Operator Vectorization Library – A TensorFlow Plugin

Matthew Pickett, Karen Brems, Florian Raudies

Hewlett Packard Labs
HPE-2016-94

Keyword(s):
Machine learning; GPU acceleration; TensorFlow

Abstract:
TensorFlow is an interface for implementing machine learning applications that can be accelerated by using Graphics Processing Units (GPUs). It is rapidly becoming a standard tool in this space. TensorFlow consists of a high level API for constructing stateful dataflow graphs and a runtime which distributes and schedules the evaluation of operations in the graph onto various compute devices. TensorFlow provides an extensive library of operators, particularly those commonly used for machine learning applications. However for some deep learning applications such as recurrent networks, graph analytics or problems solved by dynamic programming, the library operators are not enough. The Operator Vectorization Library, or OVL, is a Python plugin into TensorFlow. OVL provides a Python API for defining high performance custom operators for the TensorFlow framework. It enables TensorFlow users to easily write, test, and use custom operators within Python and the OVL API instead of programming in C++/CUDA, without sacrificing performance. Applications running on GPUs may compute operations faster than they transfer data in and out of the host memory. Thus, these applications are memory bandwidth limited. Kernel fusion is the process of taking multiple operators, represented as vertices in the data flow graph, and merging them into a single operator. Fusion improves performance because it eliminates the fixed costs of launching an operator and can lower bandwidth requirements by eliminating extraneous copies from the compute device out to main memory and back, which may happen between operator calls. Despite the simplicity of programming and the acceleration gains exhibited by TensorFlow for some applications, the architecture of the TensorFlow framework does not support kernel fusion. The OVL optimizer provides automated kernel fusion of OVL operators at runtime. We chose a recurrent neural network known as the long short-term memory (LSTM) as a test case for the OVL optimizer. Using OVL and its optimizer we show a 2.3x speed-up over a pure TensorFlow implementation. OVL was released under the Apache 2.0 license in August 2016 at https://github.com/opveclib/opveclib. This paper describes the OVL interface and implementation.
Abstract
TensorFlow [1] is an interface for implementing machine learning applications that can be accelerated by using Graphics Processing Units (GPUs). It is rapidly becoming a standard tool in this space. TensorFlow consists of a high level API for constructing stateful dataflow graphs and a runtime which distributes and schedules the evaluation of operations in the graph onto various compute devices.

TensorFlow provides an extensive library of operators, particularly those commonly used for machine learning applications. However, for some deep learning applications such as recurrent networks, graph analytics or problems solved by dynamic programming, the library operators are not enough. The Operator Vectorization Library, or OVL, is a Python plugin into TensorFlow. OVL provides a Python API for defining high performance custom operators for the TensorFlow framework. It enables TensorFlow users to easily write, test, and use custom operators within Python and the OVL API instead of programming in C++/CUDA, without sacrificing performance.

Applications running on GPUs may compute operations faster than they transfer data in and out of the host memory. Thus, these applications are memory bandwidth limited. Kernel fusion is the process of taking multiple operators, represented as vertices in the data flow graph, and merging them into a single operator. Fusion improves performance because it eliminates the fixed costs of launching an operator and can lower bandwidth requirements by eliminating extraneous copies from the compute device out to main memory and back, which may happen between operator calls. Despite the simplicity of programming and the acceleration gains exhibited by TensorFlow for some applications, the architecture of the TensorFlow framework does not support kernel fusion. The OVL optimizer provides automated kernel fusion of OVL operators at runtime.

We chose a recurrent neural network known as the long short-term memory (LSTM) as a test case for the OVL optimizer. Using OVL and its optimizer we show a 2.3x speed-up over a pure TensorFlow implementation. OVL was released under the Apache 2.0 license in August 2016 at https://github.com/opveclib/opveclib. This paper describes the OVL interface and implementation.

1. Introduction
Hewlett Packard Labs has focused research efforts in the area of GPU accelerated cognitive computing and deep learning platforms for over five years. The culmination of this research was the open source release of the Cognitive Computing Toolkit (CCT) in April 2016 [2]. With the open source release of TensorFlow in November 2015, we decided to bring some key ideas of CCT to the TensorFlow community.

First, we wanted to add the ability to easily create high performing custom GPU operators in the language of the platform – i.e. Python, without forcing users to modify the TensorFlow code base. TensorFlow does provide an API for defining custom operators, but it requires the programmer to implement, build, and link custom C++ and CUDA code and to register that code in TensorFlow. It may
also require propagating operators into the Eigen [3] codebase, which is used by TensorFlow. These additional steps can be a productivity bottleneck for many data scientists who are more familiar with Python.

Second, we wanted to add the ability to fuse kernels and, thus, to improve the performance of TensorFlow applications. OVL was implemented to solve both of these problems. The OVL Python API allows users to easily write custom operators that can be executed within a TensorFlow application. The OVL optimization process will automatically merge compatible operators at runtime when allowed.

The ability to generate optimized machine code that runs on a CPU or GPU from a pure Python API is not new. A good example of this is the Numba framework [4]. Also pyCUDA [5] provides a Python wrapper to the CUDA driver API. OVL is different because it is integrated into TensorFlow. Theano [6] also provides the ability to create Python expressions that can be compiled and run on the GPU. But Theano does not have the distributed deployment capabilities or rapidly growing user community of TensorFlow.

Key features of OVL include:

- A single Python implementation is used to generate both C++ and CUDA operators and transparently link them into the TensorFlow run time, cutting out the overhead of implementing, testing, and maintaining operators for multiple hardware architectures.
- An optimizer which can fuse an unbounded number of qualifying OVL operators into a single function call, mitigating performance bottlenecks related to global memory bandwidth limits and operator launch overhead.
- A Python testing framework so that users can directly test and profile their operators, on both the CPU and GPU, against a Python-native reference like NumPy [7].
- Straightforward gradient definition, enabling OVL operators to be fully integrated in the middle of a broader neural network or another gradient-dependent TensorFlow graph.
- A standard library of OVL operators which can be optimized in conjunction with user defined ops and used to define operator gradients.

2. Implementation
Figure 1: Architectural diagram of OVL, which consists of the high level API, an intermediate representation that fully describes an operator, an operator merging optimizer, a code generator, and support for linking OVL ops into TensorFlow.

2.1 Python API
OVL operators are written by users in OVL’s Python-embedded domain specific language (DSL). The programming model of operators is similar to that of CUDA or OpenCL: conceptually an operator is a stateless function that is mapped over a set of user-defined thread or worker indices. Operators are defined as the Python interpreter encounters them but are evaluated lazily, allowing for numerical and performance optimizations to be applied to the entire graph of defined operators before evaluation time. OVL uses the Python interpreter to parse operators into its own intermediate representation, an approach which comes with some limitations on the use of Python-native calls such as assignment and conditional expressions. OVL provides its own conditional and assignment expressions to be used instead.

The OVL DSL is designed to strike a balance between productivity, performance, and portability and as such there are some important restrictions that differentiate it from similar approaches:
- The DSL is strongly typed and tensor shapes are part of the type system. This means that inputs and outputs must have a concrete shape - there is no support for dynamic tensor sizing.
- Individual worker elements cannot communicate and, as such, there are no synchronization barriers.
- Operators have no persistent state between calls.

Designing an OVL operator requires understanding a few key concepts and their corresponding implementations in the OVL API:
- An OVL operator is defined by creating a Python function and decorating it with the operator() decorator.
- Arguments to the operator are the tensors that it will operate on it at evaluation time.
- Output tensors are the only thing that can be returned from operators. They are defined with the output and output_like functions.
- Operators are implicitly mapped over a set of workgroup positions. The workgroup shape must be statically defined based on the arguments to the operator and must be either a single integer or a list of integers. The position_in function is used to define the workgroup shape and returns a PositionTensor object which is used to identify the position of the current worker element.
- Operators can be tested independently of the TensorFlow runtime using the OVL test infrastructure via the evaluate function. An explicit evaluate function is used so that operators can be lazily evaluated, enabling the opportunity for optimization.
- Operators are linked into the TensorFlow runtime by explicitly converting operator outputs to TensorFlow tensors with the as_tensorflow function.
- An OVL gradient operator is a special type of operator. It is defined by creating a Python gradient function and decorating it with the operator() decorator as well as the gradient() decorator.

2.2 Protobuf Intermediate Representation
OVL operators are parsed into their corresponding expression graph (expression DAG). Nodes of the graph are computations and edges are data flow tensors. Each expression DAG also defines input and output tensors. Protocol Buffers (protobuf) are Google’s language-neutral, platform-neutral, extensible
mechanism for serializing structured data [8]. The OVL expression DAG is serialized into a protobuf representation. The expression DAG protobuf is deserialized by both the OVL optimizer and code generator.

In addition, we use a protobuf serialized representation of the OVL gradient operator graph (operator DAG) in order to register the gradient function for OVL operators in TensorFlow.

Having a language and platform independent intermediate representation of OVL operators opens up the future possibility of other language bindings for OVL, or integrating OVL operators into other machine learning platforms.

2.3 Optimizer/Merger

A key motivation for designing OVL was to enable automatic merging of operators at runtime. Typically, dataflow languages are not restrictive enough for automated merging. Their language constructs and programming paradigm make automated merging impossible. For instance, TensorFlow has an operator graph, where many operators map to function calls in NVIDIA's cuDNN library [9]. Merging these library calls is impossible, as the thread organization is not part of the operator definition. For automated merging we need to know the thread organization to avoid read-after-write conflicts.

Likewise, CUDA's programming model has not been designed to enable automated merging. Because of the flexibility of interleaving host and device code in CUDA, merging is generally not possible. It is difficult for the compiler to detect places where kernels are directly executed after each other without any host code execution in between. Some research has been done in the area of optimizing CUDA code by kernel fusion on limited use cases with some promising results [10].

We designed OVL to give visibility into the thread organization of the operator. We represent the operators in an OVL application as a data flow graph (operator DAG). Furthermore, expressions within operators are represented as an expression DAG. Each expression DAG defines input and output tensors that are connected to other operators or are inputs or outputs. Inputs and outputs of expression DAGs define the edges of the operator DAG. In a nutshell, merging in OVL reduces the depth of the operator DAG while increasing the depth of the expression DAG. In OVL merging is simplified by: 1) defining one workgroup shape per operator, 2) visibility into the indexing pattern for reads and writes, and 3) the DAG representations.

We explain the merger through an example of splitting a matrix, manipulating row vectors, and then concatenating row vectors into another matrix (Figure 2). For this example we assume a is a 4 x 5 matrix, b, c, d, e, f, g, h are row vectors with 5 columns, and k is a 3 x 5 matrix. In code line 1 we split the matrix a across rows returning the four row-vectors b, c, d, e. In code line 2, we add the row vectors b and c and store the result in the row vector f. In code line 3 we add the row vectors d and e and store the result in the row vector g. In code line 4 we multiply the two row vectors f and g element-wise and store the result in the row vector h. In code line 5 we concatenate the row vectors f, h, g along the first dimensions of rows, resulting in the matrix k, which has three rows and five columns. We show this example as a matrix annotated dataflow graph in Figure 3.
Figure 2 – Example pseudo-code as defined by the user as operators op1-5. Very common operators like op2, op3, and op4 are contained in the standard operator library of OVL.

```
1  b, c, d, e = split(a, 0);   // Operator op1
2  f = b + c;                // Operator op2
3  g = d + e;                // Operator op3
4  h = f * g;                // Operator op4
5  k = concatenate(f, h, g, 0); // Operator op5
```

Figure 3 – A matrix annotated dataflow graph for our example.

We illustrated the work of the merger for operators by providing pseudo-code for the implementation of op1-5 and a hand-merged version, op6 (Figure 4). The keyword position in defines the workgroup shape and returns the worker index or thread index. For op1 there are nCol=5 workers, where nCol is a variable for the number of columns. Continuing in op1, we create output variables for four row vectors using the keyword output. We extract the 0th, 1st, 2nd, and 3rd row and assign these to separate variables b, c, d, and e. When merging op5 with op4 we follow the input variable h. In both ops the indexing pattern is the same and the workgroup shape is the same. We can merge op4 with op5.
In general, we can merge two operators if 1) their workgroup shapes match, 2) the write indexing pattern in the from-operator (here op4) matches the read indexing pattern in the to-operator (here op5), and 3) the tensor shape of the output variable matches the tensor shape of the corresponding input variable (here h). We employ our merger iteratively, starting with output variables. For each output variable and its corresponding operator (which computes this output) we find mergeable operators traversing the operator DAG from outputs to inputs. While we have operators that can be merged, merge them. We continue searching for mergeable operators using the merged operator DAG. See Figure 5 for the pseudo code.

```python
merger(opDag) {
    mergeInfo = getInfoMergeInfo(opDag); // Merge info contains op pairs
    while (mergeInfo.isNotEmpty) {
        opDag = merge(opDag, mergeInfo[0]); // Take op pair and merge
        mergeInfo = getInfoMergeInfo(opDag);
    }
    return opDag; // Return merged op dag
}
```

**Figure 4** - example for the merging of op4 and op5.

**Figure 5** – Pseudo-code for the automated merger.
We explain the merger code using our example. The operator DAG has five nodes and 11 edges (Figure 6 - first DAG). The workgroup shapes between op4 and op5 match and the indexing patterns for reads and writes match. We merge op5 with op4 (Figure 6 – second DAG). For this newly merged DAG we compute again the merge information. For this example, we assume that op2 and op4-5 are first in the list of mergeable operators (the pair op3 and op4-5 is part of the list too). We merge op2 and op4-5 (Figure 6 – third DAG). Then op3 becomes mergeable with op2,4-5, which gives the merged op2-5 (Figure 6 – fourth DAG). Finally, we can merge op1 with op2-5, which gives the merged op1-5 (Figure 6 – fifth DAG). In this example our merger stops after four iterations.

![DAG diagram](image)

**Figure 6 – Example sequence of our automated merger.**

2.4 Code Generator

When an OVL operator is evaluated using the OVL evaluate function, or it is turned into a TensorFlow operator using the `as_tensorflow` function, we generate both C++ and CUDA code using the expression DAG representation of the operator. At generation time, the exact sizes and types of the input and output tensors are known and are part of the generated source code. Note, the OVL operator being generated could be the result of merging two or more OVL operators. The generated code is then compiled using `g++` and NVIDIA’s `nvcc` compiler. The result is two shared libraries for each operator: one for C++ and one for CUDA that are stored on disk in an operator cache.

2.5 TensorFlow Custom Operator

The integration between OVL and TensorFlow is accomplished by registering a single custom TensorFlow operator called DynamicLib, using the TensorFlow custom operator API. This operator takes as arguments an arbitrary length list of input tensors of arbitrary number types, and an arbitrary length list of output tensors of arbitrary number types. Additional arguments specify call parameters of the OVL generated code like the shared library name, function name for CUDA and C++, and the output tensor types and shapes. The DynamicLib Compute method builds up the specific input parameter list, allocates and builds the output parameter list, and then loads and calls the corresponding OVL operator function from the operator cache.

The `as_tensorflow` function creates a DynamicLib op in TensorFlow for the OVL operator with specific tensor inputs and outputs of the operator. OVL also registers a gradient function for DynamicLib, which will take the serialized OVL gradient operator DAG for that operator if there is one, and also create DynamicLib ops for the gradient operators in TensorFlow. Note that both the OVL operator and its
gradient operators might have been merged by the OVL optimizer and the DynamicLib op that is created is for the resulting merged operator.

3. Results

To evaluate the potential performance gains of the OVL optimizer, we chose a recurrent neural network known as the long short-term memory (LSTM) model as our target use case. The LSTM is useful in applications of sequence analysis including speech and language analytics. The reference implementation we used was taken from the TensorFlow tutorial (available: https://www.tensorflow.org/versions/r0.11/tutorials/recurrent/index.html). To create a roughly equivalent level of complexity to the end user, we only changed the nonlinear mixing function that is at the core of the LSTM cell and added all necessary API functions and their gradients used by the LSTM function to the standard OVL library. See Figure 7 for a comparison of the user-level implementations.

![Figure 7: Comparison of the code used to implement the LSTM nonlinearity using (top) the OVL standard operator library and (bottom) the TensorFlow standard API.](image)

We then benchmarked the performance of the two implementations with and without optimization and compared the results to a manually optimized OVL operator (see Table 1). The kernel fusing optimizer reduces the operator runtime by a factor of 2.33. For the entire training process the runtime is reduced by a factor of 1.45. The number of separate GPU events went from 21 to 5. The manually optimized version still outperforms the automatically optimized version. It decreases the number of GPU events down to 2: one for the forward pass and one for the gradient. The automatic merger merges all the forward pass operators down to a single operator, but cannot merge all the gradient operators. This indicates that there is additional room for improvement in the OVL optimization strategy.

<table>
<thead>
<tr>
<th>method</th>
<th>op duration</th>
<th>op speedup</th>
<th># GPU Events</th>
<th>end-to-end training time</th>
<th>end-to-end training speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow</td>
<td>1255 µs</td>
<td>1.00</td>
<td>21</td>
<td>1428 s</td>
<td>1.00</td>
</tr>
<tr>
<td>OVL un-optimized</td>
<td>1405 µs</td>
<td>1.27</td>
<td>19</td>
<td>1405 s</td>
<td>1.02</td>
</tr>
<tr>
<td>OVL optimized</td>
<td>539 µs</td>
<td>2.33</td>
<td>5</td>
<td>983 s</td>
<td>1.45</td>
</tr>
<tr>
<td>Manually optimized</td>
<td>87 µs</td>
<td>14.43</td>
<td>2</td>
<td>922 s</td>
<td>1.55</td>
</tr>
</tbody>
</table>

Table 1: Results of profiling the isolated LSTM nonlinearity function and the end to end training time of the entire LSTM for un-augmented TensorFlow, for OVL un-optimized, for OVL optimized, and a manually optimized
implementation. TensorFlow version 0.11.0rc1, opveclib version 1.0.1, running on an HP Z820 workstation with a GeForce GTX Titan.

4. Future Work

OVL was released under the Apache 2.0 license in August 2016 at https://github.com/opveclib/opveclib. Moving forward, we aim to improve the optimizer and demonstrate the value of the OVL to additional applications. We also plan to add to the standard library of OVL operators.

5. References