C++ Mapreduce

ISO/IEC JTC1 SC22 WG21 N3446=12-0136 - 2012-09-21
Chris Mysen, mysen@google.com, ccmysen@gmail.com
Lawrence Crowl, crowl@google.com, Lawrence@Crowl.org
Adam Berkan, aberkan@google.com

Introduction

The difficulties in writing race-free and efficient threaded and distributed code often lies in the difficulty in coordinating access to shared data in an efficient manner, particularly when different logic is attempting to access this data across a number of threads. There are several classes of problems, though, which can be mapped onto a simple map-reduce pipeline which allow for efficient multithreaded access and very simple program creation. This is done via a simple execution model with map and reduce stages, as well as reduction of the coordination problem by staging the execution of work.

This mechanism has been put to great use in data-heavy distributed processing problems, but has also found good use in the implementation of many threaded algorithms which can be implemented in district stages representing data-parallel operations. Examples of effective uses of map-reduce for this purpose include image processing pipelines, iterative machine learning tasks, matrix manipulation, and others. *** Add a couple of references here.

Overview

For large scale distributed problems, the map-reduce framework has proven to be a highly effective way at creating highly parallel workflows working over distributed filesystems on petabyte scale operations and has been used for analysis, machine learning, and

implementation of many distributed computation problems at Google. It is widely popular with several implementations, including some open source because of the simplicity of the programming model and the abstraction provided by a simple flow of operations:

- A set of mappers each operating on a single input value and emitting one (or more) key+value pairs (logically acting like a key->value multimap)
- A set of reducers which receive a key and list of values from the mappers and reducing the set values to a key+output pair
- A map-reducer controller which splits up inputs for the mappers, then when mappers complete, a shuffle stage which coalesces and sorts the mapper outputs, and a reduce phase where inputs are generated for the reducer

Because of the conceptually simplicity provided by mapreduce, it translates well into threaded applications and can be designed in such a way as to make highly data parallel operations simple to write and optimize without requiring particularly deep understanding of threaded programming or data synchronization. As the programmer generally need only write stateless map() and reduce() functions, the amount of data synchronization and threading knowledge required by the user is fairly minimal. The framework also provides controls which allows for more advanced use cases, like sharing data or making optimizations for performance.

This proposal outlines the details of a version of mapreduce which is logically simple but very extensible, based heavily off of both distributed and threaded versions of mapreduce, allowing for implementations which work as distributed, threaded, or both. There are some simplifications in this implementation which are detailed later, but much of the flexibility of many of the implemented map-reduces is retained.

Basic Use

Usage of the mapreduce library includes the following main steps:

1. define a concrete mapper class which subclasses the mapper class. The mapper class defined a set of pre-defined types and receives a mapper_tag classification (the default is input_unaware). this mapper class will have to define start(), flush(), abort(), and map(). map() is the core worker function which receives an input value and an output iterator, and is expected to output key,value pairs to the output iterator (e.g. receive a string which is to be converted into word,count pairs).
2. define a concrete reducer class which subclasses the reducer class. The functions are effectively the same, with a reduce() function instead of a map() function. The reduce() function is similar to map() but instead receives a key, a value iterator, and an output iterator. The expectation is that the reducer() will “reduce” all values for a given key and will reduce those down to a single value.
3. declare a set of map_reduce_options which contain a set of parameters for the map_reduce class (number of workers, thread pool class, etc).
4. declare a concrete instance of the map_reduce class, templatized on the input/output iterator classes used to process values, and on the mapper/reducer classes previously defined.
5. call the run() function of the map_reduce object, which then consumes all inputs of the iterator passed in and passes all output to the output iterator (provided to the map_reduce object via the map_reduce_options). Functionally, the map_reduce works in 3 stages.

a. In the first stage, the mapper consumes all of the input values from the input iterator. These values are converted into key,value pairs and stored until all mappers complete.
b. In the second stage, the map_reduce class shuffles the values so that all values for a given key can be accessed at once, then sets of keys are put into a work unit and passed to the reduce phase (logically this acts like a multimap of key-value pairs).
c. In the last stage, the reducers are started and process all values for a given key, outputting the final output of the map-reduce pipeline (with the output target unspecified but normally this will be to some map structure as well).

Note, there are some additional optimizations which are available on top of the basic map-reduce:

- Ability to configure a combiner class. A combiner is a commutative/associative reducer which allows you to pre-aggregate values before they are shuffled for use by the final reducer. This mechanism only works when the output_type == value_type, but can provide a large impact on storage footprint (and potentially performance due to data locality and the cost of reading/writing to storage). For the library implementer, this would sit on top of the output of the map stage and would combine the output of the mapper with the previously stored map output and store the new combined output. For example, a sum combiner would take the old sum and add the new value on top.
- Ability to specify an input_splitter function which processes an iterator value and converts to 0 or more input type values. This can easily be used to chunk or store data in more efficient/natural ways (for example storing values in a set of binary files). This can also be used to create work units which are larger than a single input value (for example NxN blocks of a source input image). The identity_splitter is provided as a default, which passes the values untouched to the mapper, but replacement splitters can also be specified.
- Ability to specify a shard function which defines which keys are sent to which reducers. This capability allows for better control of object locality. The default shard function assumes keys can be hashed.

Example Code

```
class BucketizingMapper : public mapper<int, int, float> {
 public:
  void start() {}
  void flush() {}
  bool abort() { return true; }

  template <typename OutIter>
  void map(const int& input, OutIter output) {
    output = std::pair<int, float>(input % 10, static_cast<float>(input));
  }

};

class AveragingReducer : public reducer<int, float, float> {
 public:
  void start(unsigned int shard_id, bool is_combiner) {}
  void flush() {}
  bool abort() { return true; }
```

```
  template <typename InIter, typename OutIter>
  void reduce(
      const int& key, InIter* value_start, InIter* value_end,
      OutIter output) {
    double value_sum = 0.0;
    int num_values = 0;
    for (InIter* value_iter = value_start;
         (*value_iter) != (*value_end); ++(*value_iter)) {
      value_sum += *(*value_iter);
      ++num_values;
    }
    output = std::pair<int, float>(key, value_sum / num_values);
  }
};

// Execute the map-reduce
blocking_output_map out_map;
output_iter out_iter(&out_map);

map_reduce_options<BucketizingMapper, AveragingReducer, output_iter> opts
  {3, 19, 5, new simple_thread_pool, out_iter};

map_reduce<input_iter, output_iter,
           BucketizingMapper,  AveragingReducer> mr(opts);
buffer_queue<int> input_queue(1000);
input_queue.push(10); // avg = 10.0
input_queue.close();
queue_wrapper<buffer_queue<int> > input_wrap(input_queue);
mr.run(input_wrap.begin(), input_wrap.end());
```

Proposed Interface

```
// Configuration class which contains implementation specific configuration
// for the map-reducer library. Implementations may include things like
// * number of mapper tasks
// * number of reducer tasks
// * thread pool to execute from
// * mapper args
// * reducer args
// * configuration files
// This also contains a pointer to an OutIter type which is the mechanism for
// handling output from the reducer phase.
template <typename Mapper, typename Reducer, typename OutIter>
struct map_reduce_options {
  // Args to be passed into the mapper for construction
  Mapper::arg_type* mapper_args;
  // Args to be passed into the reducer for construction
  Reducer::arg_type* reducer_args;
};

enum input_tag {
  // capability which indicates the mapper/reducer wants a reference to the
  // input passed to map-reduce as well as to the actual mapper input,
  // providing some additional information about the source data (e.g.
```

```
  // filename, work task identifier). The data passed is implementation
  // specific.
  input_aware,
  // this indicates that the mapper/reducer does not care about the source
  // data so the mapper does not get called with the source work unit data
  // input_unaware, this is an even simpler interface which allows the
start()
  // and flush() functions to be called only once over the lifetime of the
  // mapper/reducer.
  input_unaware,
  // in the case that no work needs to be done on a work-unit level, the
  // start() and flush() functions are effectively part of the
  // constructor/destructor of the class
  input_ambivalent
};


template <typename I, typename K, typename V,
          typename Args=void,
          input_tag tag=input_unaware>
class mapper {
 public:
  // type of input to the map function
  typedef I input_type;
  // type of the key output of the map function
  typedef K key_type;
  // type of the value output of the map function
  typedef V value_type;
  // type of args passed to the mapper constructor
  typedef Args arg_type;
  // tag defining the type of mapper constructed
  input_tag mapper_tag = tag;
};

// prototype of the core mapper class (input_ambivalent and
// map_input_unaware). this is the core interface that any implementer will
// need to define.
class mapper_prototype : public mapper<I, K, V> {
 public:
  // Function which is called at the start of processing of some unit of
work.
  // When an input splitter is used, this is called on the boundary of each
  // work unit to allow for communication of data at the granularity of
  // the source work unit. Calling this should happen at least once per
  // mapper, but may be called in a cycle which looks like:
  // start() -> map(), map(), map() -> flush()
  //
  // This is primarily to allow for setting up data shared between mapper
  // inputs but which needs to be flushed or stored as part of a commit of
  // some data (e.g. statistics counters, or some additional output)
  void start();

  // main worker function of the mapper class.
  // input - represents the input value being passed to the mapper by the
  //   calling library
  // OutIter - output iterator which receives the mapper output. expected
that
```

```
  //   OutIter takes pair<key_type, value_type> as input.
  template <typename OutIter>
  void map(const I& input, OutIter output);

  // optional behavior which allows the calling library to abort the
  // mapper in case it is no longer needed. this will cause the map
  // function to stop executing its current input and become ready for
  // execution again. the caller may call flush if abort() returns true.
  // false implies the mapper cannot abort. may block the caller until
  // abort is successful.
  // Returns - whether abort was successful or not
  bool abort();

  // Finalizer function which is called once per work unit (per input
provided
  // to the map reduce library. Used to flush internal state accumulated
  // per work unit.
  void flush();
};

// Traits base class used to define a set of traits required of any
// mapper class (mainly typedefs). This class is not required by
// implementers but is convenient to use.
template <typename K, typename V, typename O,
          typename Args=void,
          input_tag tag=input_unaware>
class reducer {
 public:
  // key type of the input to the reducer (also the key type of the output)
  typedef K key_type;
  // value type of the input to the reducer
  typedef V value_type;
  // value type of the output of the reducer
  typedef O output_type;
  // type of constructor args to the reducer
  typedef Args arg_type;
  // tag defining the type of reducer constructed
  input_tag reducer_tag = tag;
};

// prototype of the core reducer class (input_ambivalent and
// map_input_unaware). this is the core interface that any implementer will
// need to define.
class reducer_prototype : public reducer<K, V, O> {
 public:
  // this is called once per unit of work before reduce is called for
  // any values in that work unit.
  // shard_id - represents an id which indicates which worker this is
  //   to allow for optimizations based on which worker is active.
  //   shards are assumed to be a zero-indexed list of workers from
  //   0..num reducer workers.
  // is_combiner - a special bit which indicates that the reducer is being
  //   used as a combiner and to potentially behave differently as a result
  //   (combiners require that the reduce function be associative and
  //   commutative, as combiners may only see a subset of data and will be
  //   asked to reduce other pre-reduced values).
  void start(size_t shard_id, bool is_combiner);
```

```
  // reduce function which takes a set of input values and processes
  // them as a unit. this receives input as an iteration over a set of
  // values and is expected to call the output = with a
  // pair<key, output_value> for each output value.
  template <typename InIter, typename OutIter>
  void reduce(
      const int& key, InIter* value_start, InIter* value_end,
      OutIter output);

  // this is the same as the abort function on the mapper class and should
  // return true if abort succeeded on the current reducer input.
  bool abort();

  // like the mapper flush function, this is called at least once in the
  // lifetime of the reducer, but should generally be called once per
  // work unit. the work unit is defined at the level of a shard of
  // data, which is an implementation specific unit of work optimization.
  void flush();
};

// map_reduce class is templatized on the mapper/reducer classes as well as
// some helpers which define how sharding is handled and if input needs to be
// split before processing. Uses a thread-pool to create mapper tasks.
//
// Must override shard_fn if the key type does not have a std::hash
// specialization defined or you want to use a non-hash based sharding
// function.
//
// template params:
// Iter - iterator type over input
// OutIter - output iterator where reduced values will go
// Mapper - mapper class which supports start/map/flush functions
// Reducer - reducer class which supports start/reduce/flush functions
// Combiner - reducer used to combine values prior to sending to the shuffler
// shard_fn - sharding function which defines which values go to which
//   reducers. defaults to using a hash-based sharding function and assumes
//   that std::hash is defined for the key type
// input_splitter - special function which can take the value type of the
//   input iterator type and generates values which can be consumed by the
//   map function. this defaults to an identity_splitter which does nothing
//   to the input and passes untouched to the mapper function.
template <typename Iter,
          typename OutIter,
          typename Mapper,
          typename Reducer,
          typename Combiner = identity_reducer<typename Reducer::key_type,
                                               typename Reducer::value_type>,
          typename shard_fn = default_shard<typename Mapper::key_type>,
          typename input_splitter =
              identity_splitter<typename Iter::value_type,
                                typename Mapper::input_type> >
class map_reduce {
 public:
  map_reduce(
      const map_reduce_options<Mapper, Reducer, OutIter>& opts)
     : opts_(opts) {}
```

```
  // blocking interface to execute the map-reduce library.
  // this function does all the work of setting up the mapper and reducer
  // tasks, defining work for the reducers, shuffling data, and waiting
  // for all data to be processed.
  // the contract is that when this function completes, iter will internally
  // be equal to end, and all values between iter and end will have been
fully
  // processed by the entire map reduce pipeline and all outputs will have
  // completed.
  // the pipeline is defined
  void run(Iter iter, Iter end, OutIter out_iter);

  // returns immediately after the map-reduce has started and will not
  // block waiting for the mapreduce to complete
  void run_async(Iter iter, Iter end);

  // blocks until run_async() completes. acts as a noop if run() has been
  //  called.
  void wait();
};
```

Input Aware Mapper and Reducer

There is an alternative version of the mapper and reducer classes which provides more visibility into the raw data and input metadata. For distributed mapreduce, a number of optimizations can be made when the mapper and reducer have some metadata, including things like shard id, filenames, directories, or reader objects. The google distributed mapreduce provides access to this information directly, though they complicate the interface significantly and are very implementation specific. The difficulty about making this generic is to provide access to such implementation specific metadata. Google's implemention places this metadata in a mapper-input object, but this API modifies this approach a bit by providing access to the raw work unit passed into the map reduce framework. The contents of this will vary. For a file based input, this may be as simple as the source filename, or for a fully distributed implementation, may contain filename, file offset, temp directory, shard id, etc.

This variation on the mapper and reducer is the same as the input_unaware version, but with a single extra parameter.

```
class mapper_prototype : public mapper<I, K, V> {
 public:
  void start();
  bool abort();
  void flush();

  // main worker function of the mapper class.
  // raw_input - reference to the input passed into the mapreduce framework
  //   with the same type as Iter::value_type for the Iter type passed into
  //   the map_reduce class.
  // input - represents the input value being passed to the mapper by the
  //   calling library
  // OutIter - output iterator which receives the mapper output. expected
that
```

```
  //   OutIter takes pair<key_type, value_type> as input.
  template <typename RawInputType, typename OutIter>
  void map(const RawInputType& raw_input, const I& input, OutIter output);

};

class reducer_prototype : public reducer<K, V, O> {
 public:
  void start(size_t shard_id, bool is_combiner);
  bool abort();
  void flush();

  // raw_input - raw input work unit passed into the reducer, which contains
  //   data required to generate value_start and value_end (e.g. the set of
  //   keys to be processed by this reducer and the map of all data, or the
  //   filename/offset being processed by the reducer)
  // key - next key to be processed by the reducer
  // value_start - iterator pointing to the first value for key
  // value_end - ending value iterator
  // output - output iterator to which output is sent from the reducer
  template <typename RawInputType, typename InIter, typename OutIter>
  void reduce(
      const RawInputType& raw_input,
      const K& key, InIter* value_start, InIter* value_end,
      OutIter output);
};
```

Input Ambivalent Mapper and Reducer

The input_ambivalent tags can actually simplify the map/reduce interfaces significantly by not requiring any explicit initialization/destruction. It's reasonable to just provide empty start/flush functions, but an input_ambivalent mapper/reducer do not require these functions, so a minimal interface for these is:

```
// prototype of the core mapper class (input_ambivalent).
// this is the core interface that any implementer will need to define for
// an input_ambivalent mapper.
class mapper_prototype : public mapper<I, K, V> {
 public:
  // main worker function of the mapper class.
  // input - represents the input value being passed to the mapper by the
  //   calling library
  // OutIter - output iterator which receives the mapper output. expected
that
  //   OutIter takes pair<key_type, value_type> as input.
  template <typename OutIter>
  void map(const I& input, OutIter output);

  // optional behavior which allows the calling library to abort the
  // mapper in case it is no longer needed. this will cause the map
  // function to stop executing its current input and become ready for
  // execution again. the caller may call flush if abort() returns true.
  // false implies the mapper cannot abort. may block the caller until
  // abort is successful.
```

```
  // Returns - whether abort was successful or not
  bool abort();
};


// prototype of the core reducer class (map_input_ambivalent and
// map_input_unaware). this is the core interface that any implementer will
// need to define.
class reducer_prototype : public reducer<K, V, O> {
 public:
  // reduce function which takes a set of input values and processes
  // them as a unit. this receives input as an iteration over a set of
  // values and is expected to call the output = with a
  // pair<key, output_value> for each output value.
  template <typename InIter, typename OutIter>
  void reduce(
      const int& key, InIter* value_start, InIter* value_end,
      OutIter output);

  // this is the same as the abort function on the mapper class and should
  // return true if abort succeeded on the current reducer input.
  bool abort();
};
```

Construction

The construction of mapper and reducer object is generally assumed to use the default constructor since mappers and reducers are generally stateless. But, there is a facility which allows an arg type to be defined by the mapper and reducer (default is a void argument). These values can then be added to the map_reduce_options object which can pass arguments to the mappers/reducers at construction time.

Note that this type can be left as void and no constructor provided and the library should be able to detect this case and not call the constructor if a constructor is not available.

Differences from the Google Implementation

The above interface is focused on an in-memory version of the map-reduce library with concessions for the distributed form of it. There are a number of interface changes which were intended to simplify the interface and abstract out a number of google-specific changes, including the following:

- Keys/Values in this library are not restricted to string types (though using this as a serialized format for communication is still possible, but is up to the implementer)
- At google file input is initially sharded before creating work units for the mappers. This is somewhat considered part of the map-reduce framework, but can be offloaded as a pre-processing stage, so it has been omitted from this implementation.
- There is a complex set of features where mappers/reducers can get metadata about themselves and the data which they are processing (filenames, shard id, file readers/ writers, output directories). This is all used to provide mechanisms to optimize the logic, but makes the code very implementation specific, so it has been dropped and included in the mapper_tag and reducer_tag traits which allow external data to be passed into the mapper optionally. This still requires that the implementer know the data format and details about the implementation, but is abstracted out from the core API (and generally

mappers/reducers can be written agnostic to this and optimized and specialized only as needed).

- The google implementation uses factory methods and method registration to configure remote workers in a very flexible way, but this a complex mechanism that assumes inheritance to make it work effectively. This concept has been dropped for the purpose of simplification (though without a major loss in terms of flexibility as it's still possible to create factories of pre-configured map-reduces for a given type specification).
- There are a couple of specialized functions which have some utility but are hard to map into this framework, including:
    - output_to_every_shard - this provides the ability to send a particular key-value pair to all work units sent to the reducer. doing so allows for special tracking and counting mechanisms to be built, but doesn't fit well with the output-iterator abstraction. this can be emulated by adding sentinel keys which provide metadata, or by creating an out-of-band communication mechanism keyed on the shard_id.
    - secondary_key - this allows there to be a value which isn't in the key but participates in how the data is sorted for processing. This is useful for having a particular ordering of data sent to the reducer phase, but doesn't map well into the simple Key-Value mapping function. There is no way to emulate this without making the keyspace significantly more complicated and incompatible with some C++ containers, so implementing this is left as a special feature of the implementation (e.g. exposing the value_comp in the multimap container)

Standard Library Interoperability

Because of the design of the current proposal (which uses iterators on input and output), the framework is compatible with a number of standard library mechanisms.

The expectation is that this will use a currently unstandardized executor interface, which would be passed in via the mapreduce options class. The framework could create its own threads as well, but having an executor/thread pool interface provides greater control and the current implementation assumes one. For a distributed version of this interface, the library will also have to manage launching tasks on remote machines, but that is out of scope of this proposal.

Conceptual Variations

Object Passing

The proposed interface above defaults to mapper and reducer classes have a default constructor, but can initialize a single-parameter arg pointer by specifying a non-void arg_type in the template. This allows the library to construct mapper/reducer objects on the fly. Moreover this type of approach is compatible with distributed map-reduce which would need to construct mapper and reducer objects on each remote machine.

An alternative to this is to have the mapper and reducer classes by copy-constructable and a prototype object be passed into the framework at initialization, allowing the objects to be more easily constructed and initialized. Having a copy-constructable mapper, would, for example allow for shared state to be easily created in the mapper constructor. It also enables mappers which could receive lambdas which would simplify the framework further to allow callers to be able to write map-reduces without writing a mapper or reducer class.

For example, this potentially enables a lambda-based map-reduce, something like the following (note this would actually require the ability to have template-based lambdas to work properly, as the input and output iterators are not fixed types):

```
map_reduce<queue::iterator, queue::iterator, lambda_mapper<int, int, float>,
lambda_reducer<int, float, float>> mr(opts,
 lambda_mapper<int, int, float>::create([] (int i) { return pair<int,
float>(i, sin(i))),
 lambda_reducer<int, float, float>::create([](int k, ...) {} ));
```

In fact, such an approach would allow for a functional variant of the mapreduce class which could perform the mapreduce operation without any notion of threading behavior (basically an algorithm which operates on iterators and outputs to an iterator using lambda mapper and reducers, similar to how Python can map/reduce over any iterable structure).

The big disadvantage of the copy-construction approach is that it makes a distributed implementation of the framework infeasible as the class instances would need to be serialized somehow.

Combiner and input_tag implicit

There is some complexity added to the interface because of the input_tag enumeration which allows for multiple implementations of the mapper class. There has been some discussion about making these concepts implicit such that the the presence of an input-aware map() function can be called using an enable_if style block (this could also be used to automatically define a combiner when a reducer declares itself as combinable). This was left out but is worth additional discussion.

Input Value Iteration

The proposed implementation provides two dimensions of iteration, the framework receives a begin and end iterator when run, but also supports an input_splitter concept which allows each input value to be divided into multiple inputs to the map-function. This is done to allow the caller to enqueue larger work units (e.g. a file chunk, or a range of values) for a number of reasons (particularly for performance reasons).

An alternative interface doesn't expose the input_splitter to the template interface of the map_reduce class and instead asks the input iterator to wrap the source input in an iterator-over-iterators. Doing so hides the source unit of work to the map reduce framework, effectively making the input appear to be an iterator over the input-type of the mapper, rather than another unit.

An example of this is the source providing an iterator over filenames (files containing lines of text), and mappers receiving a single line of text from each filename. With the input-splitter in the template, the map-reduce framework would read a filename, then the splitter would read the file one line at a time, passing the values into the mapper one line at a time. For the iterator-of-iterators approach, the framework would receive an iterator over lines of text and the iterator would be responsible for internally reading source values and converting the files into an iterator of map input values.

The difference between these approaches seems nominal except the iterator-over-iterators is logically simpler for the framework, but it also hides the source work unit from the map-reduce

framework, which imposes a number of problems for optimization. Particularly, the ability to track and change behavior at a work-unit level becomes impossible. The distributed versions of map-reduce, for example, use the chunking of inputs to implement fault-tolerance on a per-work-unit level. Threaded map-reduce can use the work-unit concept to keep statistics, implement more performance friendly work progress tracking, and to allow for flushing and reset operations at a natural unit of granularity. As such, the flexibility of the proposed interface outweighs the slight increase in interface complexity.

Distributed MapReduce

Much of the discussion about the proposed map_reduce framework talks about the ability to provide a distributed implementation without changing the programmer implementation significantly. Logically this is a goal of the proposed implementation, though in practice changes would need to be made in practice.

The proposed interface supports distributed map-reduces fairly easily as well as a number of distributed systems optimizations and fault tolerance mechanisms.

A few examples examples of this are given below:

- Source data and output data serialized by configuring string types for all of the system types
- Shuffle stage which coalesces mapper outputs and sorts them in prep for the reducer stage (this can be hidden in the map_reducer and iterators created on top of the sorted source files)
- Fault tolerance support at the work-unit level, allowing the framework to drop output from failed workers
- Duplicate work processing/deduplication per work-unit with worker abort
- **Specialized interface to source work-unit information via capabilities model (e.g. mapper which receives work-unit metadata in addition to the input value), for worker and input specific optimizations.