Polyhedral Optimization of TensorFlow Computation Graphs

Benoît Pradelle, Benoît Meister, Muthu Baskaran, Jonathan Springer and Richard Lethin
Reservoir Labs
lastname@reservoir.com

***Abstract*—We present R-Stream·TF, a polyhedral optimization tool for neural network computations. R-Stream·TF transforms computations performed in a neural network graph into C programs suited to the polyhedral representation and uses R-Stream, a polyhedral compiler, to parallelize and optimize the computations performed in the graph. R-Stream·TF can exploit the optimizations available with R-Stream to generate a highly optimized version of the computation graph, specifically mapped to the targeted architecture. During our experiments, R-Stream·TF was able to automatically reach performance levels close to the hand-optimized implementations, demonstrating its utility in porting neural network computations to parallel architectures.**

I. Introduction

Deep Convolutional Neural Networks (DCNN) [1], and more generally deep learning, recently reached maturity. Impressive results achieved in recent years demonstrated the technology was ripe for general, practical use. New applications are developed every day, and deep learning is already ubiquitous in our lives. This considerable activity around machine learning is becoming increasingly structured around a few common tools. For instance, Caffe [2], Torch [3], CNTK [4], and TensorFlow [5] are popular frameworks commonly used to develop and exploit neural networks. These frameworks are based on a similar concept: high-level operations such as convolutions and pooling are exposed to the user, who can design networks simply by composing them as operators. The frameworks also empower users by facilitating data preprocessing and streamlining back-propagation for training.

Applications based on neural networks often require strict performance constraints to be enforced, such as when performing interactive tasks. They also require high throughput (bandwidth) such as when performing many queries simultaneously. To minimize the latency and bandwidth required to process an input sample, neural networks frameworks rely on highly optimized implementations. A common approach for speeding up neural network processing consists in building a hand-optimized library of DCNN operators.

While this method significantly improves the performance of the computations, the hand optimization effort is tedious and error-prone. It is also inherently unable to exploit optimization opportunities available by combining successive operations. For instance, an element-wise operation and a convolution can be computed more efficiently if both operations are fused. In order to benefit from these optimization opportunities, several graph-based optimizers have been proposed and are currently being developed. The most representative approach is XLA, a just-in-time compiler for TensorFlow computation graphs. XLA has shown its ability to significantly speed up the computations performed in TensorFlow, but it seems limited to basic pattern-matching optimizations. Only simple cases of fusion and array contraction can be realistically achieved with this method.

Our contribution is to extend and generalize the approach of graph optimizers through polyhedral optimization techniques. Compilers and optimizers based on the polyhedral model can apply powerful code transformations on programs, using a precise mathematical representation of the code.

Polyhedral optimizations encompass fusion and array contraction, but they also subsume any combination of loop fusion/fission, interchange, skewing, and reversals. Data dependencies are exact in the polyhedral model, enabling automatic parallelization and other common memory-oriented optimizations such as loop tiling [6] and data layout transformations [7]. The polyhedral model is most precise on regions with affine constructs [8], which include most of the classical neural network operators.

In this paper, we present R-Stream·TF, a new optimizer for TensorFlow computation graphs. R-Stream·TF emits high-level sequential C code implementing the exact computations performed in the input TensorFlow graph. The generated C code is specific to the graph: it is specialized to the exact tensor shapes and types used as the input and output of every operation. The generated C code is then automatically parallelized and optimized by R-Stream [9], a polyhedral compiler developed by Reservoir Labs. R-Stream optimizes the computation specifically for the target architecture. R-Stream supports numerous targets including common x86 CPUs and GPUs and can generate the code to parallel programming models including OpenMP, CUDA, OpenCL, POSIX threads and task-based runtimes APIs [10], [11].

R-Stream·TF extracts and merges TensorFlow operator subgraphs, and lets R-Stream apply a full range of polyhedral optimizations to the underlying computations. Such transformations are specific to the target platform, which can have several levels of parallelism and any combination of caches and scratchpads [12]. The result is a set of highly optimized parallel TensorFlow operators, which are reintegrated into the original TensorFlow computation graph.

The main benefit of our approach is the ability to use the full set of polyhedral optimizations within computation

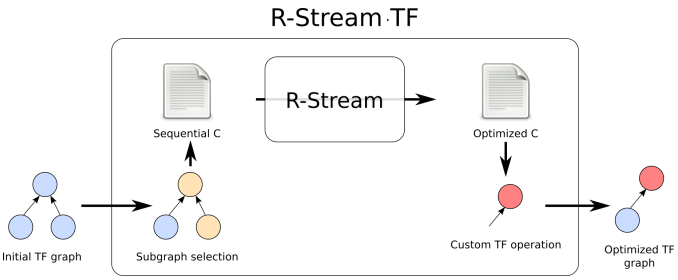

Fig. 1. TensorFlow graphs are converted into simple sequential C code, optimized using the R-Stream compiler, wrapped in a custom TensorFlow operators, and finally stiched back in the computation graph.

subgraphs. This ability is superior to current approaches based on domain-specific optimizations, since it enables several additional optimizations to be performed automatically on computation graphs. Because the optimizations applied to the graph are both specialized to the graph itself and to the target architecture, R-Stream·TF generates highly optimized code, specifically tailored for the target platform. This makes R-Stream·TF an adequate automatic porting tool, providing an optimized support for TensorFlow computations on new architectures.

The rest of the paper is organized as follows. The design of R-Stream·TF is presented in details in Section II. The tool has been implemented and evaluated on popular DCNNs. The evaluation results are presented in Section III. We compare R-Stream·TF to existing systems in Section V, before concluding in Section VI.

II. Design

A. Overview

The overall flow of R-Stream·TF, described in Figure 1, starts with a TensorFlow computation graph and results in an optimized graph with the same semantics. The optimization process is performed in several successive steps. First, optimizable subgraphs of the overall operator graph are identified. Restricting the process to well-structured subgraphs ensures that optimization is possible and tractable. Simple sequential C code is then generated for every identified operator. The sequential source code is sent as-is to R-Stream to be parallelized and optimized. The resulting parallel code is wrapped in a custom C++ TensorFlow operator automatically generated by R-Stream·TF. The operator itself implements the operation API defined by the framework, allowing the optimized code to be seamlessly reintegrated to the TensorFlow computation graph. As a result, R-Stream·TF produces an optimized computation graph based on automatically-generated custom operators. The optimized graph can then be used in lieu of the original graph.

B. Subgraph Selection

Polyhedral optimization scales super-linearly in the number of statements, hence practical optimization time constraints somewhat limit the number of nodes in subgraphs. While this number can be quite large, the number of nodes in current DCNNs is typically larger. In order to ensure that the optimization remains tractable, R-Stream·TF pre-processes the input graph, extracting subgraphs that are optimized independently from each other.

Fig. 3. Three specializations of the element-wise addition. The generated functions have specific data types, data sizes, and broadcasting even though the TensorFlow operator is the same.

Partitioning operator graphs in order to expose better optimization opportunities and maintain scalability (i.e., tractable optimization times) has been studied many times over, from instruction set synthesis [13] to streaming graphs (with e.g., [14], [15]). Optimality of subgraphs is typically defined by the amount of computation and data reuse within the subgraph, and the amount (and weight) of resulting in- and out-edges. In parallel computing frameworks, grain of parallelism and load balancing are also important optimality criteria.

The most impactful constraints are similar in our case. Additionally, code generators may not be available for some operators, which should then not be included in a subgraph to optimize. While we plan to implement more sophisticated subgraph selection algorithms in the future, we meet the proof-of-concept objective of this paper using a simple two-step approach. First, we identify connected subgraphs in the overall computation graph that are exclusively made of operations for which a code generator is available. Second, the connected subgraphs are partitioned when they are estimated to be too large for the optimizer. These steps are illustrated in Figure 2. The second step is expressed as a balanced k-way partitioning problem, where the objective is to minimize the number of graph edges between two partitions. Edges in the computation graph represent tensors on which the operations are performed. While R-Stream is free to change the data layout in the tensors *within* partitions, transforming the layout of tensors used across several partitions is illegal. Thus, by minimizing the number of edges between the partitions, R-Stream·TF increases data layout optimization opportunities for R-Stream, including array contraction, expansion, and spatial locality optimizations.

R-Stream works from C code, which is generated as the next phase of the R-Stream·TF() optimization process.

C. Operator Code Generators

To optimize the selected subgraphs, R-Stream·TF first generates sequential C code implementing the operations performed in the subgraph. For every operator in a selected subgraph, the code generator corresponding to the operator kind is identified. It first generates a function header where all the tensors are passed as pointers to C arrays arguments. The tensor arguments are monomorphic, meaning that if several types are allowed in the TensorFlow graph, they will result in different functions being generated. The body of the generated function implements the operator semantics, generally as a set of loop nests.

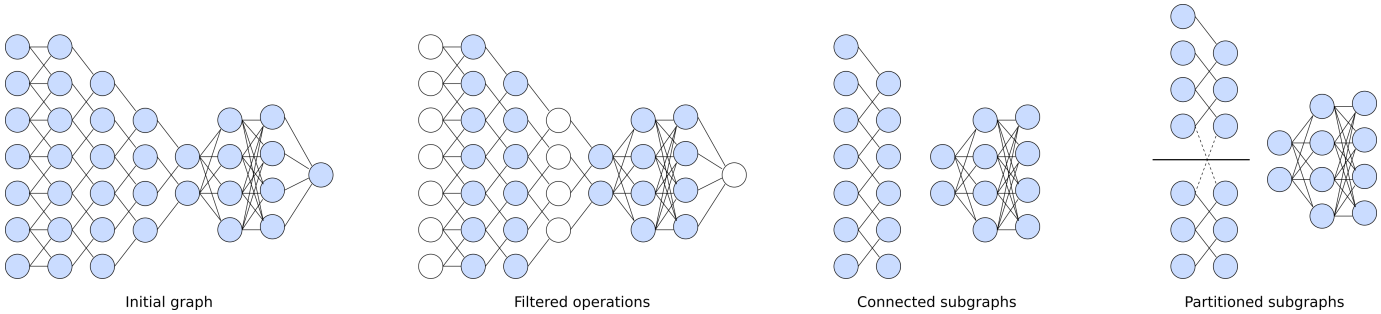

Fig. 2. Connected subgraphs of supported operations are computed first, before partitioning the large subgraphs into smaller ones to improve the optimization scalability.

D. Subgraph Code Generator

Once a function is generated for every operator of a selected subgraph, a subgraph function is generated. The subgraph function calls the individual subgraph operator functions, materializing the actual subgraph computation. R-Stream·TF currently restricts the subgraphs to be acyclic, simplifying the generation of subgraph functions. The operators in the subgraph are topologically sorted and a sequence of calls to the corresponding operator C function is generated as the subgraph function body. Similarly to the operator functions, the subgraph function accepts as its inputs a pointer to a C array for every tensor used in the computation.

The subgraph function is marked as being a region of interest for R-Stream using a pragma directive. R-Stream is also instructed to inline all the operator functions, which is easily done since their definitions are generated in the same compilation unit. The operator functions along with the subgraph functions of every selected subgraph are generated using the same process and finally sent to R-Stream to be optimized.

E. R-Stream Optimization

Using the polyhedral model, R-Stream infers the data dependencies of the input program from its C form. Based on the exact dependence information, R-Stream computes a new schedule for the program statements. The schedule can be seen as a combination of loop fusion, fission, skewing, interchange, and reversal. The scheduler used in R-Stream tries to maximize parallelism while optimizing locality, contiguity, vectorization, and data layout [9], [16]. After scheduling, R-Stream performs other important non-affine transformations such as tiling, along with the generation of explicit memory (e.g. scratchpad, virtual scratchpad) management and communication (e.g. DMA) instructions. The optimizations that R-Stream performs are all parameterized by a model of the target platform encoded in an XML hierarchical/heterogeneous architecture machine modeling description language. R-Stream then generates output C using "unparsing" techniques that conform to idioms appreciated by downstream or "backend" compilers (e.g., gcc, icc); this enables further backend optimizations such as typical scalar optimizations and the use of vector opcodes.

Tensor computing is an excellent match to the polyhedral model.[1] The loop extents and access functions are affine. Working on tensors rather than matrices results in deeper loop nests presents optimization challenges beyond the classical techniques used by library writers and beyond the reach of non-polyhedral classical loop optimizers, but which are very much in scope of a modern polyhedral compiler such as R-Stream.

While it would be straightforward to engineer a translator to go directly from TensorFlow IR to a polyhedral IR (in the case of R-Stream, the Generalized Dependence Graph (GDG)), we chose for this initial prototype the simple path of going through C. The code generators implemented in R-Stream·TF generate C code specifically targeted at the polyhedral model. The generators may produce visually non-intuitive code for some operators, but the code is specifically structured to be easily and immediately raised to the polyhedral representation. In some sense, R-Stream·TF uses a subset of C as an intermediate representation to communicate TensorFlow computations to polyhedral compilers. The code generator also benefit from extensions of the polyhedral model implemented in R-Stream to support more operators. For instance, R-Stream supports data-dependent conditions, which allows operators such as the rectifier activation function to be supported.

F. TensorFlow Operator

TensorFlow exposes a public C++ API to specify custom operations. R-Stream·TF generates the code implementing the API for every subgraph, in effect generating a custom optimized TensorFlow operator for every selected subgraph.

The generated TensorFlow operator declares itself to the framework, detailing the expected inputs and outputs for the

[1] Raising generic C codes into a polyhedral representation can be a complex problem, when programmers or library writers have made manual optimizations (e.g., parallelization, tiling, ...) based on domain knowledge which cannot be easily inferred from their program source. Such manual optimizations are often not performance portable (or portable at all) to new platforms, thus their action of performing manual optimization "bakes the code" to that one original target. To re-optimize to a new architecture through the polyhedral model, such manual optimizations often have to be reverted to produce an efficient polyhedral representation of the program. Unknown aliasing and overflowing arithmetic are among the challenges of such "un-baking." With modern compiler tools like R-Stream now available, it would a much more sustainable practice for programmers to express their code originally in a high-level, domain-specific manner.

custom operator. The operator also checks and validates its inputs. Although this is not required, generating such guards helps in debugging R-Stream·TF itself and ensures that the user does not change the input specification by transforming the graph after it has been optimized. Finally, the operator performs the required memory allocations for the tensors identified as being temporary or output tensors. Temporary tensors are tensors that do not exist before the operation is run but do not need to be maintained in memory after the tensor execution. Output tensors are created by the operation but are consumed later by other operations in the graph. Both temporary and output tensors are allocated before starting the computation and a pointer to the raw tensor data content is acquired and transmitted to the code optimized by R-Stream. R-Stream·TF reuses tensors as much as possible to limit the number of tensors required at any time during the optimized subgraph execution.

All the custom operators generated by R-Stream·TF are compiled into a shared library. Because the tool implements the TensorFlow API, the generated library can be easily loaded by calling a standard TensorFlow function.

As a final step, R-Stream·TF edits the graph structure by removing the original subgraphs and replacing them by the optimized custom operators. The result is generated as a standard protobuf file which can also be loaded in TensorFlow using the standard API.

G. Leveraging Broadcast

TensorFlow automatically expands the input tensors to match the largest one in several operations. This expansion is called *broadcasting* in the TensorFlow terminology and consists of inflating a tensor by duplicating it. Broadcasting is a convenient flexibility allowed by the framework to help the user express operations such as an element-wise addition with a scalar using the general tensor-based element-wise addition operator. R-Stream·TF generates operator functions that account for broadcasting without explicitly copying the data. For instance, an element-wise tensor addition can be generated as an addition with a scalar to match the broadcasting semantics. Broadcasting is one of the several specializations performed when generating the operator code. As illustrated in Figure 3, the different data types, data sizes, and broadcasting can result in many different variants of the operator functions, which would not be tractable if the code were written by hand but can be easily managed when the code is generated automatically.

III. Experiments

In order to assess the relevance of our approach, we evaluate R-Stream·TF by running it on popular deep neural networks. The time required to perform an inference of the optimized graph has been measured and compared when using different optimizers as well as the default TensorFlow setup.

We first evaluate R-Stream·TF when calling no optimizer to parallelize and optimize the code generated. The goal of this measurement is to determine what performance level can be reached if the graph operators are naively implemented, using their textbook definitions. This is typically the performance that would be reached by an inexperienced developer when porting TensorFlow to a new platform. Next, we evaluate R-Stream·TF with several polyhedral compilers, allowing us to better estimate the range of performance that can be reached by this class of tools and how performance is impacted by the capabilities of each optimizer. Finally, we compare R-Stream·TF to the standard TensorFlow performance. TensorFlow defers the operation optimization to the Eigen library, a collection of highly-optimized kernels. Because a library cannot implement optimized versions of any combination of kernels, the Eigen library is limited to some individual kernels and a few common combinations. On the other hand, the library provides extremely well-optimized implementations for the supported kernels. Hence, the performance reached by TensorFlow can be considered as that of a well-optimized implementation, even though not all the optimizations opportunities are exploited. In order to optimize the graph computations further, TensorFlow also provides XLA, a compiler exploiting operation fusion to improve the performance of the computations. Despite all our efforts, we were not able to have XLA to produce correct results with our experimental setup, which ruled it out of evaluation.

We ran the experiments on a standard Ubuntu 16.04 system with an Intel Core i7-4600U processor, using the standard version 1.2.1 package of TensorFlowprovided for our system. We evaluated R-Stream·TF in different configurations using two popular deep learning graphs: Inception versions 3 and 4. We froze the graphs using the learned weights provided by Google, emulating a production-ready setup. The graphs were evaluated when inferring the sample image of Admiral G. Hopper provided with TensorFlow. This image choice allowed us to guarantee the correctness of our setup, since the expected output of the graphs has been published for this input image. For comparison, we also used PPCG version 0.07 plus all the commits performed in the master branch until June 19, 2017. PPCG was run using the following options: "`--target=c --openmp --tile`". We also evaluated Polly, as provided in LLVM 5.0, as a polyhedral optimizer for the code generated by R-Stream·TF. Polly was run using the following Clang options: "`-mllvm -polly -mllvm -polly-parallel -mllvm -polly-vectorizer=stripmine`".

In the evaluated implementation of R-Stream·TF, the element-wise addition, subtraction, and multiplication, convolutions, and rectifying linear units were optimized. R-Stream·TF can optimize only operations for which a code generator is available and we limited our implementation effort to these common operations. The system can easily be extended with more operations as needed. The effort required to add a new operation is a matter of hours for a single developer, and this can then be used for any input network on any target.

The measured execution times of one inference on the experimental platform are presented in Figure 4 and Figure 5 for different optimization backends. The execution time of the naive code generated by R-Stream·TF before applying

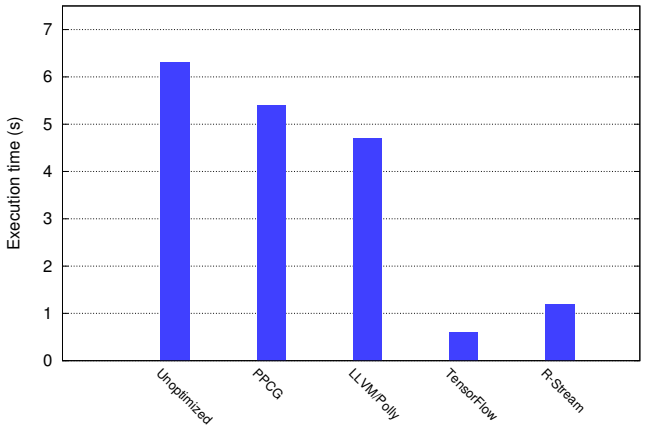

Fig. 4. Execution time of an inference of Inception 3 using different optimization backends.

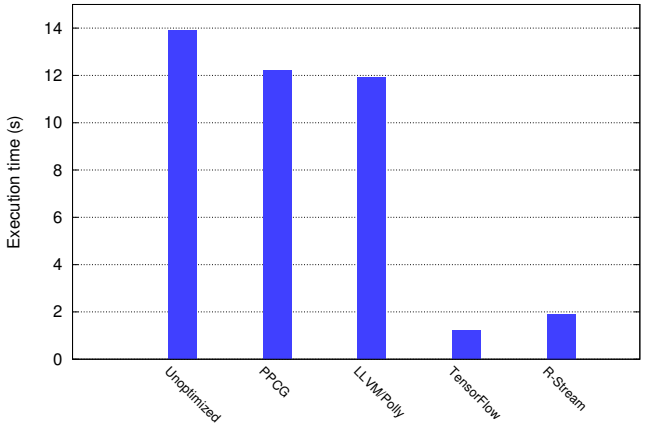

Fig. 5. Execution time of an inference of Inception 4 using different optimization backends.

any optimizer corresponds to the "Unoptimized" entry in the figures. The open-source PPCG polyhedral compiler was used to optimize the code generated by R-Stream·TF and is shown as the "PPCG" entry in the figures. Similarly, the code optimized by Polly is shown as "LLVM/Polly" and "R-Stream" is used to represent the entries where R-Stream is used to optimize the code generated by R-Stream·TF. Finally, the "TensorFlow" entry represents the reference execution time using the hand-optimized Eigen library provided with TensorFlow.

The results presented in Figure 4 and Figure 5 show that a polyhedral compiler can reach a performance level close to that achieved using highly optimized libraries. The execution time reached by the R-Stream optimized code, "R-Stream" in the figures, is significantly lower than that of the unoptimized version, emphasizing the optimization power of the polyhedral compiler. Specifically, R-Stream speeds the generated code up by 425% in Inception v3 and 630% in Inception v4, a substantial achievement considering that the experimental platform is a modest dual-core processor. Despite this achievement, the optimized subgraphs still suffer from a $2\times$ slowdown in Inception v3 to a 60% slowdown in Inception v4 compared to the hand-optimized "TensorFlow" implementation, exposing optimization opportunities currently not exploited by R-Stream. The measured slowdown should be contrasted with the amount of human effort required to produce both codes. In the case of the TensorFlow reference, a significant amount of effort had to be invested to produce the highly optimized libraries used by TensorFlow. Such optimizations are not only complicated to produce and debug, even for expert developers, but they are also not portable, even across different processors of the same family (if we include performance portability). When a new platform needs to be supported, a new effort has to be initiated by experts, often using different techniques than those used for the other supported platforms. On the other hand, the optimization performed by R-Stream was achieved with no human intervention in a few minutes. Such extreme productivity improvement and the resulting performance level indicate that R-Stream·TF is a useful tool for quickly generating highly optimized implementations of TensorFlow. The optimized code generated by R-Stream·TF can be used as-is on platforms for which no optimized implementations of the kernel libraries used by TensorFlow are available. It can also provide an optimized baseline implementation of TensorFlow which can be progressively replaced by hand-optimized kernels when needed and as the resources become available to perform those optimizations.

From the measured execution times, it is also clear that not all the polyhedral optimizers reach the same level of performance. We evaluated three different polyhedral optimizers considered as robust optimizers but the achieved performance varied considerably across optimizers during our experiments. The different levels of performance are due to the different heuristics and designs employed in the tools. For instance, PPCG and Polly both use the scheduling algorithm implemented in the ISL library [17]. On the other hand, R-Stream uses the JPLCVD scheduler [18], [19], which exploits a different set of heuristics and techniques to determine the best schedule for a program. Similarly, R-Stream is able to automatically determine relevant tile sizes for the loop nests, while PPCG cannot and falls back to default tile sizes in the absence of further instructions. Such different designs result in different optimization decisions and explain the variation in the performance reached by the various optimizers.

IV. Enabled Experiments/Work

With R-Stream·TF many new experiments and future developments are enabled, including:

- Expansion of the set of TensorFlow operators supported.
- Greater exploration of subgraph formation heuristics. R-Stream includes special features for greatly improving the scalability of polyhedral optimization, which may enable subgraphs of large size to be handled, and for more complex architecture targets to be addressed.

- A comparison with XLA and a deeper investigation of the gap with hand code.
- Expansion to additional deep-learning frameworks (Caffe, etc.)
- Addressing the opportunities of sparsity. R-Stream has some ability to optimize code working on sparse matrices and tensors.
- Generating optimized code for training.
- Exploitation of R-Stream's ability to generate code for distributed architectures, GPUs and event-driven task (EDT) dataflow runtimes.
- Applicaiton of R-Stream to model and generate code for specialized deep-learning architectures.
- Just-in-time compilation improvements and direct-to-GDG translation.

V. Related Work

Polyhedral optimizations are integrated in most of the mainstream compilers to perform advanced code transformations and improve execution time. For instance, Graphite [20] is a polyhedral optimizer in GCC and Polly [21] is the counterpart for LLVM. Independent polyhedral compilers also exist, though most of them are research projects and tend to become unsupported quickly. Notable polyhedral compilers still maintained are Pluto [22] and PPCG [23]. Most of the polyhedral tools are currently backed by the ISL library [24]. R-Stream [9] is a commercially supported polyhedral compiler with additional capabilities and supported platforms. R-Stream·TF is naturally based on R-Stream but can exploit any optimizer, polyhedral or not, that accepts C files as its input. This flexibility is demonstrated in the experimental section.

R-Stream·TF is essentially a polyhedral compiler for a domain specific language: TensorFlow graphs. Polyhedral optimizers based on DSLs have already been proposed for domains where regular data structures and computations are common [25], [26]. R-Stream·TF is however, to the best of our knowledge, the first attempt at performing automatic polyhedral optimizations on neural network computations.

There is currently intense activity around software frameworks oriented towards neural network computations. The high interest in this domain is reflected in the numerous frameworks available, most of them backed by significant companies or organizations. Caffe [2], the Cognitive Toolkit (CNTK) [4], Torch [3], Theano [27], and TensorFlow [5] are among the most popular frameworks. Interestingly, while all these frameworks compete against each other by providing roughly the same set of capabilities, they are all based on the same design, with an operation graph as their core concept. Such uniformity across all the platforms is doubly beneficial to our approach. First, the ecosystem of frameworks is still unsettled and it is unlikely that all the competing approaches will be maintained in the future. However, since all the popular frameworks are based on the same design, it is likely that the approach implemented in R-Stream·TF will remain relevant. Second, because the frameworks share similar designs, R-Stream·TF could be easily ported to another framework, extending its applicability beyond the sole TensorFlow framework.

Neural network software frameworks are sometimes able to optimize the computation graphs in a holistic way, similarly to what is done in R-Stream·TF. Relevant examples include NNVM in MXNET [28], Intel Nervana Graph, and, closer to our work, the XLA compiler of TensorFlow. The most advanced optimizations available in these tools propagate the constants in the graph, reduce memory usage, and fuse operators. TensorFlow provides a dedicated optimizer, XLA, performing ahead-of-time and JIT optimizations on the graph. XLA relates to R-Stream·TF since both tools have the same goals: specializing the graph code to the specific operation parameters and hardware platform in order to improve the computation performance. However, R-Stream·TF exploits the polyhedral model and all the associated optimization techniques to optimize the computation instead of ad-hoc optimizations specifically designed for the graphs. Using the polyhedral model generalizes the representation of the graph operations present in TensorFlow and significantly extends the set of optimizations that can be performed to the graphs.

VI. Conclusion

R-Stream·TF exploits the polyhedral model to optimize TensorFlow computations. The optimizations are performed by an existing polyhedral compiler specifically for a graph and a target architecture and without human intervention. During its evaluation, R-Stream·TF reached performance levels close to that of the hand-optimized code provided with TensorFlow for modern x86_64 processors. Such ability to automatically produce highly-optimized code makes R-Stream·TF an adequate tool for generating an optimized baseline implementation for TensorFlow computations on a new architecture.

Acknowledgment

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the Defense Advanced Research Projects Agency or the U.S. Government. This document was cleared by DARPA on August 23, 2017. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," 2014.

[3] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, no. EPFL-CONF-192376, 2011.

[4] D. Yu, K. Yao, and Y. Zhang, "The computational network toolkit [best of the web]," *IEEE Signal Processing Magazine*, vol. 32, no. 6, pp. 123–126, Nov 2015.

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Man, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Vigas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2015. [Online]. Available: http://download.tensorflow.org/paper/whitepaper2015.pdf

[6] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*. New York, NY, USA: ACM Press, Jan. 1988, pp. 319–329.

[7] P. Clauss and B. Meister, "Automatic memory layout transformation to optimize spatial locality in parameterized loop nests," *ACM SIGARCH, Computer Architecture News*, vol. 28, no. 1, 2000.

[8] P. Feautrier, "Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time," *International Journal of Parallel Programming*, vol. 21, no. 5, pp. 313–348, Oct. 1992. [Online]. Available: citeseer.ist.psu.edu/feautrier92some.html

[9] B. Meister, N. Vasilache, D. Wohlford, M. M. Baskaran, A. Leung, and R. Lethin, "R-stream compiler," in *Encyclopedia of Parallel Computing*, 2011, pp. 1756–1765.

[10] N. Vasilache, M. Baskaran, T. Henretty, B. Meister, M. H. Langston, S. Tavarageri, and R. Lethin, "A tale of three runtimes," arXiv:1409.1914.

[11] B. Meister, M. M. Baskaran, B. Pradelle, T. Henretty, and R. Lethin, "Efficient compilation to event-driven task programs," *CoRR*, vol. abs/1601.05458, 2016. [Online]. Available: http://arxiv.org/abs/1601.05458

[12] B. Pradelle, B. Meister, M. M. Baskaran, A. Konstantinidis, T. Henretty, and R. Lethin, "Scalable hierarchical polyhedral compilation," in *International Conference on Parallel Processing (ICPP)*, 2016.

[13] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proceedings of the 40th annual Design Automation Conference*. ACM, 2003, pp. 256–261.

[14] "Cola: Optimizing stream processing applications via graph partitioning," in *International Middleware Conference, ACM/IFIP/USENIX (Middleware)*, 2009.

[15] M. Dayarathna and T. Suzumura, "Automatic optimization of stream programs via source program operator graph transformations," *Distributed and Parallel Databases*, vol. 31, no. 4, pp. 543–599, Dec 2013. [Online]. Available: https://doi.org/10.1007/s10619-013-7130-x

[16] R. A. Lethin, A. K. Leung, B. J. Meister, and N. T. Vasilache, "Methods and apparatus for joint parallelism and locality optimization in source code compilation," Sep. 2009.

[17] S. Verdoolaege and G. Janssens, "Scheduling for ppcg," Department of Computer Science, KU Leuven, Tech. Rep., 2017.

[18] R. A. Lethin, A. K. Leung, B. J. Meister, and N. T. Vasilache, "Methods and apparatus for joint parallelism and locality optimization in source code compilation," Sep. 2009.

[19] C. Bastoul, R. A. Lethin, A. K. Leung, B. J. Meister, P. Szilagyi, N. T. Vasilache, and D. E. Wohlford, "System, methods and apparatus for program optimization for multi-threaded processor architectures," Apr. 2010.

[20] S. Pop, A. Cohen, C. Bastoul, S. Girbal, P. Jouvelot, G.-A. Silber, and N. Vasilache, "GRAPHITE: Loop optimizations based on the polyhedral model for GCC," in *Proc. of the 4th GCC Developper's Summit*, Ottawa, Canada, Jun. 2006, pp. 179–198.

[21] T. Grosser, A. Groesslinger, and C. Lengauer, "Pollyperforming polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.

[22] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model," in *International Conference on Compiler Construction (ETAPS CC)*, Apr. 2008.

[23] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 54:1–54:23, Jan. 2013.

[24] S. Verdoolaege, "isl: an integer set library for the polyhedral model," in *Proceedings of the Third international congress conference on Mathematical software (ICMS'10)*. ACM Press, 2010, pp. 299–302.

[25] B. Meister, D. Wohlford, J. Springer, M. Baskaran, N. Vasilache, R. Lethin, T. Benson, and D. Campbell, "Sane: an array language for sensor applications," in *Proceedings of a Workshop on Domain-Specific Languages and High- Level Frameworks for High Performance Computing, Salt Lake City, Utah, USA, November 16, 2012*, 2012.

[26] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, R. Dávid, S. V. Haastregt, A. Kravets, A. Lokhmotov, and E. Hajiyev, "PENCIL: a Platform-Neutral Compute Intermediate Language for Accelerator Programming," in *Parallel Architectures and Compilation Techniques (PACT)*, San Francisco, California, United States, 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01257236

[27] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: A cpu and gpu math compiler in python," 2011.

[28] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015. [Online]. Available: http://arxiv.org/abs/1512.01274