FCL: A Functional Domain-specific Embedded Language for GPU Programming

Allen Hsu, Kevin A. Wortman

Abstract. We propose a Functional Programming language for utilizing the Graphics Processing Unit for General Purpose Computation. General Purpose Programming on the GPU (GPGPU) has been a long sought after endeavor; the number of threads available on a GPU vastly outnumber the number of threads on a CPU. However due to the difficulties of programming for the GPU, this potential has been largely untapped. FCL is our attempt at making the programming process much simpler, both syntactically and algorithmically. Because of the inherent statelessness of parallelization, we believe a functional language is the most appropriate. Functional Programming is inherently stateless; where an algorithm written for a sequential environment would be exactly the same as one written for an parallel environment.

1 *Introduction*

The future of computing is parallel processing. As clock speeds will eventually max out and the only way to improve speeds is with a larger quantity of threads. Most applications in the software industry are not optimized to use parallel processing. Applications that do take advantage of parallel processing can be optimized further by processing their computation on a GPU rather than a CPU.

The main obstacle impeding programmers from attempting general purpose GPU computing is the difficulty of implementation. All GPGPU APIs at the moment are written in low level languages such as C, C++, and Fortran; the most notable ones being CUDA and OpenCL. The difficulty, or more accurately tediousness, of the steps required to use these APIs can be avoided. Wrapping these APIs in high level languages such as Haskell or Python have been attempted before. However, the problem with theses attempts is that they still follow the same workflow as the original low level APIs. They are simply extensions meant to allow access to CUDA or OpenCL, rather than offering true higher level abstraction.

Our answer to this problem is FCL. FCL is a domain-specific embedded language in Chicken Scheme. FCL allows one to write in a high level functional language, based on Scheme, that compiles into OpenCL kernel code. Next, FCL offers powerful control procedures for applying these 'kernels' onto CPU memory. We believe that the low level operations of transferring data from buffer to buffer, device to device, or vice versa should be abstracted away from the programmer. That will allow the programmer more freedom to solve bigger and more dynamic problems.

2 Overview

2.1 Domain-specific language

A domain-specific language, commonly referred to as a DSL, is a language designed for a specific purpose in a specific domain. Embedded variants of DSLs are extensions of the parent language and are implemented as a library. The purpose of FCL is to make creating a GPU accelerated program to be as simple as creating a normal single threaded CPU program and the domain of the language is Chicken Scheme.

2.2 Functional Paradigm

To unify non-parallel and parallel code, we chose the Functional Paradigm. Functional code is written in a way that is stateless and not dependent on execution order. If a program were written in a functional manner parallelization would essentially be painless. In contrast, when programs are written in a non-functional way, they usually are inherently full of state and rely on sequential execution. When an program is dependent on variable states and execution order, it is not naturally parallelizable. Having stateless code allows a program to be made parallel without modification, which means parallel and non-parallel code would look exactly the same.

2.3 Chicken Scheme

Chicken is a compiler for the Scheme programming language; Chicken produces very portable and efficient C. What is unique about Chicken is that it has a very powerful foreign function interface for embedding C code. It is for this reason we chose Chicken over other implementations of Scheme. The foreign function interface allows us to include the OpenCL library directly into our FCL library and call its functions.

2.4 OpenCL

The back end to FCL is OpenCL. In fact, the name FCL is a reference OpenCL. The ‘CL’ comes from OpenCL and stands for ‘Computing Language’ and the ‘F’ stands for ‘Functional’, in total ‘Functional Computing Language’. We chose OpenCL because of its portability and Open Source nature. OpenCL is an API framework for creating programs that utilize parallelization on either a CPU or GPU. It offers a C99 based language for writing kernels and a robust C language API for transferring data from a host program to a parallelization platform. In OpenCL there is a device scope and an host scope. An OpenCL program can be briefly described as a program that takes a kernel defined in the device scope and maps it to data buffers allocated on the host scope. This mapping is done in parallel, where the kernel is run N times (N represents number of items in the buffer). After the kernel has been applied to the buffer, the result is then transferred back to the host scope and can be then used in other operations.

2.5 FCL Scope

FCL, like OpenCL, has two scopes; the device scope and the host scope. The device scope resides on the parallel processing unit, i.e. the GPU. The host scope resides on the CPU, in the same scope as all normal scheme code. FCL keywords for defining things in the device scope are indicated with a 'g' as its first letter, e.g. gdefine, glambda, etc. The 'g' scope is compiled by the FCL compiler into OpenCL kernel source code and then loaded onto an OpenCL kernel program. The kernel program can then be invoked using host scope command procedures, i.e. pmap, pfilter, and 'g' data types, i.e. gdata, gbuffer. The FCL kernel compilation does not disrupt the compile process of Chicken Scheme because the kernel compilation occurs at run time. Essentially there are 2 compilations and 2 runs. First there is the host code compilation and run time. Then there is the device code compilation and run time that occurs within the host's run time.

2.6 FCL Compilation

FCL is a true embedded language. Meaning it is included like any other library in Scheme and the program is only manually compiled once by the user.

2.7 FCL Typing

FCL is a strongly typed language. All variables upon initialization must have a type specification.

3 *Description*

3.1 Fundamental Types

Base:

```
bool, char, short, int, float, double, record
```

Unsigned:

```
uchar, ushort, uint
```

List

```
bool*, char*, short*, int*, float*, double*, record*
```

3.2 Functions

Standard Definition

```
(gdefine add (glambda ((int x) (int y) -> int)
               (g+ x y)))

(gdefine sin-1 (glambda (int)
                 (gsin 1)))
```

Syntactic Sugar Definition

```
(gdefine (add (int x) (int y) -> int)
  (g+ x y))

(gdefine (sin-1 -> int)
  (gsin 1))
```

Syntactic Sugar Definition, marked for early compilation

```
(gdefine* (add (int x) (int y) -> int)
  (g+ x y))

(gdefine* (sin-1 -> int)
  (gsin 1))
```

3.3 Conditional Statements

if

```
(gdefine (minimum (int x) (int y) -> int)
  (gif (g> x y)
       x
       y))

(minimum 4 3)  ; returns 3
(minimum 2 10) ; returns 2
```

cond

```
(gdefine (clamp (int x) (int min) (int max) -> int)
  (gcond ((g< min x) min)
         ((g> max x) max)
         (else x)))

(clamp 4 1 10) ; returns 4
(clamp 3 5 10) ; returns 5
```

case

```
(gdefine (grade (int x) -> char)
  (gcase x
    ((4) "A")
    ((3) "B")
    ((2) "C")
    ((1) "D")
    (else "F")))

(gdefine (tier (char g) -> int)
  (gcase g
    (("A" "B") 1)
    (("C" "D") 2)
    (("F") 3)
    (else 4)))

(grade 4)        ; returns "A"
(grade 2)        ; returns "C"
(grade 0)        ; returns "F"
(grade -1)       ; returns "F"
(tier (grade 2)) ; returns 2
(tier (grade 4)) ; returns 1
(tier 10)        ; returns 4
```

3.4 Lists

Standard Initialization

```
(gdefine lst (glist int 1 2 3 4 5 6 7 8 9 10))
```

Conversion Initialization

```
(gdefine lst (list->glist int (list 1 2 3 4 5 6 7 8 9 10)))
```

3.5 Higher-order Expressions

Map

```
(gdefine (square (int x) -> int)
  (* x x))

(define result (gmap square (glist int 1 2 3 4 5)))

;; result returns (glist int 1 4 9 16 25)
```

Fold

```
(gdefine (find-largest (int* lst) -> int)
  (gfold (glambda ((int curr) (int prev) -> int)
           (gif (g> curr prev)
                curr
                prev))
         0
         lst))

(find-largest (glist int 1 2 3 4 5 6 7 8 9)) ; returns 9
```

3.6 Embedding Scheme

Scheme Procedures (behave like macros in FCL 'g' namespace)

```
(define (formula a b)
  (g/ (g+ a b) 2))

(gdefine (foo1 (float x) (float y) -> float)
  (formula x y))

;; equivalent to foo1
(gdefine (foo2 (float x) (float y) -> float)
  (g/ (g+ x y) 2))

(define (sum-list a)
  (fold + 0 a))

(gdefine (foo3 (float x) -> float)
  (g+ (sum-list (list 1 2 3 4 5)) x))

;; equivalent to foo3
(gdefine (foo4 (float x) -> float)
  (g+ 15 x))
```

3.7 Objects

Record Types

```
(gdefine-record-type <vec>
  (vec (float x) (float y) (float z))
  (x vec-x)
  (y vec-y)
  (z vec-z))

(gdefine-record-type <triangle>
  (triangle (vec v0) (vec v1) (vec v2))
  (v0 triangle-v0)
  (v1 triangle-v1)
  (v2 triangle-v2))

(gdefine-record-type <sphere>
  (sphere (vec pos) (float radius))
  (pos sphere-pos)
  (radius sphere-radius))

(gdefine vec1 (vec 1.0 3.2 1.0))
(gdefine vec2 (vec 4.0 2.1 3.5))
(gdefine vec3 (vec 9.9 2.4 3.0))
(gdefine a-sphere (sphere vec1 5.0))
(gdefine a-triangle (triangle vec1 vec2 vec3))

(sphere-pos a-sphere)    ; accesses pos property of a-sphere
(triangle-v0 a-triangle) ; accesses v0 property of a-triangle
```

3.8 Polymorphism

Compile Time Selection

```
(gdefine (gmember-int? (int value) (int* lst) -> bool)
  (gfold (glambda ((int curr) (int prev) -> int)
           (gif (g= curr value)
                gtrue
                prev))
         gfalse
         lst))

(gdefine (gmember-float? (float value) (float* lst) -> bool)
  (gfold (glambda ((float curr) (float prev) -> float)
           (gif (g= curr value)
                gtrue
                prev))
         gfalse
         lst))

(gdefine (gmember-double? (double value) (double* lst) -> bool)
  (gfold (glambda ((double curr) (double prev) -> double)
           (gif (g= curr value)
                gtrue
                prev))
         gfalse
         lst))
```

```
(define gmember-dictionary
  (list (cons "int" gmember-int?)
        (cons "float" gmember-float?)
        (cons "double" gmember-double?)))

(define (gmember-poly type)
  (cdr (assoc type gmember-dictionary)))

(define (gmember? value lst)
  ((gmember-poly (gtype lst)) value lst))
```

Run Time Selection

```
(gdefine (intersect? (record obj) (ray r) -> bool)
  (gcond ((sphere? obj) (sphere-intersect? obj r))
         ((triangle? obj) (triangle-intersect? obj r))
         (else gfalse)))
```

3.9 Intermediate Objects

Lists

```
(define buffer (list->gbuffer int (iota 1000)))

;; usable by FCL parallel higher order functions
(define new-buffer (pmap gfoo buffer))

;; set-gbuffer! reuses buffer without allocating a new one
(set-gbuffer! buffer (pmap gfoo buffer))

;; accessible within gprocedures
(gdefine (can-use-buffer (int x) -> int)
  (g+ x (gfold + 0 buffer)))
```

Atoms

```
(define data (gdata int 100))
(set-gdata-values! data 50)

;; accessible within gprocedures
(gdefine (can-use-data (int x) -> int)
  (g+ x data))
```

3.10 Parallel Higher-Order Expressions

Map

```
(gdefine* (double (int x) -> int)
  (g+ x x))

(pmap double (list->gbuffer int (list 1 2 3 4 5)))
;; returns a gbuffer containing (2 4 6 8 10)
```

Filter and Remove

```
(gdefine* (even? (int x) -> bool)
  (g= (gmodulo x 2) 0))

(define buffer (list->gbuffer int (list 1 2 3 4 5)))

(pfilter even? buffer) ; returns gbuffer with (2 4)
(premove even? buffer) ; returns gbuffer with (1 3 5)
```

3.11 FCL Procedures

FCL Context

```
(fcl-begin) ; allocates and initializes a FCL context on the GPU device

#| your code would be here |#

(fcl-end)   ; releases FCL context and any unfreed intermediate objects
```

FCL Expressions

```
(define-fcl-exp (run-gpu-code n)
  (define buffer (list->gbuffer int (iota n)))

  (gdefine* (double (int x) -> int)
    (g+ x x))

  (gbuffer->list (pmap double buffer)))

;; FCL expressions ensure that all gprocedures, gbuffers and gdata defined
;; within it are released when it goes out of scope.
```

3.12 Simple Example

First we must allocate and initialize a FCL context.

```
(fcl-begin)
```

Next lets create a simple gprocedure that adds two integers.

```
(gdefine* (gadd-nums (int x) (int y) -> int)
  (g+ x y))
```

Notice we use gdefine* for gadd-nums instead of the normal gdefine. This will mark this gprocedure to be compiled and loaded onto the gpu as soon as it is defined; otherwise compilation would occur when the gprocedure is used in host scope or explicitly included using fcl-include. An equivalent procedure to gadd-nums in the host scope would be

```
(define (add-nums x y)
  (+ x y))
```

One way to use a gprocedure is to call it within another gprocedure.

```
(gdefine (gsub-sums (int x) (int y) -> int)
  (g- (gadd-nums x y) (gadd-nums x y)))
```

Another way to use a gprocedure is to apply it to data allocated in the host scope. This is done using gbuffers and control procedures like pmap. Pmap is the parallel equivalent to map.

```
(define a (list 1 2 3 4 5 6 7 8 9 10))
(define b (list 1 1 1 1 1 1 1 1 1 1))
(define a-buffer (list->gbuffer int a))
(define b-buffer (list->gbuffer int b))

(define c-buffer (pmap gadd-nums a-buffer b-buffer))
(define c (gbuffer->list c-buffer))
(define d (map add-nums a b))
```

The lists c and d would be equivalent each containing ‘(2 3 4 5 6 7 8 9 10 11). Finally we release the FCL context.

```
(fcl-end)
```

The procedure fcl-end also releases all gprocedures, gdata, and gbuffers that were not previously released. Releasing ‘g’ objects with fcl-end is acceptable for simple cases. However, normally we would want to make code reusable by wrapping it in a procedure. A normal scheme procedure would not release ‘g’ objects when it goes out of scope. To make a reusable procedure that allocates and releases ‘g’ objects we would need to wrap all of the above, excluding fcl-begin and fcl-end, in a FCL expression. This also allows us to reuse the same FCL context.

```
(define-fcl-exp (simple-example a b)
  (gdefine* (gadd-nums (int x) (int y) -> int)
    (g+ x y))
  (define a-buffer (list->gbuffer int a))
  (define b-buffer (list->gbuffer int b))
  (define c-buffer (pmap gadd-nums a-buffer b-buffer))

  (gbuffer->list c-buffer))
```

All ‘g’ objects created in a FCL expression are automatically released when the FCL expression goes out of scope.

4 *Experimental results*

4.1 Floyd-Warshall, All Pairs Shortest Path

Chicken Scheme (with FCL) implementation of Floyd-Warshall All Pairs Shortest Path algorithm

```
  (define-fcl-exp (gpu-floyd-warshall n graph)

    (define indices (list->gbuffer int (iota (length graph))))

    (define adj-graph (list->gbuffer int graph))

    (define k (gdata int 0))

    (define (graph-ref g x y)
      (glist-ref g (g+ y (g* x n))))

    (gdefine* (shortest-path (int idx) -> int)
      (glet* ((j (gmodulo idx n))
              (i (g/ (g- idx j) n))
              (a (graph-ref adj-graph i k))
              (b (graph-ref adj-graph k j))
              (c (graph-ref adj-graph i j)))
        (gmin (g+ a b) c)))

    (for-each (lambda (k-index)
                (set-gdata-value! k k-index)
                (set-gbuffer! adj-graph (pmap shortest-path indices)))
              (iota n))

    (gbuffer->list adj-graph))
```

Chicken Scheme (without FCL) implementation of Floyd-Warshall All Pairs Shortest Path algorithm

```
  (define (cpu-floyd-warshall n graph)
(define indices (iota (length graph)))

(define adj-graph graph)

(define k 0)

(define (graph-ref g x y)
  (list-ref g (+ y (* x n))))

(define (shortest-path idx)
  (let* ((j (modulo idx n))
         (i (/ (- idx j) n))
         (a (graph-ref adj-graph i k))
         (b (graph-ref adj-graph k j))
         (c (graph-ref adj-graph i j)))
    (min (+ a b) c)))

(for-each (lambda (k-index)
            (set! k k-index)
            (set! adj-graph (map shortest-path indices)))
          (iota n))

adj-graph)
```

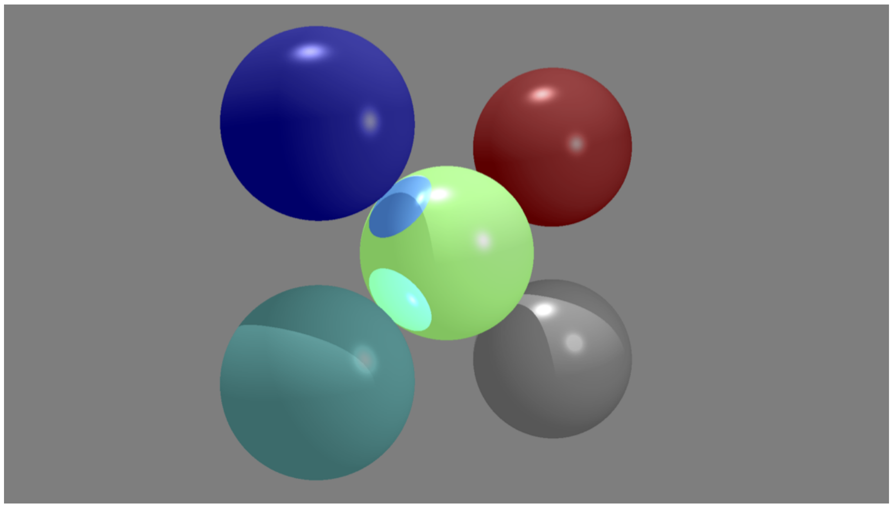
The two implementations are algorithmically identical and syntactically they are almost identical. The main difference is FCL version has to take extra steps of converting lists to gbuffers.

4.1.1 Run Time

In contrast to the syntax, run times between the implementations are drastically different. The GPU implementation has a time complexity of O(N + M), where N is the number of nodes and M is the work needed to transfer data from the host to the device. The CPU implementation has a time complexity of O(N^3), where the N also equals the number of nodes.

Nodes	GPU (in sec)	CPU (in sec)
40	0.038974	0.64531
80	0.044431	10.951671
120	0.063592	170.744784

4.2 Phong Illumination Raytracing



4.2.1 Run Time

5 *Limitations*

Currently there are a few limitations to FCL. They are connected to the limitations that plague many other GPGPU platforms. Most notably is that gbuffers and glists cannot be dynamically sized.
- glists and gbuffers are homogenous and one dimensional - glists and gbuffers cannot be dynammically sized - filter not possible because of lack of dynamically sized lists - procedures cannot be used as input or output types - lists cannot be output types - parallelization is only work it if alg is O(n squared) and above or n is extrememly large

6 *Future work*

7 *Conclusion*

8 *Related work*

Here is an example of a citation [1].

9 ***Bibliography***

References

[1] M. Patrascu and M. Thorup. The power of simple tabulation hashing. *arXiv preprint arXiv:1011.5200*, 2010.