization. While this is mostly believed to incur too much of a performance penalty, it might still be an acceptable alternative in some cases.

Runtime performance.

The runtime performance of the Hywar benchmark compared to the C++ benchmark is worrying at first sight. However, it underlines that for a GPU implementation to be faster than its sequential version, it should either have very low memory requirements or be a very compute-intensive algorithm. If a more complex function would have been used in each program (like a Fourier transform or a prime finding function), Hywar performance might have been on par with a sequential implementation.

Usability.

When a Hywar program is compared to a program generated by the Hywar compiler the advantages from a Hywar program are immediately clear (see table 1). Most of the lines of code in the generated OpenCL program come from the aggregation step at the end (e.g. reduce or plow). For bigger programs this overhead might be insignificant, but in the benchmarks it is dominant. I also argue that a Hywar program is clearer than its OpenCL version. This is not only because of the simplicity of the Hywar language, but also because of the more high-level semantics attached to its constructs. To illustrate this, a hand-written OpenCL version of listing 3 is shown in listing 6.

When discussing the translation from high-level Haskell to low-level OpenCL, the Hywar compiler takes care of numerous details: synchronization between processors, final aggregation of the data, and other tasks that are hard to get right. However, there is more to running a GPU program than only writing it. To get a GPU program to run a lot of bookkeeping has to be done in the host language: buffers have to be allocated, sources have to be compiled, errors have to be checked, and much more. While this complexity is not visible in table 1, I think this additional complexity is non-negligible.

A developer using OpenCL either has to do this bookkeeping manually or do the bookkeeping aided by a library. Hywar solves this problem by packing all the individual bookkeeping into one function the developer can call. The function takes care of compiling the kernel, error handling, the execution of the kernel, the extraction of the results, and all the glue code in-between. This reduces code clutter, and allows the developer to spend time on the correctness of his program. The way of solving this problem is heavily inspired by the Bacon programming language[15]. This is a very powerful way of encapsulating complexity and I therefore advice other researchers to try and incorporate this structure into their languages and libraries.

Implicit versus explicit parallelism.

Lastly, while Hywar can be used to program GPUs, it is still an explicitly parallel language. For a high-level language implicit parallelism is desired. Research done on data flow analysis (such as [4]) can reduce the complexity of determining what part of a program is a candidate for parallelisation, and how it should be parallelized. Ideally, such a data flow analyser would analyse regular Haskell code, and replace parallelisable parts with Hywar kernellet expressions. This can be a very interesting direction of research. Research should also be done on how expressions can be collapsed into other expressions. For example, in the expression `fold addTuple 0 (zipWith xs ys)` the `zipWith` does not have to finish before the fold can start. It only has to turn two values into a tuple, after which the tuple can be passed to the `fold` function. This can be done in parallel, but the compiler has to "collapse" two functions into one. This is possibly also in the domain of data flow analysis.

8. CONCLUSION

In this research experts were interviewed for their experience with GPU programming. From the results it is clear that experts spend the most time on GPU specific optimization of a GPU program. Beginners have more trouble with getting the GPU environment up and running. Experts think a high-level language cannot be efficient enough for HPC. They would also like the GPU feature landscape to stabilize. A high-level language could reduce the time spent on optimization by having deep knowledge and rule of thumbs for GPUs. The high-level language could also provide a consistent platform to reason about.

Based on the expert suggestions the high-level functional language Hywar was designed and implemented. The benefits of a functional high-level GPU language is that code is shorter and clearer, and that the developer does not have to deal with the complex OpenCL API. Its drawback is that it is less flexible and that there is a loss of performance. Especially the absence of global synchronization and dynamic memory allocation can make it hard to implement an efficient and powerful high-level language. More research has to be done to make trivial GPU programs faster than their CPU counterparts. However, the prototype proves that a high-level language to program GPUs in is definitely a feasible possibility. All in all, one thing is certain: with enough research a free lunch is on the horizon.

9. REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A

view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. Chapter 5.

[2] Z. Bach. 28nm: The last node of moore's law. `http://www.eetimes.com/author.asp?doc_id=1321536`, 2014. Online. Last accessed 2016-05-16.

[3] D. Göddeke. Gpgpu - basic math tutorial. `http://www-lsiii.mathematik.uni-dortmund.de/papers/Goeddeke2005b.pdf`, 2005. Online. Last accessed 2016-06-10.

[4] R. D. Groote, P. K. F. Hölzenspies, J. Kuper, and G. J. M. Smit. Incremental analysis of cyclo-static synchronous dataflow graphs. *ACM Trans. Embed. Comput. Syst.*, 14(4):68:1–68:26, Dec. 2015.

[5] D. Grossman. Lecture 3: Parallel prefix, pack, and sorting. `http://homes.cs.washington.edu/~djg/teachingMaterials/spac/`, 2016. Online. Last accessed 2016-06-13. Slide 12.

[6] E. Holk, R. Newton, J. Siek, and A. Lumsdaine. Region-based memory management for gpu programming languages: Enabling rich data structures on a spartan host. *SIGPLAN Not.*, 49(10):141–155, Oct. 2014.

[7] T. Johnsson. *Lambda lifting: Transforming programs to recursive equations*, pages 190–203. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985.

[8] Khronos OpenCL Working Group. The opencl c specification. `https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf`, 2016. Online. Last accessed 2016-05-16.

[9] M. Kruliš, T. Skopal, J. Lokoč, and C. Beecks. Combining cpu and gpu architectures for fast similarity search. *Distributed and Parallel Databases*, 30(3):179–207, 2012.

[10] K. Matsumoto, N. Nakasato, and S. Sedukhin. Implementing a code generator for fast matrix multiplication in opencl on the gpu. *2013 IEEE 7th International Symposium on Embedded Multicore Socs*, 0:198–204, 2012.

[11] G. Moore. Cramming more components onto integrated circuits. *Electronics Magazine, Volume 38, No. 8*, 1965.

[12] H. Mulder. Concurrent manipulation of dynamic data structures in opencl. 23rd Twente Student Conference on IT, 2015.

[13] Y. Shengen, L. Guoping, and Z. Yunquan. Streamscan: Fast scan algorithms for gpus without global barrier synchronization. *SIGPLAN Not.*, 48(8):229–238, Feb. 2013.

[14] M. Steuwer and S. Gorlatch. *Parallel Computing Technologies: 12th International Conference, PaCT 2013, St. Petersburg, Russia, September 30 - October 4, 2013. Proceedings*, chapter SkelCL: Enhancing OpenCL for High-Level Programming of Multi-GPU Systems, pages 258–272. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[15] N. Tuck. Bacon: A gpu programming system with just in time specialization. `http://www.cs.uml.edu/~ntuck/bacon/BaconPaper.pdf`. Online. Last accessed 2016-05-16.

APPENDIX

A. INTERVIEW QUESTIONS

1. What is your background?
2. For what kinds of tasks do you use OpenCL (for example, certain kind of image processing, or specific scientific simulations, or something entirely different)?
3. What takes the most time when making a GPU program?
4. Was it often a problem that recursion is not possible in OpenCL? How was this problem dealt with?
5. Was it often a problem that dynamic memory allocation is impossible during executino of a kernel? How was this problem dealt with?
6. Which datatypes are missing in OpenCL? In what context would you use these missing datatypes? (I mean types in two ways: abstract "types", like sum types, tuples, etc. Or more concrete types: hashmaps, sets, etc.)
7. What kind of functionality would you like to be in a "standard library" for OpenCL? (Better arrays, foreach, standardised arithmetic functions e.g. min/max/cos, etc.)
8. What kind of "boilerplate code" repeats itself often in projects, but is hard to encapsulate in a function/class/library?
9. What kind of programmingpatterns/structures is recurring in GPU programs? (For example, a kernel that applies a stencil function to an image, a kernel that maps/folds a function on a range of elements, or something entirely different)
10. Does it happen often that multiple kernel invocations occur on the same data? What do you think of the performance impact?
11. Are there bugs that often appear, even though there is a well-known remedy for it?
12. What is the coolest thing you've used OpenCL for, and why?