Appeared in *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPHAM 2009)*

GPU Kernels as Data-Parallel Array Computations in Haskell

Sean Lee
University of New South Wales
seanl@cse.unsw.edu.au

Manuel M. T. Chakravarty
University of New South Wales
chak@cse.unsw.edu.au

Vinod Grover
NVIDIA Corporation, USA
vgrover@nvidia.com

Gabriele Keller
University of New South Wales
keller@cse.unsw.edu.au

Abstract

We present a novel high-level parallel programming model for *graphics processing units* (GPUs). We embed GPU kernels as data-parallel array computations in the purely functional language Haskell. GPU and CPU computations can be freely interleaved with the type system tracking the two different modes of computation. The embedded language of array computations is sufficiently limited that our system can automatically isolate and extract these computations and compile them to efficient GPU code. In this paper, we outline our approach and present the results of a few preliminary benchmarks.

1. Introduction

For the last decade, *graphics processing units* (GPUs) evolved from fixed-function pipelines, which were aimed purely at graphics acceleration, to programmable parallel processing units that are suitable for general purpose computations. Not only programability, but also performance improved. Recent GPUs are multicore processors with up to 240 cores per GPU, optimised for data-parallel workloads with large amounts of excess parallelism to cover memory access latencies via multi-threading. Current GPUs exceed current CPUs by about a factor of 10 in peak floating-point performance and by about a factor of 5 in memory bandwidth [16], which makes them an interesting target for *general purpose computations on GPUs* (GPGPU) [13, 17, 7].

GPUs realise this performance advantage only for highly data-parallel work loads with regular data access patterns. GPU programming environments aimed at GPGPU, such as NVIDIA's CUDA [16] and the recent OpenCL standard [8], are based on the C programming language. They extend C with support for managing GPU devices, transferring data between the host CPU and devices, and executing multiple simultaneous instances of data-parallel threads. They also limit side effects in data parallel computations and place restrictions on the use of control flow constructs, such as conditionals, loops, recursion, and function pointers. For example, the currently latest generation of GPUs does not support function pointers in GPU kernels (i.e., in the code executing in data-parallel GPU threads); recursion is only supported to a fixed depth; and conditionals and loops need to be used in a limited form to avoid the serialisation of the parallel operations and to achieve good performance. Furthermore, GPU kernels need to be defined as separate functions, which leads to program decomposition on the basis of architectural constraints instead of standard software engineering considerations.

In this paper, we propose an approach to GPGPU programming that is at a considerably higher level of abstraction. We embed GPU kernels as collective array computations in the purely functional language Haskell. This is attractive for several reasons. Collective array operations have a well-known data-parallel interpretation [4]. Moreover, Haskell's type system ensures that all parallel computations are side-effect free —this is checked by the compiler— and our embedding ensures that all array computations can be executed on GPUs without any informal, ad-hoc constraints on control-flow constructs. Finally, GPU kernels may be freely embedded in CPU code, liberating program decomposition of architectural constraints. The main contributions of this paper are as follows:

- We present an embedded array language in Haskell whose operational semantics is massively data parallel and suitable for execution on GPUs (Section 2).
- We outline an online compilation scheme for this array language (Section 3).
- We present preliminary benchmark results (Section 4).

We defer the discussion of related work to Section 5.

We don't assume close familiarity with Haskell. Instead, we explain the peculiarities of Haskell's notation as we use it. The work discussed in this paper is still in progress, and the reported results are preliminary.

2. An embedded array language for GPU kernels

Instead of imposing restrictions on an existing language for GPU computations, we design a new *domain-specific language* (DSL) that supports the specification of GPU computations, exposing most of the core capabilities of GPUs. To facilitate the seamless integration of sequential CPU code and data-parallel GPU code, we *embed* the DSL as a set of expressions in Haskell [9]. This *embedded domain-specific language* (EDSL) (also called an *internal language*) is a *two-level language* — i.e., it statically distinguishes between (1) *collective array operations* (such as, mapping an operation over all elements, filtering elements with certain properties, and reducing all elements to a single result) and (2) *scalar computations* (such as, arithmetic operations) used to parametrise collective operations. More specifically, we define the EDSL as a Haskell library that exports a rich set of array operations and compose these operations using Haskell's standard functional notation.

Array computations scheduled for execution on a GPU can be embedded anywhere in a Haskell program and are distinguished from sequential CPU code by their type. Expressions of type `GPU a` denote collective array operations yielding a result value of type `a`, whereas expressions of type `GPU.Exp a` denote scalar computations that we use to parametrise higher-order, collective operations.

2.1 Scalar types

As scalar types, we have Haskell's standard fixed-precision integer type `Int`, the floating-point types `Float` and `Double`, characters `Char`, and the standard Boolean type `Bool`. In addition, we support the explicitly sized integral types `Int`n (signed) and `Word`n (unsigned) for $n \in \{8, 16, 32, 64\}$.

Moreover, in purely functional computations, it is crucial to have tuples to return multiple results. We regard tuple types $(\mathtt{t}_1, \mathtt{t}_2)$, $(\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3)$, $(\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3, \mathtt{t}_4)$, and so on as scalar types, provided the $\mathtt{t}_i$ are again scalar types.

2.1.1 Scalar expressions and functions

The result of a scalar GPU computation of type `GPU.Exp s` is always a scalar type `s`. Moreover, scalar GPU functions of type `GPU.Fun` $(\mathtt{s}_1 \rightarrow \mathtt{s}_2)$, map a scalar argument to a scalar result type — the notation `arg -> res` is standard in Haskell for a function that takes a value of type `arg` and returns a value of type `res`. Haskell represents n-ary functions in their curried form $\mathtt{arg}_1 \rightarrow \cdots \rightarrow \mathtt{arg}_n \rightarrow \mathtt{res}$, rather than their uncurried form $(\mathtt{arg}_1, \cdots, \mathtt{arg}_n) \rightarrow \mathtt{res}$. The curried form is more convenient in a pervasively higher-order language and, although our scalar GPU computations are first-order, we stick to Haskell's standard notation to integrate the GPU EDSL smoothly into the host language.

2.1.2 Polymorphism

Many GPU operations are *polymorphic,* but we need to restrict that polymorphism to range over the scalar types of GPU computations. Haskell programmers like their programs to be strongly statically typed (i.e., all type errors are caught by the compiler); hence, we need to ensure that the embedding of our GPU DSL represents all type constraints of the embedded language as host language constraints. Then, a type-incorrect GPU EDSL computation will result in a type-incorrect Haskell program and will be rejected by the Haskell compiler.

Haskell's standard mechanism to specify a set of types with associated polymorphic functions are *type classes.* We introduce a type class `GPU.IsScalar s` that has an instance for each scalar type `s`. For example, the full type for a unary scalar GPU function of a type `a` to `b` is the following:

```
(GPU.IsScalar a, GPU.IsScalar b)
  ⇒ GPU.Fun (a → b)
```

In a Haskell type signature, type class constraints are separated from the rest of the type by a double arrow ⇒.

2.2 Array types

GPUs are most efficient on massively data-parallel computations, which are typically composed from operations on images, streams, or arrays [2, 16, 8]. Our GPU EDSL provides a range of collective operations on multi-dimensional arrays. The choice of operations was informed by the *scan-vector model* [4], which is suitable for a wide range of algorithms [1], and of which Sengupta et al. demonstrated that these operations can be efficiently implemented on modern GPUs [21].

2.2.1 Arrays are shaped

We have multi-dimensional arrays of type `Array shape e`, where `shape` determines the type of *shapes* for the array and e the element type, which needs to be one of the scalar types discussed in Section 2.1. The shape of a zero-dimensional array (i.e., a *singleton* array) is `()`, the shape of a one-dimensional array is `Int`, of a two-dimensional array is `(Int, Int)`, of a three-dimensional array is `(Int, Int, Int)`, and so on.

Singleton arrays of type `Array () e` play an important role in our framework as collective GPU computations of type `GPU a` can only yield one or more arrays; they cannot produce scalar values. Hence, collective operations resulting in scalars —such as summing the elements of an array— result in a singleton.

2.2.2 Host versus device memory

Values of type `Array shape e` can be manipulated from non-GPU Haskell code; for example, such an array may be initialised from disk, and hence, these arrays are allocated in main memory. However, GPUs usually operate on data located in memory that is physically distinct from that of the host CPU, and arrays need to be explicitly transferred from host to device memory, before they can be processed in collective GPU operations.

We use a separate type `GPU.Arr shape e` to represent arrays that are available in GPU memory. To use a CPU `Array` in a GPU computation, we need the following function:

```
use :: IsScalar e => Array dim e -> GPU (Arr dim e)
```

Data transfer between CPU and GPU is expensive and to make GPU off-loading worthwhile, we need to amortise the data transfer by a sufficiently increased execution speed on the GPU. To reduce the overhead of transferring data, we avoid transferring data from the GPU to the CPU and back to the GPU on successive and —if sufficient memory is available— also on temporally close GPU computations, where one computation consumes the result of a previous GPU computation.

2.3 Collective versus scalar computations

As mentioned before, we use a two-level language for GPU computations. The outer level describes a sequence of collective array operations of type `GPU`. The inner level comprises scalar operations of type `GPU.Exp`, which we use to parametrise the collective operations.

2.3.1 Parametrised collective operations

A concrete example for the interaction of these two levels is the following code fragment:

```
GPU.map (λx → x + 1) arr  – of result type
                          – GPU (Arr shape Int)
```

The collective operation `GPU.map` applies the scalar function `(λx → x + 1)` of type `GPU.Fun (Int → Int)`, which is defined anonymously via a *lambda abstraction*, to every element of the array `arr`. In Haskell, a lambda abstraction of the form `λx → e` is a unary function with argument `x` and function body `e`.

The two-level nature of this expression follows from its type structure. The scalar function `(λx → x + 1)` operates on integer values manipulated on the GPU, i.e., values of type `GPU.Exp Int`, whereas the collective operation `GPU.map` yields a value of type `GPU (Arr shape Int)`, which tags it as a collective operation executing on the GPU and producing an array of integers.

2.3.2 Scalar GPU expressions and functions

The use of specialised types for scalar GPU computations, `GPU.Exp a`, and GPU functions, `GPU.Fun (a → b)`, enables us to restrict the range of scalar GPU operations to be a subset of what is support by Haskell in general. This is crucial to prevent the use of language features of which it is unclear whether they can be efficiently compiled to GPU kernel code at all, such as unbound recursion, higher-order functions, closures, and so forth.

Currently, we support standard integer and floating-point arithmetic, comparisons, Boolean and bitwise logical operations, conditionals as well as array indexing and obtaining array shape information. As far as possible, we use overloading to keep the standard Haskell operator symbols for these operations, such as `x + 1` in the previous example. However, in some cases, the standard Haskell type class definitions force us to adopt alternative symbols. In those case, we append a star character `*` to the standard Haskell symbols; for example, we use `(==*)` and `(/=*)` for testing equality and inequality, respectively.

2.3.3 Types enforce flat data parallelism

The native GPU programming model only supports a single level of parallelism. It can execute many instances of a kernel function in parallel, but each of these instances itself needs to be purely sequential code. This is unlike the classical use of collective operations, such as `map` in Haskell. For example, the type of the *standard list version* of `map` in Haskell is

```
List.map :: (a → b)  – function applied to elements
         → [a]       – input list
         → [b]       – result list
```

where `List.map f [x`$_1$`, ..., x`$_n$`] = [f x`$_1$`, ..., f x`$_n$`]`.[1] Here nothing prevents us from nesting collective operations; for example, we may use `List.map (List.map f) myListOfLists` to apply the function `f` to every element in a list of lists.

In contrast to Haskell's `List.map`, our collective GPU operation `GPU.map` is much more restrictive. It's type is

```
GPU.map :: (IsScalar a, IsScalar b)
        ⇒ GPU.Fun (a → b)       – mapped function
        → Arr shape a           – input array
        → GPU (Arr shape b)     – result array
```

This is restricted in two ways. Firstly, the type variables `a` and `b` may only be instantiated to scalar types, due to the `IsScalar` type constraints — c.f., Section 2.1. Secondly, `GPU.map` can only map unary GPU functions, of type `GPU.Fun (a → b)`. This prevents the use of arbitrary Haskell computations in GPU code and it prevents the use of collective GPU operations (as their result type is always of the form `GPU t`). We need to forbid the use of arbitrary Haskell computations as it would be difficult to compile, say, the recursive traversal of a graph structure and higher-order functions to efficient GPU code. Moreover, we need to forbid the nested use of collective GPU operations as nesting leads to multiple levels of parallelism, which we also cannot easily map to the strictly synchronous data-parallel GPU execution model. However, with the support of the operations from the scan-vector model (c.f., Section 2.2), the programmer may explicitly encode nested and irregular computations.

Overall, the two-level structure of our array EDSL in Haskell reflects the two-level structure of the GPU execution model, where sequential kernel functions are invoked at multiple data-parallel instances. Moreover, our use of a two-level language is loosely related to the idea of *algorithmic skeletons* where parallel programs are decomposed into templates encoding a fixed parallel structure and scalar code that is injected at predefined points into that structure [5]. However, we overall have a three level structure with (1) vanilla Haskell code executed on the CPU, (2) collective array operations for the GPU, and (3) scalar GPU kernel code.

2.4 Monads to control GPU code execution

The restriction of scalar GPU code to computations over values of type `GPU.Exp a` not only prevents nesting, but also statically prevents the use of side-effecting operations, such as I/O. To explain this in more detail, we need to take a brief detour and look at the concept of *monads* in Haskell [25]. In essence, monads are type constructors with an associated set of functions that abide by certain algebraic rules.

2.4.1 Isolating side effects

The best known monad in Haskell is the `IO` monad, which Haskell applies to control the use of a range of side effects, including I/O operations and mutable objects, in an otherwise purely functional language. The basic idea is that all I/O operations explicitly mention the type of the `IO` monad in their type signature and that all functions that directly or indirectly use these I/O operations need to do the same. This type discipline is enforced by the Haskell compiler and enables spotting functions that may have side-effects. These functions are then excluded from appearing in pure, side-effect free computations. By requiring that `GPU.Exp a` encapsulates only pure computations that do not mention the `IO` monad, we can statically guarantee the absence of side effects in parallel code, thus avoiding race conditions and similar problems. Any program that is in violation of this policy will be rejected by the Haskell compiler.

2.4.2 I/O in Haskell

An I/O operation in Haskell which returns a value of type `a` has type IO a. For example, `getChar` has type `IO Char`, and `putChar` has type `Char → IO ()` — it has a side effect, but doesn't actually return a value (that is, it always returns the value `()` of the singleton type `()`, much like a C function without a result returns `void`).

Haskell's `do` notation allows us to conveniently combine a sequence of expressions of monadic type. For example, the function `putStr :: String → IO ()` emits a string to standard out. To combine several `putStr` actions into a new `IO` action, we can use a `do` expression to sequence the I/O

[1] In Haskell, we write `e :: t` to express that an expression or function `e` has type `t`.

operations, just like in an imperative language (we need no semicolon or curly braces as layout has meaning in Haskell):

```
do
  putStr "Hello"
  putStr " and "
  putStr "Goodbye"
```

The type of the entire `do` expression is the same as the type of the last expression in the list. In this case, all the `IO` actions happen to be of type `IO ()`; so, the whole `do` expression has this type as well.

If we want to bind the return value of an I/O operation to a variable for later use, we use the ← notation. The function `isSameChar` reads two characters from standard in, and returns `True` if both are identical; `False` otherwise.

```
isSameChar :: IO Bool
isSameChar =
  do
    c1 ← getChar
    c2 ← getChar
    return (c1 == c2)
```

The first character is bound to the variable `c1`, the second to `c2`. If we omit `return` in the last line, the type checker would complain and tell us that it found a Boolean expression (`c1 == c2`) where it expected an expression of type `IO Bool`; we need to invoke `return` to inject the result into the monad type `IO`. The function `return` has type

```
return :: Monad m ⇒ a → m a
```

This signature is to be read as "if type constructor `m` is a member of the type class `Monad`, then `return`, given a value of type `a`, returns a value of type `m a`".

The ← notation only allows us to temporarily "unpack" the value inside the monad. If we want to pass it on, however, we have to inject it back into the monad using, for example, `return`. There is *no general monad operation* inverting the effect of `return`, like `impossible :: Monad m ⇒ m a → a`. In this way, the type system ensures that every calculation whose value depends on the result of an `IO` operation will itself also always have type `IO`.

2.5 Running GPU code

As the signature of `GPU.map` (in Section 2.3.3) indicates, the result type of collective operations always has the form `(GPU t)`, where `t` is the type of the computed result. This raises two questions: (1) How do we compose two collective operations, and (2) how do we use a collective operation in Haskell code executed on the CPU? The answer to both questions rests on the fact that the type `GPU` is a monad; specifically, it is an instance of the standard Haskell `Monad` type class which we briefly discussed in the previous subsection.

Consequently, we can use Haskell's `do` notation to compose collective array operations, just like we combined `IO` operations, and thus, answer the first question. A function performing two map operations followed by combining the two resulting arrays into an array of the element-wise pairs of the intermediate results might be defined as follows:

```
zipIncAndDouble :: (Num a, IsScalar a)
                ⇒ Arr sh a → GPU (Arr sh (a, a))
zipIncAndDouble arr
  = do
      arr1 ← GPU.map (λx → x + 1) arr
      arr2 ← GPU.map (λx → 2 * x) arr
      GPU.zip arr1 arr2
```

Similarly to the `isSameChar` example of Section 2.4, we bind the result values of two monadic operations `GPU.map` to variables (`arr1` and `arr2`, within the `GPU` monad), and combine `arr1` and `arr2` to a new value. Here, we don't need to explicitly inject the result back into the monad as `GPU.zip` has type

```
GPU.zip :: (IsScalar a, IsScalar b)
        ⇒ Arr shape a
        → Arr shape b
        → GPU (Arr shape (a, b))
```

That is, the result value is already a value of type `GPU`. `GPU.zip` is comparable to Haskell's standard `List.zip` operation of type `[a] → [b] → [(a, b)]`, where we have

```
zip [x1, x2, ...] [y1, y2, ...] =
  [(x1, y1), (x2, y2), ...]
```

Composition of collective GPU operations in monadic notation requires the programmer to fix the order of execution of multiple GPU operations — in `zipIncAndDouble`, we first increment and then double the input array. Explicit sequencing gives programmers control over the number of live temporary arrays. This can be important in GPU code, which often operates on very large arrays and executes on a resource-constrained device. However, more important is that the `do` notation enables us to *name and share* intermediate results, `arr1` and `arr2` in `zipIncAndDouble`. Otherwise, we may duplicate expensive subcomputations when reflecting GPU computations for subsequent compilation to device code [6] — we shall discuss the use of reflection, to implement the EDSL, in slightly more detail in Section 3.2.

Let's turn to the second question raised at the beginning of this subsection: how do we execute GPU code from CPU code? Computational monads usually come with a run function that executes a monadic computation. In our case, this is

```
GPU.run :: GPU a → GPU.Result a
```

So, given `arr :: Arr sh Int`, we have that

```
GPU.run (zipIncAndDouble arr) :: Array sh (Int, Int)
```

Note also how `GPU.run` marshals the array resulting from the GPU computation (type constructor `Arr`) back to a CPU array (type constructor `Array`). The corresponding type transformation, which applies to individual arrays, but also tuples of arrays in case of multiple results, is formalised by the type family `GPU.Result`.

The GPU computation is *shape polymorphic;* i.e., it applies to arrays of any shape as `sh` is a type variable that may be instantiated to any of the array shapes described in Section 2.2.

2.6 Examples

To illustrate the use of our GPU EDSL in Haskell, we briefly discuss two simple example applications.

2.6.1 SAXPY

As a first example, consider the *Scalar Alpha X Plus Y* (SAXPY) operation of the *Basic Linear Algebra Subprograms* (BLAS) package. In plain Haskell, we might code it as follows:

```
saxpy_CPU ::
  Float → [Float] → [Float] → [Float]
saxpy_CPU alpha xs ys =
  zipWith (λx y → alpha * x + y) xs ys
```

The function zipWith takes a binary function as a first argument, and two lists as second and third argument. It applies the function to the first elements of both lists, the second elements, and so on, returning the list of results:

```
zipWith f [x1, x2, ...] [y1, y2, ...]
  = [f x1 y1, f x2 y2, ...]
```

By using GPU arrays instead of lists and collective GPU operations wrapped into a GPU.run, we implement SAXPY as Haskell GPU code as follows:[2]

```
saxpy_GPU :: GPU.Exp Float
          → Array DIM1 Float
          → Array DIM1 Float
          → Array DIM1 Float
saxpy_GPU alpha xs ys
  = GPU.run $ do
      xs' ← use xs
      ys' ← use ys
      zipWith_GPU (λx y → alpha*x + y) xs' ys'

zipWith_GPU f xs ys
  = do
      xys ← GPU.zip xs ys
      GPU.map (uncurry f) xys
```

Except for the need to explicitly sequentialise collective operations, Haskell GPU code enjoys a similar level of abstraction as conventional Haskell.

2.6.2 Sparse matrix-vector multiplication

As a second, more complex example, consider the multiplication of a sparse matrix with a dense vector. We represent sparse matrices in the *compressed row format* [4, 3]. The matrix consists of an array of matrix rows, where only the non-zero elements of each row are explicitly stored, paired with their column index. For example, the following matrix

$$\begin{pmatrix} 7 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 2 & 3 \end{pmatrix}$$

corresponds to the value

```
[[(0,7.0)],[],[(1,2.0),(2,3.0)]]
```

in compressed row representation (note that we start indexing with 0). However, we represent the matrix not as array of arrays —since we don't directly support irregular structures– but use a second array which contains the number of non-zero elements of each row. Our example matrix is, in this format, represented as the pair

```
([1,0,2],
 [(0,7.0),(1,2.0),(2,3.0)])
```

and we use the following type synonyms for values of sparse matrix type:

```
type Vector       = Array DIM1 Float
                    – dense float vector
type Segd         = Array DIM1 Int
                    – segment descriptor
type SparseMatrix =
  (Segd, (Arr DIM1 Int, Arr DIM1 Float))
```

[2] The standard Haskell operator $ is defined as f $ x = f x. Moreover, the function zipWith_GPU is pre-defined as GPU.zipWith, but we'd like to restrict ourselves here to previously introduced GPU operations.

Sparse-matrix vector multiplication in this representation is coded as follows — again, see [3] for a more detailed explanation of the algorithm; GPU.bpermute extracts those values from the vector which have to be multiplied with the corresponding non-zero matrix values. zipWith_GPU performs this multiplication, and finally GPU.fold_segmented adds all the products that correspond to the values of one row.

```
smvm_GPU :: SparseMatrix → Vector → Vector
smvm_GPU (segd, (inds, values)) vector
  = GPU.run $ do
      segd'   <- use segd
      inds'   <- use inds
      values' <- use values
      vector' <- use vector
      vectorValues ← GPU.bpermute vector' inds'
      products     ←
        zipWith_GPU (*) vectorValues values'
      sum_segmented products
  where
    sum_segmented = GPU.fold_segmented (+) 0 segd
```

3. Code generation

We are using NVIDIA's CUDA [16] framework to produce GPU code. CUDA is a C/C++ dialect that supports general-purpose computations on GPUs by enabling the data-parallel execution of specially-marked C functions that have no global side-effects and abide by some further restrictions. The restrictions on code that can be efficiently executed on current GPUs are sufficiently severe that compiling plain Haskell for execution on a GPU is not feasible — in particular, we can neither use function pointers nor garbage collection.

3.1 An online code generator

The inability to execute arbitrary Haskell programs on GPUs implies that we cannot execute expressions of the form GPU.run gpuCode directly on a GPU, where gpuCode is in our domain-specific language for GPU computations. Instead, the domain-specific language is sufficiently limited that we can compile it to CUDA code using a special-purpose compiler, called GPU.gen.

GPU.gen is an online compiler; i.e., it turns code in our domain-specific language at program runtime into GPU code and dynamically links it into the running Haskell program. This happens in four steps:

1. *GPU program reflection:* At runtime, we need to obtain a representation of the GPU program as an abstract syntax tree.
2. *CUDA code generation:* We need to turn the collective array operations with result type GPU as well as any GPU.Exp computations used to parametrise higher-order collective operations into CUDA C code.
3. *CUDA code compilation:* We use the CUDA compiler to generate an object file from the previously produced CUDA C code.
4. *GPU code linking:* We dynamically link the object file generated by the CUDA compiler into the running Haskell program.

The third and fourth stage of GPU.gen are straight forward uses of existing infrastructure code, namely the CUDA environment [16] and the Haskell plugins library [18]. We summarise the approach that we take for the first and second stage in the next subsection.

To avoid unnecessary overhead, we ensure that a single GPU sub-program is only compiled once per program run. The current CUDA environment provides the facility for dynamic linking at the driver level, which makes it possible to load a GPU sub-program multiple times after it has been compiled once.

3.2 Program reflection & code generation

We represent the abstract syntax tree of embedded GPU programs by means of *generalised algebraic data types* (GADTs) in Haskell [19]. Those GADTs ensure that we can only express GPU programs that are type correct. In other words, type errors in embedded GPU code are spotted at Haskell compile time, instead of being deferred until GPU code compile-time, which is Haskell run-time.

The various collective array operations (such as, `GPU.map`, `GPU.zip`, and so on) do not directly invoke the specified functions, but instead construct fragments of the abstract syntax of the GPU program that represent the application of the respective array operations. Similarly, the monadic combinators that we indirectly use by way of the `do` notation are also reflected into GADT constructors. Finally, we use the so-called *dictionary view* enabled by Haskell type classes to reflect `GPU.Exp` expressions into abstract syntax trees [11, 6].

Overall, in an expression of the form `GPU.run gpuCode`, the value of `gpuCode` is actually an abstract syntax tree describing the intended GPU computation. `GPU.run` turns the abstract syntax tree over to `GPU.gen` for code generation and plugin loading, and then, executes the newly loaded code. Code generation for unsegmented and segmented data-parallel array operations is based on the scan vector model [4] and it's adaptation to GPU programming [22].

The details of reflecting GPU computations into their abstract GADT representation, the use of type classes to realise overloading, and the details of the compilation process of the abstract syntax of GPU computations to CUDA code is beyond the scope of this paper. We plan to describe them in detail elsewhere.

4. Implementation status & preliminary results

Our implementation of `GPU.gen` is still work in progress. It performs code generation in roughly two phases, which are linked by an intermediate language that is closer to CUDA than the collective array operations of Section 2. We still need to implement most of the first phase as well as add more source-to-source optimisations to the second phase. Consequently, the *preliminary* benchmark results reported in this section have been obtained with code where manual translation has replaced still missing compiler functionality.

We executed the benchmarks on two different CPUs: an AMD Sempron 3400+ 1.8GHz (running Debian Linux) and a Intel Xeon E5405 2.0GHz (running Ubuntu Linux Gutsy). We also executed them on two different GPUs: an NVIDIA GeForce 8800GTS GPU (96 cores with 320MB memory) and an NVIDIA Tesla S1070 (four Tesla GPUs with overall 960 compute cores and 16GB memory of which the benchmarks use only *one Tesla GPU with 240 cores and 4GB of memory*). Haskell code was compiled with the Glasgow Haskell Compiler (GHC) 6.10.1 (and optimised with `-O3`) and CUDA C code was compiled with CUDA 2.0.

We tested three small benchmark programs: SAXPY and `smvm` from Section 2.6 as well as the valuation of Black-Scholes Call Options. For each of the two CPUs, we measured the performance of plain Haskell code entirely executed on the CPU and using `Data.Array.Storable` (a Haskell library providing low-level, C-like arrays) — we call this "Plain Haskell, only CPU" in the benchmark figures. The plain Haskell code is not multi-threaded nor vectorised. For each of the two GPUs, we measured the performance of GPU code generated from our embedded GPU array language. We obtained two measurements: firstly, just the GPU computation time (we call this "Comp only") and, secondly, the CPU-GPU data transfer time in addition to the GPU computation time (we call this "Comp + Transfer"). For, the Black-Scholes benchmark, we also measured a plain CUDA implementation that is being distributed by NVIDIA as part of the CUDA SDK. This implementation comes with both call options and put options; we disabled the put options for this benchmark. Neither benchmark includes the runtime of the online compiler `GPU.gen`; we currently cannot include it due to the mentioned manual intervention.

The results are depicted in Figure 1. (NB: The time axis uses a logarithmic scale.) In all three benchmarks, both GPUs widely outperform both CPUs in their computational capacity. When adding the CPU-GPU data transfer costs, which are very high in these data-intensive benchmarks, the picture gets more varied. The NVIDIA Tesla system still delivers the absolute best performance in all benchmarks. In the `smvm` and Black-Scholes benchmarks, the GeForce 8800GTS is consistently faster than both CPUs, but in SAXPY this is not the case. (The increase in runtime of the GeForce 8800GTS for Black-Scholes by around 60mil is due to the limited memory of the GeForce 8800GTS and the resulting need to execute the computation in multiple phases.)

Our measurements suggest that GPU acceleration for array code embedded in Haskell can be very beneficial, even for rather small and data-intensive kernels. We expect the ratio between CPU-GPU transfer time and GPU compute time to be much more favourable in realistic applications, where more arithmetic operations are performed on every transferred data element.

For Black-Scholes, the plain CUDA implementation is about 2.5 times faster than the CUDA generated from the Haskell-embedded GPU language. We believe that this gap is due to better scheduling of parallel thread executions in the CUDA SDK version. In return, our embedded GPU language provides a significantly higher level of abstraction.

5. Related Work

There is a significant and increasing interest in the use of GPUs for general purpose computations [17]. This led to the development of programming models for general-purpose GPU code that are more convenient than the graphics-oriented development environments of high-level shader languages. An early example is Sh, which is based on C++ meta-programming [14]. Sh shares a number of architectural traits with our embedded Haskell GPU language: they also use reflection to dynamically compile an embedded domain-specific language for execution on GPUs. However, Sh is more graphics-centric than our approach and works with an entirely different host language, namely C++.

Less graphics-oriented are Brook [2] and Scout [15], although the latter is aimed at visualisation. Brook, much like CUDA, is based on C and supports the definition of

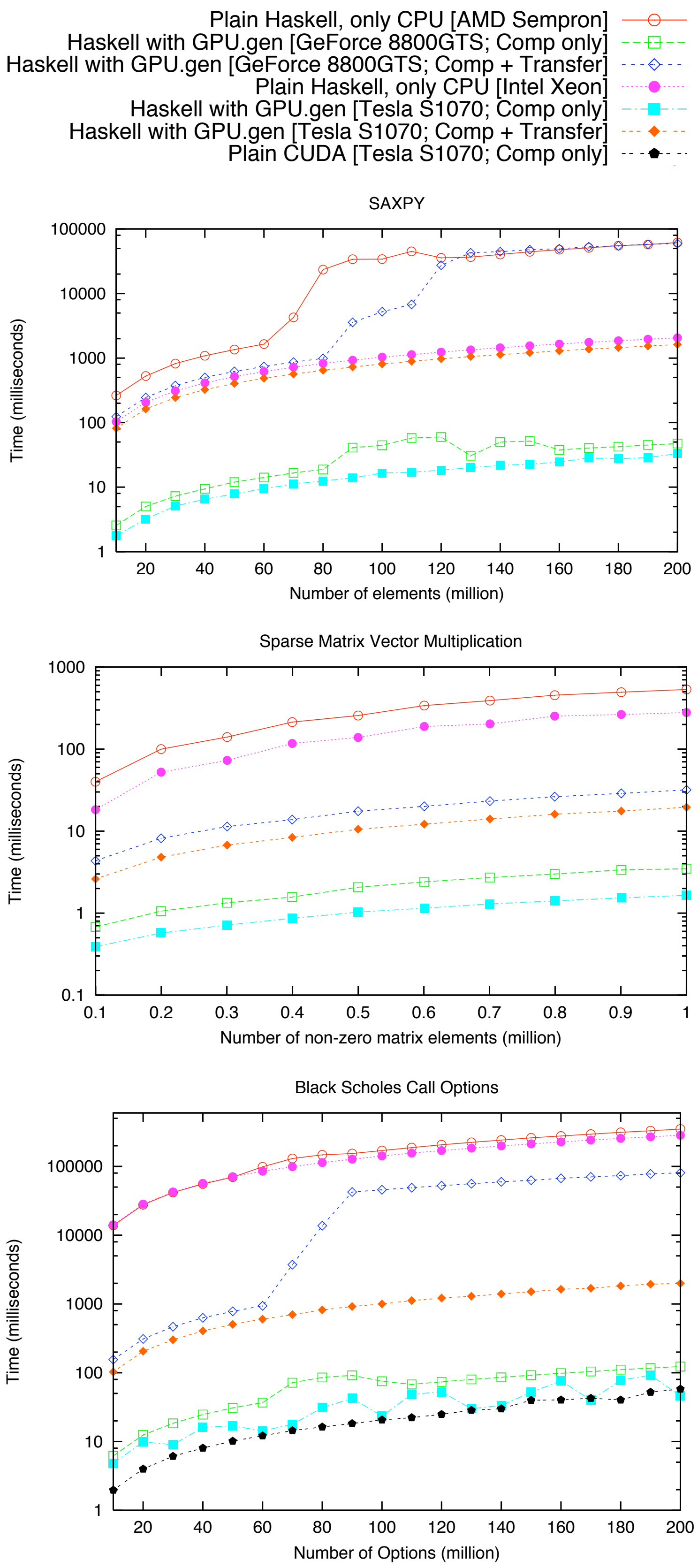

Figure 1. CPU and GPU performance for (a) `saxpy`, (b) `smvm`, and (c) Black-Scholes

functions that are compiled for GPU execution and invoked in multiple data-parallel instances from CPU code. As in CUDA, functions for the GPU cannot have global side effects and are also otherwise restricted to meet the constraints of GPU hardware. Programmers also need to be aware of memory-access, alignment, and scheduling constraints; otherwise, performance may suffer very significantly [20]. Finally, as GPU code needs to be in separate functions, program decomposition is partially constrained by hardware considerations.

Most programming environments for general-purpose programming of GPUs are based on C and C++. This is in the tradition of graphics-oriented GPU programming environments, such as the DirectX High-Level Shader Language and the OpenGL Shader Language, and eases adoption by programmers familiar with systems programming in C and C++. However, it also leads to the obvious mismatch between a host language where pointers and side effects are central and a massively parallel programming environment, where side effects need to be tightly controlled. A notable exception to the use of C/C++ is PyGPU, which is a GPU language embedded in the scripting language Python [12].

To the best of our knowledge, the only general-purpose GPU programming environment in a functional language, besides ours, is Obsidian [24, 23]. Like our system, Obsidian is a language embedded in Haskell. It is based on the hardware description language Lava, and judging by the examples in [24] and [23], it is aimed at a lower-level of abstraction than our approach. At least, the examples of functions coded in Obsidian include a prefix scan and reduction operations, which are primitive operations in our model.

The first EDSL for GPU programming in Haskell was Elliott's Vertigo [6]. It was aimed at older GPUs that still distinguished between vertex and pixel shaders in hardware and with a much more limited instruction set than present GPUs. The Vertigo paper presents the use of the EDSL for graphics-centric programming and with the aim of generating graphics shaders. However, the basic compilation technique might also be used in a more general setting.

The array operations in our GPU EDSL are informed by the operations of the scan-vector model [4]. This model is also the basis for the implementation of nested data-parallel languages, such as NESL [1]. Currently, our GPU EDSL strictly supports flat data parallelism only. However, given our recent progress in the implementation of a vectorisation transformation in the Glasgow Haskell Compiler [10], which maps nested to flat data parallelism, we hope to be able to eventually support nested data parallelism in GPU computations, too.

6. Conclusion

We outlined the design of an embedded domain-specific language (EDSL) for GPU computations in Haskell. The EDSL is structured as a two level language, where the outer level sports array operations along the lines of the scan-vector model and the inner level comprises simple scalar computations that can be efficiently implemented as GPU kernels. The embedding of GPU computations is such that they adhere to a strict type discipline, where the compiler spots and rejects a wide range of type errors including the use of side-effects in parallel code and any attempt to nest parallel computations. We implement the GPU EDSL by an online compiler targeting the CUDA framework for code generation. The compiler is still work in progress, but preliminary benchmarks of simple algorithms showed a dramatic performance improvement of Haskell GPU code over low-level Haskell array code executed on the CPU only.

References

[1] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[2] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[3] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: a status report. In *DAMP '07: Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, pages 10–18, New York, NY, USA, 2007. ACM.

[4] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Supercomputing '90: Proceedings of the 1990 conference on Supercomputing*, pages 666–675. IEEE Computer Society Press, 1990.

[5] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* The MIT Press, 1989.

[6] Conal Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 Haskell Workshop*. ACM Press, 2004.

[7] GPGPU. General Purpose Computations on GPUs. http://www.gpgpu.org.

[8] Khronos OpenCL Working Group. The OpenCL specification. Technical report, Khronos Group, 2008. `http://www.khronos.org/opencl/`.

[9] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.

[10] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In Ramesh Hariharan, Madhavan Mukund, and V Vinay, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[11] Oleg Kiselyov. Most optimal symbolic differentiation of compiled numeric functions. `http://okmij.org/ftp/Computation/Generative.html#diff-th`, 2006.

[12] Calle Lejdfors and Lennart Ohlsson. Implementing an embedded GPU language by combining translation and generation. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1610–1614. ACM Press, 2006.

[13] Michael Macedonia. The GPU enters computing's mainstream. *Computer*, pages 106–108, October 2003.

[14] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.

[15] Patrick McCormick, Jeff Inman, James Ahrens, Jamaludin Mohd-Yusof, Greg Roth, and Sharen Cummins. Scout: a data-parallel programming language for graphics processors. *Parallel Comput.*, 33(10-11):648–662, 2007.

[16] NVIDIA. CUDA Compute Unified Device Architecture Programming Guide 2.0, July 2008. http://www.nvidia.com/cuda.

[17] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[18] André Pang, Don Stewart, Sean Seefried, and Manuel M. T. Chakravarty. Plugging Haskell in. In *Haskell '04:*

Proceedings of the ACM SIGPLAN workshop on Haskell, pages 10–21. ACM Press, 2004.

[19] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ACM SIGPLAN Intl. Conf. on Functional Programming (ICFP'06)*. ACM Press, 2006.

[20] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-Mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP '08: Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82. ACM, 2008.

[21] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan Primitives for GPU Computing. In *Graphics Hardware 2007*, pages 97–106. ACM, August 2007.

[22] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 97–106. Eurographics Association, 2007.

[23] Joen Svensson, Koen Claessen, and Mary Sheeran. Obsidian: An embedded language for data-parallel programming. In *Proceedings of International Workshop on Designing Correct Circuits*, 2008.

[24] Joen Svensson, Mary Sheeran, and Koen Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In *Proceedings of 20th International Symposium on the Implementation and Application of Functional Languages*, 2008.

[25] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.