# Language Proposal: Sheets

*What's your thread count?*

**September 24th 2014**

**COMS W4115 - Programming Languages and Translators**

**Benjamin Barg** `bbb2123` - The Heart (Language Guru)

**Gabriel Blanco** `gab2135` - The Muscle (Testing)

**Ruchir Khaitan** `rk2660` - The Brains (Systems Architect)

**Amelia Brunner** `arb2196` - The Boss (Manager)

## Motivation

In recent years, parallel computation hase become a powerful tool for improving performance of desktop applications. While parallelism is not a novel concept—nondeterministic finite automata existed long before digital computers—it has become much more relevant to personal computer users in the past decade because of increasing hardware support for various types of concurrent program execution.

Parallel computation may be broken down into two rough categories: task-based and data-based parallelism. Task-based parallelism is characterized by a relatively small number of threads and a relatively high complexity of such threads, while data-based parallelism requires a relatively large number of threads performing similar simple computations. Task-based parallelism is particularly suited to CPU architectures, which are optimized for low-latency computation. Many languages and APIs serve the task-based domain at fairly high levels of abstraction above the underlying hardware; examples include POSIX threads, Golang, and OpenML.

Conversely, GPUs are intrinsically better than CPUs at performing relatively simple operations across large data sets in parallel, i.e. data parallelism. By focusing on high throughput, high latency operations, and including hardware support for thousands of threads and efficient thread context switching, GPUs are able to perform massive amounts of computation across independent chunks of data much faster than could CPUs, and they scale much better than CPUs as data size grows. The GPU multiprocessing paradigm is thus both fundamentally different than that of CPUs, and is more applicable to "embarassingly parallel" problems.

However, GPU programming is currently an extremely low-level exercise. Two main languages exist for GPU programming: CUDA (Nvidia's propreitary GPU API) and OpenCL (an open-source, cross-platform language for heterogenous hardware programming). We have chosen not to use CUDA because it is not open-source, and it lacks extensibility to other hardware. Conversely, OpenCL offers support for a wide variety of GPUs, and conforms to our open-source philosophy.

Our primary target with Sheets is the programmer who wants to take advantage of the performance benefits of mass parallelism (specifically the large performance benefits of GPU execution for programs operating on large data sets) but who is unfamiliar with hardware specifics and the details of concurrent programming and does not need all the optimization features provided by OpenCL. We believe that there is a rich class of problems that, while not approaching the scale of massively parallel computation frameworks, would still enjoy the performance gains offered by parallel computation on a desktop GPU.

## Summary of Goals

- a high-level language for manipulation of large data sets that takes advantage of GPU parallelism
- provide simple abstractions of common classes of massively parallelized GPU computation
- compile into OpenCL to support multiple GPU architectures

## Domain Features

*You don't need to know what's going on underneath the Sheets*

- single and multi-dimensional array primitives
- `map` and `filter` operations implemented in underlying OpenCL library
- GPU-backed support for common matrix and vector operations
- `import` and `write` primitives for reading and writing large data files, supporting multiple binary encodings

## Language Design

Sheets makes parallel computation on a GPU more accesible by abstracting away the features already found in OpenCL. The ideal user of our language is the programmer who wants to do simple operations on large data sets and therefore wants the benefits of GPU processing, but who doesn't want to worry about the extra level of complexity involved with GPU threading. Since familiarity and ease of use is a priority in our project goals, we are modelling the aesthetic of our language on both Python and C. Python's use of white space for dilineation makes it easier to read and write code; however, we are choosing not to implement Python's inferred data types because this would introduce unnecessary ambiguity when marshalling arguments for GPU computation. For the primitives, we are using a lot of the same data types as in C, and since OpenCL is based on C99, this will make porting our language down easier. With our syntax focused on ease of use, we've also added a few language features to aid with parallelizing large-scale array operations, which are outlined in more detail below.

## Code Samples

### Primitive Declarations

```
int
long
float
double
char

int[]           // Int Arrays
long[]          // Long Arrays
float[]         // Float Arrays
double[]        // Float Arrays
char[]          // Char Arrays (Strings)
bit[]           // Bit Arrays, useful for bit masks
```

### Sample operations to be automatically parallelized

```
/* all mathematical operations applied to entire arrays
 * can be thought of as 'embarassingly parallelizable,'
 * so in Sheets, we parallelize them automatically:
 */

/* Math operations */
array3 = array1 * array2       // multiplication
array3 = array1 / array2       // division
array3 = array1 + array2       // addition
array3 = array1 - array2       // subtraction
array3 = array1 ^ array2       // power
array3 = -array1               // negation
array3 = array1++              // increment all values
array3 = array1--              // decrement all values

/* Array-specific operations */
array3 = array1                // copying
array3 = 'array1               // reversing
array3 = array1 ** array2      // matrix multiplication

/* Bit-level operations */
array3 = array1 AND array2     // 'and' operator
array3 = array1 OR  array2     // 'or' operator
array3 = array1 XOR array2     // 'xor' operator
array3 = array1 NOR array2     // 'nor' operator

/* Bitshifting to be applied to all values in array */
array3 = array1 << 1           // left shift
array3 = array1 >> 1           // right shift
```

## Use Case: Audio Mixing

```
/*
 * When mixing two audio tracks in float32 format, you have to
 * scale so that the values remain between (-1, 1). Simple addition
 * would lead to clipping audio.
 *
 * SHEETS automatically parallelizes the scalar multiplication operation
 * as well as the array addition using the GPU.
 */
void main:

    float[] audio1 = import('file1.wav', 'float32')
    float[] audio2 = import('file2.wav', 'float32')

    audio1 = audio1 * .5
    audio2 = audio2 * .5

    float[] mixed_audio = audio1 + audio2

    write('file3.wav', mixed_audio)
```