FBLAS: Streaming Linear Algebra Kernels on FPGA

Tiziano De Matteis, Johannes de Fine Licht and Torsten Hoefler
Department of Computer Science, ETH Zurich, Switzerland
{tdematt, definelicht, htor}@inf.ethz.ch

***Abstract*—Reconfigurable hardware represents an attractive alternative to load-store architectures, as it allows eliminating expensive control and data movement overheads in computations. In practice, these devices are often not considered in the high-performance computing community, due to the steep learning curve and low productivity of hardware design, and the lack of available library support for fundamental operations. We present FBLAS, an open source implementation of Basic Linear Algebra Subroutines (BLAS) for FPGAs. The library is implemented with a modern HLS tool to promote productivity, reusability, and maintainability. Numerical routines are designed to be easily composed exploiting on-chip connections, to reduce off-chip communication resulting in lower communication volume.**

I. Introduction

The end of Dennard scaling [1] and Moore's law [2] has exhibited the limitations of traditional *Load-Store Architectures* (LSA), where data movement has come to dominate both energy and performance. To eliminate this overhead, we must employ architectures that are driven by data movement itself, and design *static dataflow architectures* that are specialized to the target application. FPGAs allow prototyping and exploiting application specific circuits by laying out fast-memory, interconnect, and computational logic according to the dataflow of the application. By avoiding unnecessary lookups and control logic, this yields higher energy efficiency than traditional LSAs. With the massive parallelism offered by these devices, along with the introduction of high-bandwidth memory (e.g., Xilinx Alveo U280) and native floating point units (e.g., Intel Stratix 10), peak floating point and memory performance allows them to be competitive on HPC workloads [3], [4].

Despite the promise of massive spatial parallelism, FPGAs are rarely considered for HPC systems and applications due to the steep learning curve and low productivity of hardware design. With the introduction of *High-Level Synthesis* (HLS) tools, programming hardware has become more accessible, but optimizing for these architectures is still a cumbersome task [5]. Furthermore, even with access to HLS, development of HPC codes is further hampered by the lack of maintained and publicly available high-level *libraries*, requiring most components to be implemented from scratch

We present FBLAS, an open source implementation of the *Basic Linear Algebra Subroutines* (BLAS) for FPGAs. FBLAS is implemented with HLS, enabling reusability, maintainability, and portability across FPGAs, and easy integration with existing software and hardware codes. Numerical module interfaces are designed to natively support streaming communication across on-chip connections, allowing them to be composed avoiding costly reads from off-chip memory, providing significant performance improvements for I/O bound computations by reducing the total communication volume. In addition, FBLAS provides a standard BLAS-compliant host-side interface, allowing numerical routines to be offloaded to FPGAs directly from the host system without writing hardware code. With the methodologies used to design FBLAS, we hope to set a precedent for FPGA library design, and contribute to the toolbox of customizable hardware components that is necessary for HPC codes to start productively targeting reconfigurable platforms.

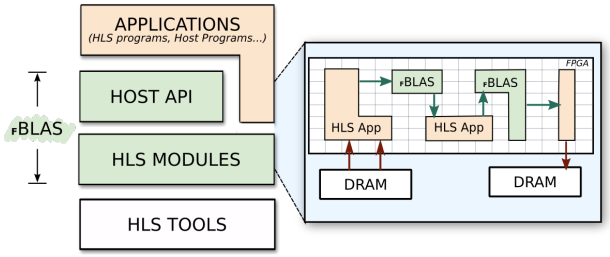

Fig. 1: Overview of the FBLAS [1]library.

II. The FBLAS Library

FBLAS exposes two layers of functionality to the programmer (Fig. 1): HLS *modules*, produced by a provided *code generator*, can be integrated into existing hardware designs; and a high-level host API conform to the classical BLAS interface. This distinction facilitates two ways of interacting with the library, depending on the use-case and level of abstraction required. Currently, FBLAS targets Intel devices, and it is implemented with the Intel FPGA SDK for OpenCL [6] tool.

A. HLS modules

HLS modules are independent computational entities that implement a library function (i.e., a routine), and have precise behaviour and interface. The *HLS programmer* can integrate HLS modules into her own HLS code: they can be invoked as functions, or composed in a streaming setting. Thanks to the massive spatial parallelism available, different modules can execute in parallel, and we enable modules to exchange data using direct on-chip resources, rather than resorting to DRAM. In FBLAS, modules implement BLAS routines (`DOT`, `GEMV`, `GEMM`, etc.). Modules have been designed with compute performance in mind, exploiting the spatial parallelism and fast on-chip memory on FPGAs. They have a *streaming interface*: data is received and produced through FIFO buffers.

[1]An initial release of FBLAS is available at: https://github.com/spcl/FBLAS

B. Host API

The Host API allows the user to invoke routines directly from the host program. The API is written in C++, and provides a set of library calls that match the classical BLAS calls in terms of signature and behavior. Following the standard OpenCL programming flow, the host programmer is responsible to transferring data to and from the device, can invoke the desired FBLAS routines working on the FPGA memory, then copy back the result from the device.

C. Code Generator

HLS modules in isolation work on streaming interfaces, which must be integrated with consumers and producers. Connections to/from DRAM, require dedicated modules to interact with off-chip memory. To produce the HLS modules required by both the high-level and low-level API, FBLAS provides a template-based *code generator*, that produces synthesizable OpenCL kernels. If the data is stored in DRAM, helper kernels must be created to read and inject data to the modules, and to write results back to memory. The code generator accepts a *routines specification file*, which is a JSON file provided by the programmer, specifying the routines that she wants to invoke. The programmer can customize *functional* parameters (e.g., if a routine accepts a transposed or non-transposed matrix) and *non-functional* parameters (e.g., vectorization widths and tile sizes). The code generator will produce a set of OpenCL files that can be used as input to the HLS compiler to synthesize the bitstream for reprogramming the FPGA.

III. Module Design

FBLAS modules come pre-optimized with key HLS transformations, but are configurable to allow the user to specialize them according to desired performance or utilization requirements. This is facilitated by tweaking the parameters given to the employed HLS transformations, decribed below.

A. Applied HLS Optimizations

To optimize FBLAS modules, we employ a set of FPGA-targeted optimizations [5], divided into three classes.

1) Pipeline-enabling Transformations: Pipelining is crucial for efficient hardware design. For a loop, this implies an *Initiation Interval* (*II*) of 1, meaning that a new loop iteration is started every clock cycle. We apply *iteration space transposition*, *loop strip-mining*, and *accumulation interleaving* to resolve *loop-carried dependencies* and *hardware resource contention* (usually to memory blocks), which can prevent the tool from scheduling the loop with an II of 1.

2) Replication: Parallelism on FPGA is obtained by replicating compute units. We achieve this by *unrolling* loop nests. Loop unrolling is applied by strip-mining the computations to expose unrolling opportunities for parallelism. We define the *vectorization width* W, which is used as the unrolling factor for inner loops. As this directly affects the generated hardware, it must be a compile-time constant.

3) Tiling: In FBLAS, tiling is used for Level 2 and Level 3 routines, where there is opportunity for data reuse. Tiling is implemented by strip-mining loops and reordering them to the desired reuse pattern, and explicitly instantiating fast on-chip buffers for the reused data. Tile sizes must be defined at compile-time, as they affect the number of memory blocks instantiated to hold the data.

Since FBLAS modules are implemented with streaming interfaces, tiling routines has implications for how data must be sent to, from, and between modules. In particular, matrices are tiled in 2D, where both the tile elements and the order of tiles can be scheduled by rows or by columns. This results in 4 possible modes of streaming a matrix. For this reason, FBLAS routines must take into account that data may be streamed in different ways and the module streaming interface specifies how input data is received and how output data is produced.

B. Systolic Implementation of `GEMM`

For the `GEMM` routine, an implementation based on unrolled and tiled loops, could result in designs characterized by high fan-in/fan-out, that prevent the module scalability. Therefore, we organized the computation using a 2D systolic array [7], such as the one shown in Fig. 2. A grid of $P_R \times P_C$ processing

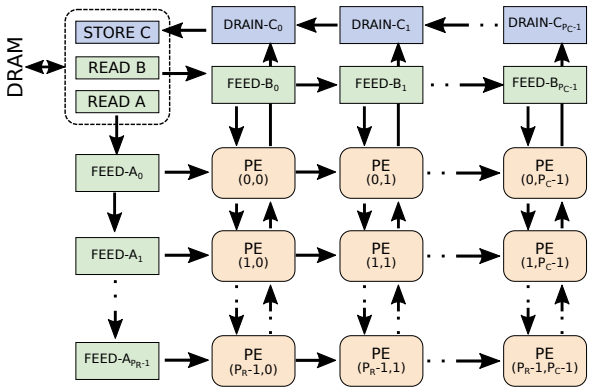

Fig. 2: `GEMM` systolic implementation

elements (PEs) is in charge of computing a tile of size $T_R \times T_C$ of the result matrix C. T_R and T_C are multiples of P_R and P_C respectively, and, therefore, each PE is responsible for evaluating $T_R T_C / P_R P_C$ elements of the C tile.

Input elements are read from DRAM (by *Read A* and *Read B* helper kernels) and forwarded by a set of *feeders*, according to the used tiling schema, to the first row and column of PEs. Then, each column of PEs forwards the elements of A, and each row of PEs forwards the elements of B, in a pipelined fashion. On each clock cycle, a PE receives one element of A and one element of B, multiplies them and accumulates over the resulting element of C. When the result is complete, each PE sends its contributions to a set of *drainers*. The *store C* kernel collects the results and writes them into memory.

IV. Streaming composition

Numerical computations may involve two or more modules that share or reuse data. The streaming interface introduced in

Sec. II-A enables modules to communicate through on-chip memory, allowing to reduce costly off-chip memory accesses, and to pipeline parallel execution of different modules.

We model a computation as a *module directed acyclic graph* (MDAG), in which vertices are hardware modules, and edges represent data streamed between modules. Source and sink vertices (circles) are *interface modules*, responsible for off-chip memory accesses. Other nodes (rectangles) are *computational modules*, e.g., FBLAS routines. Edges are implemented with FIFO buffers of a finite size. The number of elements consumed and produced at the inputs and outputs of a node is defined by the FBLAS routine interface. Stalls occur when a module is blocked because its output channel is full or an input channel is empty. We consider an MDAG to be *valid* if it expresses a composition that will terminate, i.e., it does not stall forever. Additionally, an edge in the MDAG between two modules is *valid* if *1)* the number of elements produced is identical to the number of elements consumed; and *2)* the order in which elements are consumed corresponds to order in which they are produced. Tiling schemes must be compatible, i.e., tiles must have the same size and must be streamed in the same way between consecutive modules.

In the following, we will show two common module compositions patterns that can be found in the updated set of BLAS subprograms introduced by Blackford et al. [8]. These, can be implemented by using two or more BLAS calls, and are utilized in various numerical applications. We will study the feasibility and benefits of a streaming implementation compared to executing the composed BLAS-functions sequentially. We distinguish between two cases: *1)* the MDAG is a multitree: that is, there is at most one path between any pair of vertices, and *2)* all other MDAGs.

A. Composition of multitrees

Generally, *a multi-tree module composition, with valid edges, is always valid*. Consider, for example, `AXPYDOT`, which computes $z = w - \alpha v$ and $\beta = z^T u$, where w, v, and u are vectors of length N. To implement this computation with BLAS, we need a `COPY`, an `AXPY`, and a `DOT` routine. The number of memory I/O operations (reads/write from memory) necessary to compute the result is then equal to $2N + 3N + 2N = 7N$. We can exploit module composition by chaining the `AXPY` and the `DOT` modules: the output of `AXPY` (z), will be directly streamed to the `DOT` module (see Fig. 3). This also allows omitting the first copy of w.

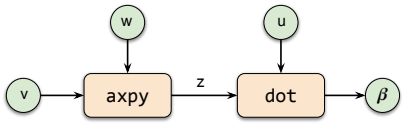

Fig. 3: `AXPYDOT` streaming implementation.

The number of I/O operations is then reduce to $3N + 1$. In addition, the `AXPY` and `DOT` modules are executed in parallel, reducing the number of cycles to completion from $C_{\text{sequential}} = (L_{\text{copy}} + N) + (L_{\texttt{dot}} + N) + (L_{\texttt{axpy}} + N)$ to just $L_{\text{copy}} + L_{\texttt{axpy}} + L_{\texttt{dot}} + N$. If N is sufficiently large, the computation time is reduced from $3N$ to just N.

B. Composition of non-multitrees

If the MDAG is not a multitree (i.e., there is more than one path between two nodes in the graph), invalid graphs can occur. For example, `GEMVER` computes $B = A + u_1 v_1^T + u_2 v_2^T$, $x = \beta B^T y + z$, and $w = \alpha B x$, where α and β are two scalars, A and B are $N \times N$ matrices, and $u_1, u_2, v_1, v_2, x, y, z$, and w are vectors of length N. With classic BLAS, this requires calling two `GER`, two `GEMV` and two copies. In a streaming implementation, the computation of B can be realized using a linear sequence of two `GER` calls. Then B is used for the computation of x and w. A streaming implementation would result in the two `GEMV` modules sharing the output of the `GER` computations, with the first `GEMV` streaming the result x to the second one. Given that replaying data is not allowed between two computational modules (condition 1. of being a valid edge), the first `GEMV`, must receive B in tiles by columns. In this way, it produces a block of results only after it receives an entire column of tiles of B, i.e., NT_M elements, where T_M is the width of a tile. Therefore, the composition would stall forever, unless the channel between the `GER` module and the second `GEMV` has a size $\geq NT_M$. Unless N is known a priori, this quantity is not fixed at compilation time, and the composition is invalid.

Invalid MDAGs can be handled by the user by either *a)* setting the channel size appropriately (according to the size of input data) or *b)* breaking the MDAG into multiple valid components, communicating through DRAM in between. In this example, although this prevents a full streaming implementation, we can resort to multiple sequential multitree streaming composition (Fig. 4).

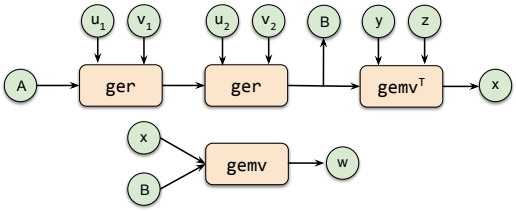

Fig. 4: `GEMVER`: a possible streaming implementation.

The first component streams between the two `GER` calls and one `GEMV` call, producing B and x and storing them in DRAM. After this component has terminated, B and x are present in DRAM, and the final `GEMV` can be executed. For this composition, the number of I/O operations is reduced from $8N^2 + 10N \approx 8N^2$ to $3N^2 + 9N \approx 3N^2$, and number of cycles to completion is reduced from $5N^2 + N$ to $2N^2$.

V. Evaluation

FBLAS implements all level-1 routines, and all generic level-2/-3 routines (`GEMV`, `TRSV`, `GER`, `SYR`, `SYR2`, `GEMM`, `SYRK`, `SYR2K`, and `TRSM`), for a total of 22 routines in single and double precision. To evaluate FBLAS, we show

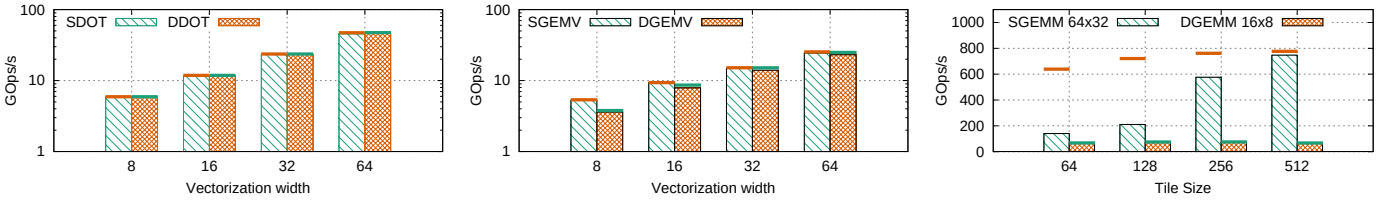

Fig. 5: Performance of modules implementing DOT, GEMV, and GEMM. Horizontal bars indicate expected performance.

the scaling of a representative set of single HLS modules, and the benefits of streaming composition. Experiments have been performed on a Nallatech 520N card, equipped with an Intel Stratix 10 GX 2800 FPGA. For synthesizing FPGA kernels, we use the Intel FPGA SDK for OpenCL v19.1.

A. Individual Module Evaluation

To evaluate the impact of vectorization and tiling on the performance of individual FBLAS modules, we considered modules that implement the DOT, GEMV and GEMM routines, as representative samples of BLAS Level 1, 2, and 3, respectively. Input data is generated directly on the FPGA, to test the scaling behavior of the memory bound applications DOT and GEMV. Averaged computation times have been considered for producing the reported performance figures. Performance is reported in floating point operations per second (Ops/s) based on the averaged execution time. Expected performance is computed by taking the number of used DSPs and multiplying by the frequency of the synthesized designed.

Fig. 5 reports the result of the evaluation. Left and middle plots, show the evaluation for the DOT and GEMV modules that operate on single and double precision, with a vectorization width spans from 8 to 64. For DOT the input data size is fixed at 100M elements. For GEMV we used square tiles of size 1024×1024, and a matrix of size 8192×8192. For both testbeds, synthesized designs are able to achieve the expected performance, implying that the instantiated compute is running at full throughput. Fig. 5 (right) shows the results obtained for GEMM with matrices 4096×4096 and different squared tile sizes. In the GEMM module we use a systolic implementation (see Sec. III-B), with an array of size 64×32 (single precision) and 16×8 (double precision). These are the highest values for which the compiler is able to generate the design without failing placement or routing. Smaller systolic arrays achieved expected performance already with small tile sizes. With larger designs, by increasing the tiles size we were able to approach the expected performance.

B. Streaming Composition Evaluation

We used the applications discussed in Sec. IV to evaluate the performance gains achieved by module composition. The streaming compositions are compared to calling the modules one-by-one via the host layer. For all the modules we fixed the vectorization width to 16 and, when used, tiles of size 1024×1024. Fig. 6 reports the *speedups* obtained with different input data sizes, computed as the ratio between the execution times of the host layer version over the streaming composition.

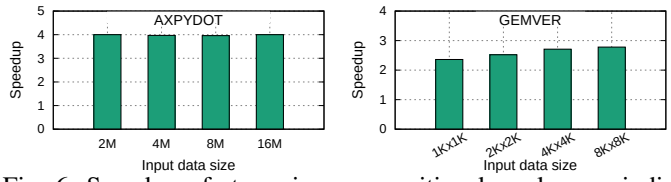

Fig. 6: Speedup of streaming composition kernels over individual kernels.

According to the analysis done in Sec. IV-A, for AXPYDOT we expected a speedup of 3. However, given that the vector z used by the AXPY routine is read/written in the same memory module, this results in a slow-down of the module, that does not affect the streaming version. This increases the achieved speedup to 4. Speedups for GEMVER confirm the analysis performed in Sec. IV-B. These experiments validate our performance analysis and highlight the performance benefits of pipelining computational modules using on-chip FIFO buffers.

VI. Conclusion

We presented FBLAS, the first publicly available BLAS implementation for FPGA. FBLAS is realized by using HLS tools, and allows programmers to offload numerical routines to the FPGA directly from a host program, or to integrate specialized FBLAS modules into other HLS codes. HLS modules expose *streaming interfaces*, enabling pipelined composition by exploiting on-chip data movement. By releasing the code as open source, we hope to involve the community in the continued development of FBLAS, targeting both current and future OpenCL-compatible devices

References

[1] H. Esmaeilzadeh et al., "Dark Silicon and the End of Multicore Scaling", IEEE Micro vol. 32, pp. 122–134,2012.
[2] M.M. Waldrop, "The chips are down for Moore's law", Nature News vol. 530, p. 144, 2016.
[3] E. Nurvitadhi et al. "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?", In Proceedings of FPGA 2017, pp. 5–14,2017.
[4] M. Duncan et al. "A Customizable Matrix Multiplication Framework for the Intel HARPv2 Xeon+FPGA Platform: A Deep Learning Case Study", In Proceedings of FPGA 2018, pp. 107–116, 2018.
[5] J. de Fine Licht et al., "Transformations of High-Level Synthesis Codes for High-Performance Computing", In CoRR vol. abs/1805.08288, 2018.
[6] Intel Corp., "Intel FPGA SDK For OpenCL", 2019.
[7] H.T. Kung et al., "Systolic Arrays for VLSI", CMU-CS, 1978.
[8] S. Blackford et al. "An Updated Set of Basic Linear Algebra Subprograms (BLAS)", ACM Trans. Math. Softw. vol. 28, pp. 135–151, 2002.