Low-Level Functional GPU Programming for Parallel Algorithms

Martin Dybdal  Martin Elsman
University of Copenhagen, Denmark
dybber@dybber.dk, mael@diku.dk

Bo Joel Svensson  Mary Sheeran
Chalmers University of Technology, Sweden
joels@chalmers.se, mary.sheeran@chalmers.se

Abstract
We present a Functional Compute Language (FCL) for low-level GPU programming. FCL is functional in style, which allows for easy composition of program fragments and thus easy prototyping and a high degree of code reuse. In contrast with projects such as Futhark, Accelerate, Harlan, Nessie and Delite, the intention is not to develop a language providing fully automatic optimizations, but instead to provide a platform that supports absolute control of the GPU computation and memory hierarchies. The developer is thus required to have an intimate knowledge of the target platform, as is also required when using CUDA/OpenCL directly.

FCL is heavily inspired by Obsidian. However, instead of relying on a multi-staged meta-programming approach for kernel generation using Haskell as meta-language, FCL is completely self-contained, and we intend it to be suitable as an intermediate language for data-parallel languages, including data-parallel parts of high-level array languages, such as R, Matlab, and APL.

We present a type-system and a dynamic semantics suitable for understanding the performance characteristics of both FCL and Obsidian-style programs. Our aim is that FCL will be useful as a platform for developing new parallel algorithms, as well as a target-language for various code-generators targeting GPU hardware.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features

General Terms Languages, Performance

Keywords Type systems, data-parallel languages, GPU programming, push arrays, pull arrays, iteration schemes, array-programming, hierarchical machine models.

1. Introduction
In recent years, several languages for general purpose, data-parallel computation on GPUs have been suggested [3, 5, 6, 12, 13, 19]. Most of these language developments have focused on providing users with high-level specifications of programs and performing a range of automatic optimizations. Often no cost-model is specified, and the language is thus a black box for users who want to reason about the performance of their programs. Parallel algorithms researchers are sidelined, as it is hard to reason about the actual efficiency and performance characteristics of algorithms. The user is decoupled from the hardware model, and cannot be sure whether an operation will result in a memory transaction or not. This makes unexpected performance hits hard to debug. Also, some algorithms require memory patterns not supported by the prevalent set of primitives, or depend critically on hardware parameters that these languages do not expose [4]. This is a shame. We want more algorithms researchers to work on parallel algorithms, and they need better languages to do their work.

In the GPU niche of data-parallel languages, Obsidian is an exception [19], allowing for playfulness and invention on the low-level where you have (almost) complete control over the GPU, and still allowing computations to be composed efficiently using so called pull arrays and push arrays. These arrays are not directly stored in a region of memory, but are rather representations of array-computations. This means that most array operations are cheap: they do not incur the overhead of writing a modified array to memory, but modifies the underlying symbolic array-computation directly. Obsidian uses a multi-staged compilation approach, which allows users to use Haskell as a meta-language generating Obsidian expressions. This can for instance be used to generate all the statements of an unrolled loop, or to precompute certain values already at code-generation time.

We present FCL, a reimplementation of Obsidian with an external syntax implemented in Haskell2010 as a self-contained compiler1. With FCL, we extend on the work on Obsidian; eliminating the need of using meta-programming techniques in program development, and introducing new operators and language constructs to maintain the same expressive power. The embedded nature of Obsidian also had its drawbacks, especially if used as an intermediate language, which is another reason this project came to be.

In both Obsidian and FCL, computations are polymorphic in their mapping to executions on the GPU hardware, by the use of level-annotations in array types. We have developed a dynamic operational semantics for FCL that details the computational model and makes it clear how the different levels map to various iteration schemes on the GPU.

The rest of the paper is structured as follows. Section 2 explains pull and push arrays. In Section 3, we introduce FCL through three example programs: array reversal, matrix transpose, and parallel reduction. Section 4, we demonstrate that FCL is able to generate efficient OpenCL-code. In Section 5, we do a rigorous introduction to FCL, defining its type system and dynamic semantics. Finally, we conclude in Section 7.

We did not find space for an introduction to GPU programming, we refer the reader to the OpenCL and CUDA programming guides by AMD [1] and NVIDIA [15].

---

1FCL is available at http://github.com/dybber/fcl

---
2. Pull and Push Arrays

FCL inherits pull and push arrays from Obsidian [8]. As mentioned in the introduction, these are not actual arrays manifested in memory, but are instead delayed array computations that describe how to produce an array. When the result of a pull or push array computation is written to memory, we say that the array has been materialized.

The two types of arrays complement each other: pull arrays allow array indexing, but array concatenation is inefficient. Push arrays on the other hand allow for efficient concatenation, but disallow array indexing. Below we will introduce a simplified view of the pull and push array representation in Obsidian, using Haskell notation.

2.1 Pull Arrays

Let Idx be the type of array indices and array lengths. A pull array with elements of type a is then represented as a length paired with an index:

```
type Pull a = (Idx, Idx -> a)
```

Materializing such an array in memory is performed by evaluating the function at each index and generating the code associated with writing the result to memory.

Operating on the individual elements of the array can be done without materializing the array. Let arr be a function of type Idx -> Double, multiplying each array element by two can then be done by building a new pull array: \(i \rightarrow 2.0 \cdot \text{arr}(i)\).

2.2 Push Arrays

Push arrays, on the other hand, already carry with them an iteration scheme, or iteration scheme, decided by the creator of that push array. A push array is represented by a function that can construct a new array by mapping the given function at each index and generating the code associated with writing the result to memory.

Let us demonstrate, and create a block-level version of reverse.

```
sig reverse : [a] -> [a]
fun reverse arr =
  let n = length arr
  in generate n (fn i => index arr (n - i - 1))
```

This program is implemented using the function generate, a language primitive that creates a new array by mapping the given function over the index-space \([0; n - 1]\). The program here cannot be compiled directly to GPU code, as it does not mention how it should be mapped to sequential or parallel loops. The arrays in this example are pull arrays, and are identified by types of the form \([a]\), where \(a\) is a type variable, representing an arbitrary non-function type. To compile an FCL program into a kernel, we require the user to add an iteration scheme, detailing how this kernel should be mapped to the threads of the GPU. Such iteration schemes are annotated by a level, which can be either thread (sequential execution), warp, block, or grid. The iteration scheme is added using a function called push. Let us demonstrate, and create a block-level version of reverse.

```
sig revBlock : [a] -> [a]<block>
fun revBlock arr = push <block> (reverse arr)
```

Notice how the iteration scheme is reflected in the array type, \([a]<block>\). This is a push array (from Obsidian). If we were to compile this function, FCL would generate a kernel reversing the entire array using a block-level computation. That is, the computation would only run in a single block, and thus only run on a single of the GPUs streaming multiprocessors. To distribute across several blocks, the input-arrays have to be partitioned and the resulting reversed array-chunks need to be concatenated back together again in the right order. In this case, the order of the chunks also needs to be reversed before concatenation.

```
sig revDistribute : int -> [a] -> [a]<grid>
fun revDistribute chunkSize arr =
  splitUp chunkSize arr
  |> map reverseBlock
  |> reverse
  |> concat chunkSize
```

The operator \(|>\) is reversed function application from F# and Elm, also known as forward-pipe. Notice that the same reverse function can be used both to reverse the order of elements and the order of the blocks. The operation concat is what distributes the computation across a grid of blocks, thereby raising the level from block to grid. This is also evident from the type of concat, where 1+level, unifies with levels of one level higher in the hierarchy (details are given in Section 5).

```
concat : int -> ([a]<level>] -> [a]<1+level>
```
This means that each subarray is executed in a separate block, and 
and concat makes sure that each block writes its result to adjacent 
subsections of the array it returns. Alternatively we could have 
applied push <grid> directly to the primitive reverse function, 
to add a grid-level iteration scheme to the array, but that is only 
possible in simple cases, where there is no dependencies between 
threads and we do not need to manipulate the amount of data 
processed by each block or how results are combined. Neither 
split2Dgrid nor concat2Dgrid is a primitive of FCL, and more complicated 
tiling and interleaving can thus be implemented, as we will see in 
the following example.

3.2 Transpose in Shared Memory

Now consider the problem of matrix transposition. In FCL we only 
have one-dimensional arrays, which means that a two-dimensional matrix 
must be represented as its flat representation together with 
number of columns and rows. We are planning to add support for 
multidimensional-arrays, see Section 7.

If we follow a naive approach we can transpose a two-dimensional matrix, using the following transpose function:

\[
\text{fun transpose } \text{cols} \text{ rows } \text{arr} = \\
\quad \text{generate } (\text{cols} * \text{rows}) \\
\quad (\text{fn } n \Rightarrow \\
\quad \quad \text{let } i = n \text{ div rows} \\
\quad \quad \quad j = n \text{ mod rows} \\
\quad \quad \quad \text{in index } \text{arr} (j * \text{rows} + i))
\]

If this version of transpose were to be executed in parallel on the 
GPU, it would lead to uncoalesced writes. When a group of GPU 
threads collectively read or write a section of memory, the memory 
transactions can be coalesced if they all fall into the same block of 
memory. In this case, when adding an iteration scheme to the final 
array, the final writes will always be in coalesced, but the indexing 
into the input array will not, and the reads from the input-array will 
thus not be able to coalesce and we will incur a huge performance penalty.

A more efficient approach is to chunk up the matrix in smaller 
2D tiles, transpose each tile in shared memory, before stitching 
the tiles back together again (in transposed order). This approach 
makes both reads and writes to global memory coalesced, as the 
threads can first collaborate on moving data to shared memory, and 
afterwards collaborate on copying data from shared memory to the 
output-array.

The important thing to note is that this reading/writing order is 
encapsulated in split2Dgrid and concat2Dgrid, and a library 
of such operations can be provided to users.

\[
\text{fun transposeTiled } \text{tileDim} \text{ cols} \text{ rows} \text{ mat} = \\
\quad \text{let } n = \text{cols} / \text{tileDim} \\
\quad m = \text{rows} / \text{tileDim} \\
\quad \text{in split2Dgrid } \text{tileDim} \text{ cols} \text{ n m mat} \\
\quad |> \text{map } (\text{transpose } \text{tileDim} \text{ tileDim}) \\
\quad |> \text{concat2Dgrid } \text{tileDim} \text{ n rows}
\]

This algorithm follows roughly the same structure as the 
reverse example. However, instead of splitting the linear input-
array into chunks (one following the other), we split and concate-
nate 2D tiles with the functions: split2Dgrid and concat2Dgrid.

3.3 Parallel Reduction

To implement a reduction kernel (prefix-sum kernel), we will per-
form a tree-reduction inside each work-group; this is implemented 
by splitting the subarray in two, and performing an element-wise 
sum of the two halves. This is very similar to what has previously 
been shown in Obsidian.

The FCL prelude provides the following functions for splitting 
arrays in two and joining arrays element-wise. These are not FCL 
primitives, but their implementation is standard and left out because 
of lack of space.

\[
\text{fun red } \text{f} \text{ arr} = \\
\quad \text{while } (\text{fn arr} \Rightarrow 1 \neq \text{lengthPull arr}) \\
\quad (\text{step } \text{arr}) \\
\quad \text{in push } \text{arr}
\]

Notice that the function is polymorphic in the level-variable \(\text{lvl}\). 
This makes it possible to postpone the decision of whether \(\text{step}\) 
will run sequentially or at one of the parallel levels of the hierarchy.

In Obsidian, we would have implemented this as a recursive 
function on the meta-level. Recursion on the meta-level would be 
possible in Obsidian, as the function is working on just a chunk of 
the array and we would statically know the chunk size. The meta-
level recursion in Obsidian would generate an unrolled loop.

In FCL we instead provide a built-in looping-construct, \text{while}, 
which accepts a \text{stop-condition} and \text{stepping} function as arguments 
as well as the initial array.

\[
\text{fun red } \text{f} \text{ arr} = \\
\quad \text{while } (\text{fn arr} \Rightarrow 1 \neq \text{lengthPull arr}) \\
\quad (\text{step } \text{arr}) \\
\quad \text{in push } \text{arr}
\]

\[
\text{Figure 1: Transpose in Shared memory. Figure by NVIDIA.}
\]
This will generate a while-loop, and automatically force values to shared memory between operations as well as performing a block-level synchronization between threads. In cases where the chunk size is known at compile time, we can use loop unrolling techniques to achieve the same code as if we had used Obsidian.

The while construct assumes that arrays never need to grow during evaluation and thus reuses the same area of shared memory on each iteration. Also, while will always materialize the input array to shared memory before starting the iteration. To avoid doing a direct copy from global memory to shared memory, in the reduction kernel, we take one initial step before starting the while-loop. This optimization is called “First add during load” by Mark Harris [11].

To get this to run over multiple blocks, we need to split a larger array and concatenate the results:

```haskell
let chunkSize = 2 * #BlockSize
in splitUp chunkSize arr
|> map (red <block> f)
|> concat 1
```

Here #BlockSize will refer to either CUDA's blockDim.x, OpenCL's get_local_size(0), or a constant specified by the user as configuration option at compilation time.

Another difference from Obsidian also comes to light here: as we no longer distinguish between statically known values and dynamically known values, we are not be able to infer that red <block> f always returns a single scalar. We solve this by requiring an extra argument to concat, an expression computing the size of each chunk to concatenate.

4. Performance

FCL is work in progress; thus certain optimizations are still not implemented. However, the performance on the previously shown examples is promising, and we have identified the bottlenecks that are currently limiting performance. We compare the performance of each benchmark with hand-written OpenCL kernels from NVIDIA's OpenCL SDK.

When an FCL program is compiled, the result is a file containing one or more OpenCL or CUDA kernels. In the future, we also want to be able to generate host-code, but right now it must be written by hand. We use the same host-code for both FCL-generated kernels and the handwritten kernels by NVIDIA.

To benchmark the generated code, we have used an NVIDIA GeForce GTX 780 Ti, which is built on the Kepler architecture. It has 2880 cores (875 Mhz), and 3GB GDDR5 ram (7 Ghz, bus-width: 768 bit = 330GB/s). In practice we can expect a 254.90GB/s maximum bandwidth, which we have measured using NVIDIA's benchmarking tool (bandwidthTest).

Each benchmark has been executed on an array of $2^{24}$ 32-bit integers (67 MB). Timing was measured as wall-clock time on 1000 executions of the same kernel, preceded by a single warm-up run. The measured bandwidths are shown in Figure 2. The theoretical maximum bandwidth is plotted as a dashed horizontal line.

In the simple reverse example, we hit the measured maximum bandwidth as we hoped. The generated code is similar to the handwritten code from NVIDIA, except for block-virtualization, which is not used in NVIDIA's version.

In the transpose example we are not quite on par with the handwritten code, and there are two reasons for that. First, we do not take care to avoid bank-conflicts, which we leave as future work. Second, we have quite a lot of extraneous divisions in the generated code. This is because we do not keep track of array shapes, and thus split2DGrid and concat2DGrid are performing some of the same work more than once. If we remove these double computations by hand, we achieve a performance boost, which is illustrated as FCL+handopt in the barplot. We are planning to add support for multi-dimensional arrays to tackle this issue, but this is also left as future work.

The reduction example is interesting: here we generate a completely unrolled loop, which performs reasonably well, but does not quite hit the performance target set by NVIDIA's heavily tuned kernel. To identify how we can improve our solution, we have inspected the difference between the two kernels. To get on par with NVIDIA's kernel we will need to make each thread do an initial sequential reduction on a few elements, before the parallel tree-reduction we already have implemented.

5. Type System and Semantics

To better understand the limitations and performance of programs written in FCL, and to validate correctness, we will now turn to a more formal treatment of the language.

Previously, we have described both of the functions concat and interleave, which are used for distributing a computation. Both functions are written in terms of a more general operation, which we have named interleave. The interleave operation is in essence a forward permutation on the indexes written to. However, in the limited treatment in this paper, we will focus on a simplified version of FCL with concat as a primitive, leaving out concat2DGrid and interleave. In all other aspects, this is a full treatment of FCL in its current state.

We use $i, d,$ and $b$ to range over integers, doubles, and booleans, respectively. Let $\alpha$ range over an infinite set of type-variables and let $x$ range over program variables. We use unaryop and binaryop to denote the sets of built in scalar operations.

Whenever $z$ is some object, we write $z'$ to range over sequences of similar objects. When we want to be explicit about the size of a sequence $z = z_0, \ldots, z_{n-1}$, we often write it on the form $z^{(n)}$. 

![Figure 2: Measured bandwidths on our three example programs. OpenCL bars are code from NVIDIA's OpenCL SDK, and we compare it to OpenCL kernels generated by FCL. The dashed line indicates the maximum bandwidth as measured by NVIDIA's benchmarking tool.](image)
lengthPull: [α] → int
lengthPush: [α](lvl) → int
mapPull: (α → β) → [α] → [β]
mapPush: (α → β) → [α](lvl) → [β](lvl)
generate: int → (int → α) → [α]
index: [α] → int → α
push: (lvl) → [α] → [α](lvl)
force: [α](lvl) → [α]
while: ([α] → bool) → ([α] → [α](lvl)) → [α](lvl) → [α]

Figure 3: Types of built-in operators.

5.1 Type System

The syntax of FCL types, kinds and type-schemes is defined as follows:

| op := unaryop | binaryop (built-in operators) |
| generate | lengthPull | lengthPush |
| index | mapPush | mapPull |
| push | force | concat |

bt ::= α | int | double | bool (base types)
τ ::= α | β | (τ₁, τ₂) | τ₁ → τ₂ (types)
| (α) → τ
| [τ]
| [bt](γ)
κ ::= BT | GT | TYP | LVL (kinds)
σ ::= forall α : κ. σ → τ (type-schemes)

The types of built-in array combinators are shown in Figure 3.

To define the set of valid types (under assumptions for free variables), we define a relation Δ ⊢ τ below, where Δ are kind environments, mapping type variables to kinds:

Δ ::= α : κ. Δ | ϵ

The kind-system divides types into four categories. Base types (BT), ground types (GT), general types (TYP) and levels (LVL). Base types are types of scalar values, which are the only types of values allowed in push arrays. Ground types are all types except function-types, and are the types allowed in pull arrays.

5.2 Dynamic Semantics

We now present the semantics of the language, which will aid understand how FCL terms can be compiled and, in particular, how level-types guide the compilation.

The evaluation relation, we define below, is annotated with a location, Locations emulate the hierarchical structure of a parallel machine, and are of the form:

loc ::= Thread(thread_id) | Group({loc₁, . . . , loc_n})

Locations relates to levels and we introduce similar shorthands for wraps, blocks and grids:

Warp(loc) = Group({Thread(loc₁), . . . , Thread(loc_n)})
Block(loc) = Group({Warp(loc₁), . . . , Warp(loc_n)})
Grid(loc) = Group({Block(loc₁), . . . , Block(loc_n)})

We also introduce a relation, loc ⊳ γ, which defines whether the location loc is respecting the level γ:

Thread(thread_id) ⊳ γ (31) loc_i ⊳ γ for all i (32)
Expression typing

\[ \Delta, \Gamma \vdash i : \text{int} \] (17)
\[ \Delta, \Gamma \vdash d : \text{double} \] (18)
\[ \Delta, \Gamma \vdash b : \text{bool} \] (19)
\[ \Delta, \Gamma \vdash e_i : \tau, \text{ for all } i \] \[ \Delta \vdash \tau : \text{GT} \] (20)
\[ \Delta, \Gamma \vdash [e_1, \ldots, e_n] : [\tau] \] (21)
\[ \Delta, \Gamma \vdash e_1 : \tau_1 \] \[ \Delta, \Gamma \vdash e_2 : \tau_2 \] (22)
\[ \Delta, \Gamma \vdash \text{fst } e : \tau_1 \] (23)
\[ \Delta, \Gamma \vdash \text{snd } e : \tau_2 \] (24)
\[ \Gamma(x) = \alpha \] \[ \Delta, \Gamma \vdash \tau : \text{TYP} \] (25)
\[ \Delta, \Gamma \vdash e_1 : \tau \] \[ e = \text{fv}(\tau) \] \[ \Delta, (\Gamma, x : \forall \alpha \tau, \tau) \vdash e_2 : \tau \] (26)
\[ \Delta, \Gamma \vdash e_1 : (\alpha) \rightarrow \tau \] \[ \Delta, \Gamma \vdash \alpha : \text{LVL} \] (27)
\[ \Delta, \Gamma \vdash \alpha(\gamma) : [\tau] \] (28)
\[ \Delta, \Gamma \vdash \text{fn } x \Rightarrow e : \tau' \rightarrow \tau \] (29)
\[ \Delta, \Gamma \vdash e_1 : (\alpha) \rightarrow \tau \] \[ \Delta, \Gamma \vdash \text{fn } (\alpha) \Rightarrow e : (\alpha) \rightarrow \tau \] (30)
\[ \Delta, \Gamma \vdash e_1 : \text{int} \] \[ \Delta, \Gamma \vdash \text{concat } e_1 e_2 : [\alpha(1+\gamma)] \] (31)
\[ \Delta, \Gamma \vdash \text{concat } e_1 e_2 : [\alpha(1+\gamma)] \] (32)

Values in FCL are either base values \((bv)\), pull arrays, push arrays, or delayed concatenation of push arrays.

\[
v ::= \begin{cases} b & \text{(base values)} \\ [e_1, \ldots, e_n] & \text{(pull array)} \\ [e_1, \ldots, e_n](\gamma) & \text{(push array)} \\ \text{concatDelay } e_1 e_2 & \text{(delayed concat)} \end{cases}
\]

We extend the typing relation above to include typing of values.

Value typing

\[ \Delta, \Gamma \vdash v : \tau \] (33)
\[ \Delta, \Gamma \vdash e : \tau \] \[ \Delta, \Gamma \vdash \gamma : \text{GT} \] (34)
\[ \Delta, \Gamma \vdash [e_1, \ldots, e_n] : [\tau] \] (35)
\[ \Delta, \Gamma \vdash e_1 : \tau \] \[ \Delta, \Gamma \vdash \gamma : \text{BT} \] (36)
\[ \Delta, \Gamma \vdash [e_1, \ldots, e_n](\gamma) : [\tau](\gamma) \] (37)
\[ \Delta, \Gamma \vdash \text{concatDelay } e_1 e_2 : [\tau](1+\gamma) \] (38)

We now define the promised dynamic semantics of FCL. Due to space limitations, we consider just the interesting cases involving force. The first two rules are administrative fusion rules, which an implementation can choose to implement at compile time (for space reasons, we show only a subset of the administrative rules here).

Only programs of type \([\alpha](\gamma)\) can be fully evaluated under this semantics. For instance, we require that a reduction-kernel returns a singleton push array instead of an integer. This requirement is intentional; all programs must be explicit about the computation hierarchy and, currently, only push arrays allows for an annotation that specifies the hierarchy of the computation.

**Proposition 1 (Type Preservation).** If \[ \Delta, \Gamma \vdash e : \tau \] and \[ e \vdash_{\text{loc}} e' \] for some location \(loc \supsete \gamma\), then \[ \Delta, \Gamma \vdash e' : \tau \].

**Proposition 2 (Progress).** If \[ \Delta, \Gamma \vdash e : \tau \] for some location \(loc \supsete \gamma\), then either \(e\) is a value or \(e \vdash_{\text{loc}} e' \) for some \(e'\).

6. **Related Work**

FCL builds on previous work on Obsidian [19], from which both the concepts of push arrays and level-variables originate. FCL distinguishes itself from Obsidian, by adding support for more involved interleaving patterns, being a self-contained language and not allowing meta-programming.

Obsidian and FCL are not the first languages for hierarchical parallel machines. Sequoia is a imperative hierarchical language [10], inspired by previous work on Parallel Memory Hierarchies (PMH) [2], supporting both cluster computing through MPI and programming multiple GPUs. Both Sequoia and PMH builds a parallel machine as a tree of distinct memory modules. Programs are written to be machine independent, where function calls corresponds to either executing a subtask on a child in the hierarchy (copying data to this memory module) or staying in the same memory module. Thus, the call/return of a subtask implies that data movement through the machine hierarchy might occur. The stopping criteria for recursive functions are left out, and instead specified separately in a mapping specification, that details how an algorithm maps to a concrete machine. Programs can also involve tunable parameters and various variants of the same algorithm; the mapping specification also controls these choices. Mapping specifications can potentially be automatically generated. The ideas from Sequoia are further generalized in the work on Hierarchical Place Trees [20].

Another hierarchical data-parallel language is HiDP by Zhang and Mueller [14] for hierarchical GPU-programming. In addition to the hierarchies of Obsidian and FCL, they add two sub-warp levels of size 4 and 8, respectively. Parallelism is embodied as nested
parallel-for loops (which are called map-blocks) together with a set of built-in parallel array-operators (partition, reduce, scan, sort, reverse). Arrays are multi-dimensional, and nested irregular segmented arrays are built-in. For optimization purposes, it is however also possible to use regular arrays. Fusion decisions and use of shared-memory are completely controlled by the compiler.

The language discussed by Dubach et al. [18] is also related to FCL, operating at a similarly low-level. The main idea is to build a language that can be automatically tuned to hardware, by applying search strategies on the provided set of rewrite rules. It might be interesting to build a similar search-based rewrite-engine on top of FCL, and allow the user to express rewrite-rules. Another interesting aspect of this work is its support for programming with vector-instructions (such as adding two int4 in OpenCL), which would correspond to a layer between warp-level and thread-level in Obsidian and FCL.

The hierarchy in FCL and Obsidian might also be compared to the concept of locales and sublocales in the Chapel language [7].

Functional approaches to GPU computing have typically concentrated on optimizing compilers that are intended to shield the user from the need to understand (or control) details of the GPU. Examples include Futhark [12], Accelerate [6], Delite [5], Harlan [13], and Nessie [3]. These projects might perhaps be considered to be at roughly the same level as NVIDIA’s Thrust library [16]. FCL and Obsidian are rather at the level of NVIDIA’s CUB library, which provides reusable software components for every level of the GPU hierarchy [17].

7. Conclusion and Future Work

We have presented FCL, a functional language for GPU algorithms. FCL is work in progress. Currently only device-code is generated, and host-code has to be written manually. In addition, memory is currently allocated implicitly, and it is thus not possible to reuse the same memory. We would want the possibility of writing an in-place version of reverse, writing the reserved array back to the same global-array.

Our limitation of only having one-dimensional arrays will in many cases lead to unnecessary shape-computations, as we saw in the transpose example. We will thus investigate how shapes can be introduced, such that split2DGrid, would split the array into a 2D array of 2D arrays.

Future work also includes bank-conflict avoidance, use of vectorized GPU-instructions, and the addition of sequential loops with array updates, perhaps in the style of Futhark [12].

Finally, we would like to implement some larger example programs in FCL, and attempt to use FCL as an intermediate language for our APL-compiler [9].

References


