Portable Linear Algebra on FPGA using Data-Centric Parallel Programming

Manuel Burger, Johannes de Fine Licht, Torsten Hoefler
Department of Computer Science, ETH Zürich, Switzerland
{*burgerm, definelj, htor*}*@ethz.ch*

Abstract—This report presents an implementation of the Basic Linear Algebra Subprograms (BLAS) in the high-level parallel programming framework DaCe. The framework and the additions provided by this work allow to write high-level python linear algebra programs which can be translated into high performance HLS code and generate bitstreams for any Vitis-enabled device. The DaCe framework further allows to apply streaming compositions by transforming the dataflow of the input graph to achieve optimal performance on FPGAs. Our streaming kernels outperform any currently available CPU and are on a par with state of the art GPUs.

1 Introduction

This report presents a high-level approach to implement streaming linear algebra programs on Vitis-enabled FPGA accelerators. We use the Data-Centric Parallel Programming Framework (DaCe) [1], which provides us with a high-level graph API interface to build complete linear algebra programs based on their dataflow.

The DaCe framework features various graph primitives which allows to fully model a program's dataflow and state transitions. Amongst these graph primitives the most important for this work is the *library node*: these abstract high-level operators, which can be *lowered* into efficient implementation targeting different architectures, such as CPU, GPU, or FPGA. Using library nodes as operators, we can build the dataflow of our linear algebra programs as shown in Fig. 1.

The DaCe framework then takes this graph representation as an input to generate fast high-level synthesis C++ code, which is compiled to FPGA accelerators using the Vitis flow. The DaCe framework allows us to call the generated bitstream directly from high-level Python code and run it on a PCIe-attached FPGA accelerator. A demo of our work is available on github[1].

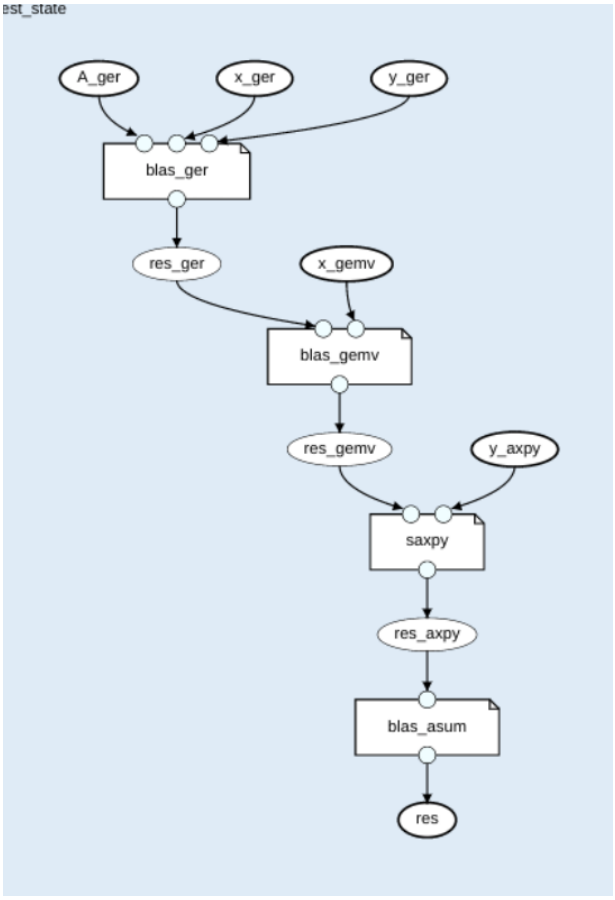

Fig. 1: Example graph of a linear algebra program with BLAS operators (rectangles) and memory containers (ovals).

2 High level BLAS operators for FPGA

To be able to build linear algebra programs such as shown in Fig. 1, we extended the DaCe framework with numerous BLAS operators library nodes, shown as rectangular nodes. The nodes can be expanded into "simple" DaCe dataflow implementations, which allows for a platform and target independent implementation. However, to further tailor the operator to the target device, the operators expose configurable properties such as vector widths and tile sizes. This allows a user to tune the operator to optimally trade off performance and FPGA resource utilization for the target device and the application requirements.

The implementations of all the provided BLAS operators have been optimized for best FPGA performance. We use the techniques described by de Fine Licht et al. [2] to enable pipelining and ensure a single cycle initialization interval of $II{=}1$ cycle, as well as maximize bandwidth utilization and leverage vector scaling capabilities. We use *full transposition* and *tiled accumulation interleaving* to optimize loop ordering and use tiling to avoid stalling pipelines due to loop-carried

1. https://github.com/manuelburger/daceBLAS_demo

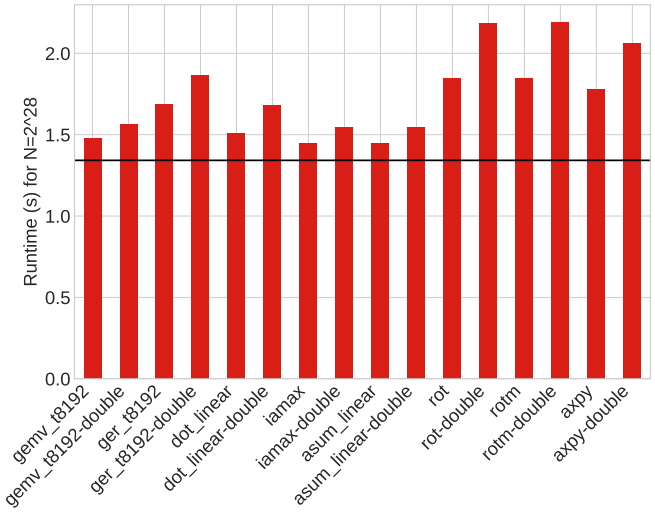

Fig. 2: Runtimes of BLAS operators for fixed input size on an Alveo U250 accelerator. The horizontal line indicates optimal performance.

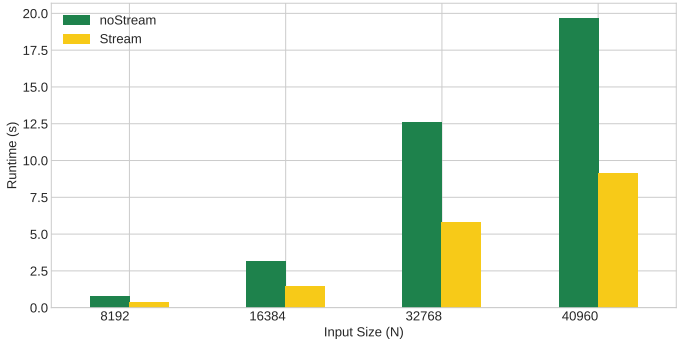

Fig. 3: Performance gains by using streaming composition benchmarked on the program visualized in Fig. 1. Green shows runtimes for before applying streaming composition, yellow after applying streaming composition.

dependencies. *Single-loop accumulation interleaving* allows us to split single loop reduction operations into two stages and remove the loop-carried dependency through interleaved reduction operations. We use *pipelined loop flattening and coalescing* to optimize filling and draining phases of the the pipelines. *Vectorization* is used to exploit more spatial parallelism and saturate the available memory bandwidth. *Buffering and data reuse* in on-chip buffers are used to reduce global memory accesses. *Memory oversubscription* is performed by reading in data into on-chip FIFO queues and then streaming the data to the operators from these queues allows for burst I/O read and write operation of larger contiguous chunks of memory to better saturate the memory bandwidth. *Memory banking* avoids conflicts between different I/O operations from/to different memory locations.

We build bitstreams with Vitis 2019.2, and execute the DaCe programs leveraging an Alveo U250 accelerator card with the XDMA 201803 DSA and 64 GB of DDR4 memory, using commit `2a5d30d` of this work's DaCe fork[2]. Fig. 2 shows runtime results of a range of implemented operators. The host programs are compiled with GCC 7.5.0 on Ubuntu 18.04 LTS, targeting a Cascade Lake Xeon 6234. The horizontal line indicates the target runtime of a perfect pipeline and one can clearly see the close to optimal runtimes of the implemented operators.

3 Streaming Compositions

To further improve the performance of complete linear algebra programs as seen in Fig. 1 on FPGA, we introduce *streaming compositions*, as introduced by FBLAS [3], where the authors shows significant improvements in performance for chains of linear algebra operators on Intel FPGAs. This is achieved by removing intermediate global memory buffers in the chain and instead directly streaming the computed elements from the output of one operator to the input of the next operator.

We implement automatic streaming composition for Xilinx FPGAs using the DaCe graph API. Having the program's dataflow represented as a graph allows us to perform an analysis on the graph and find intermediate global memory buffers, which can be removed and replaced by streams (realized as fast on-chip FIFOs).

Streaming composition are achieved using graph transformations (a tool provided by the DaCe framework), which allows us to modify the structure of the dataflow graph without changing the program semantics. The graph transformation removes the global memory nodes and replaces them with the on-chip streams.

Fig. 3 shows the performance benefits of streaming composition for the program presented in Fig. 1. For the benchmarked program the runtime is about halved or even better. This is because most of the work in the program is in the two BLAS level 2 operators `GER` and `GEMV`, which operate on a 2D matrix input. By applying streaming composition to the program, these operators are fused into one single pipeline. The removal of global memory accesses essentially increases the operational intensity of the program.

3.1 Streaming deadlock detection

A deadlock on a program after applying streaming composition can occur if there is a fork in the dataflow graph which is later again joined together leading to an undirected cycle on the dataflow graph. If the two paths from the forking to the joining node have different latencies, then the program could deadlock.

Having the dataflow available as a graph also allows us to perform a streaming deadlock analysis on the graph and prevent the streaming composition of a program which would deadlock when streaming or adjust FIFO buffer sizes to compensate for a constant offset in latencies on the two paths.

4 Cross-Platform Benchmarks

The DaCe framework allows us to provide different implementations for the same operator, and configure the operator with the right implementation to be used upon code

2. https://github.com/manuelburger/dace

```
M1 = np.outer(x, y1) + A  # GER
M2 = np.outer(x, y2) + A  # GER
M3 = np.outer(x, y3) + A  # GER
M4 = np.outer(x, y4) + A  # GER
M5 = np.outer(x, y5) + A  # GER
M6 = np.outer(x, y6) + A  # GER
M7 = np.outer(x, y7) + A  # GER
M8 = np.outer(x, y8) + A  # GER

vector1 = M1 @ x  # GEMV
vector2 = M2 @ x  # GEMV
vector3 = M3 @ x  # GEMV
vector4 = M4 @ x  # GEMV
vector5 = M5 @ x  # GEMV
vector6 = M6 @ x  # GEMV
vector7 = M7 @ x  # GEMV
vector8 = M8 @ x  # GEMV

vector1 = vector1 + vector2  # AXPY
vector3 = vector3 + vector4  # AXPY
vector5 = vector5 + vector6  # AXPY
vector7 = vector7 + vector8  # AXPY

vector1 = vector1 + vector3  # AXPY
vector5 = vector5 + vector7  # AXPY

buf = vector1 + vector5  # AXPY
result = np.sum(buf)  # ASUM
```

Listing 1: Python Numpy syntax for a larger synthetic benchmark program.

generation. As part of this work, we implement support for other architectures, such that we can emit CPU and GPU code from the same DaCe graph. To evaluate performance across different architectures, we use the synthetic program shown in Lst. 1 in single precision.

We run FPGA benchmarks on the system presented in Sec. 2, where we have aggressively performed the presented streaming compositions. The benchmark results for the Nvidia V100 GPU running the cuBLAS implementation were collected on a dual-socket AMD EPYC 7742 machine, using GCC 7.4.0 and Cuda 10.1 on CentOS 7. OpenBLAS results were obtained on a similar dual-socket EPYC 7742 machine, with quad-channel DDR4 memory at 3200 MHz, using OpenBLAS 0.3.9.

The results are presented in Fig. 4. Streaming composition increases the operational intensity of the program, such that the FPGA implementation running on the U250 accelerator performs as fast as the CPU implementation for the smallest inputs, and outperforms the CPU implementation running the highly optimized OpenBLAS code for all larger inputs, even including the time it takes to copy data to and from the accelerator.

Even compared to the latest state of the art Nvidia V100 GPU the FPGA BLAS implementation of this work performs faster for smaller inputs, and is on par with the GPU for larger inputs. This because of the increased operational intensity of the FPGA implementation, due to the streaming composition and kernel launch overhead of the many cuBLAS calls for the GPU. The GPU is additionally limited in available memory size as the V100 only features 32GB of on device memory. Therefore the program cannot run without further modifications for larger input sizes.

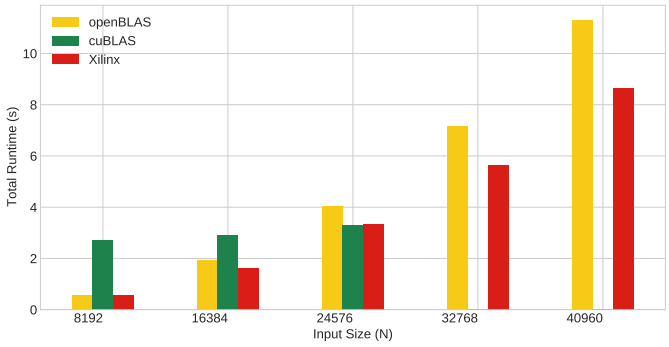

Fig. 4: Runtime comparison of the program shown in Lst. 1 across CPU, GPU and FPGA platforms. Accelerator runtime includes device-to-host and host-to-device copies.

A repository is provided to easily run the cross-platform benchmarks presented here:
https://github.com/manuelburger/daceBLAS_demo.

5 Conclusion

To conclude, we have shown that we can create high performance linear algebra operators for Xilinx FPGAs from high-level python code by using the DaCe parallel programming framework and its graph based API. We implemented streaming composition to further increase the performance of complete linear algebra programs running on Xilinx FPGAs. Finally we demonstrated that these implementations outperform a state of the art CPU BLAS library running on a powerful 128 core CPU. On top of that our implementations depending on input size can outperform or perform on par with Nvidia's fastest Tesla V100 GPU.

Acknowledgments

I would like to thank my supervisors Johannes de Fine Licht and Prof. Torsten Hoefler of the Scalable Parallel Computing Lab (SPCL) at ETH Zürich for their support. I would also thank the Systems Group at ETH Zürich for granting access to their FPGA development resources, as well as the Swiss National Supercomputing Centre and SPCL for access to their compute resources to run on different FPGA, CPU, and GPU architectures.

References

[1] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, 2019.

[2] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of High-Level Synthesis Codes for High-Performance Computing," *CoRR*, vol. abs/1805.08288, May 2018.

[3] T. D. Matteis, J. de Fine Licht, and T. Hoefler, "FBLAS: Streaming Linear Algebra on FPGA," *CoRR*, vol. abs/1907.07929, Jul. 2019.