FBLAS: Streaming Linear Algebra on FPGA

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

***Abstract*—Energy efficiency is one of the primary concerns when designing large scale computing systems. This makes reconfigurable hardware 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. With the introduction of high-level synthesis (HLS) tools, programming hardware has become more accessible, but optimizing for these architectures requires factoring in new transformations and trade-offs between hardware resources and computational performance. We present FBLAS, an open source implementation of BLAS for FPGAs. FBLAS is implemented with HLS, enabling reusability, maintainability, and portability across FPGAs, and easy integration with existing software and hardware codes. By using the work-depth model, we capture the space/time trade-off of designing linear algebra circuits, allowing modules to be optimized within performance or resource constraints. Module interfaces are designed to natively support streaming communication across on-chip connections, allowing them to be composed to reduce off-chip communication. 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.**

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. In these systems, more than 90% of the energy consumed by a floating point instruction is spent on register files, cache, and control logic [3]. 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. While some of the efficiency of FPGAs is counteracted by the fine granularity of components used to implement general purpose logic, recent FPGAs have started shipping with native floating point units (e.g., Intel Stratix 10), offsetting the penalty of small components with large, specialized units. With the massive parallelism offered by these devices, along with the introduction of high-bandwidth memory (e.g., Xilinx Alveo U280), peak floating point and memory performance allows them to be competitive on HPC workloads [4], [5].

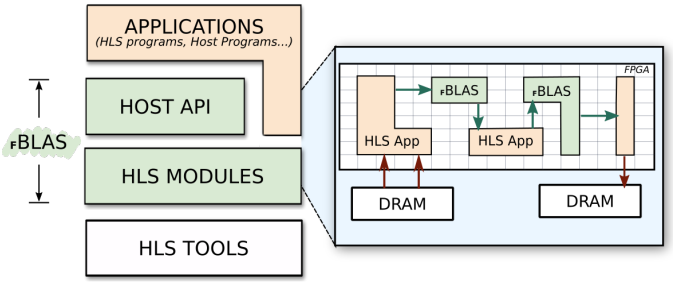


Fig. 1: Overview of the FBLAS [1]library.

Despite the promise of massive spatial parallelism, FPGAs are rarely considered for HPC systems and applications. The main obstacles preventing wider adoption are the traditionally low-level programming required, subsequent low productivity, and lack portability between vendors and generations of boards. *High-level synthesis* (HLS) tools have ushered in a new wave of interest from the community, where tools such as the Intel FPGA SDK for OpenCL [6] and Xilinx Vivado HLS [7] enable the programmers to use high-level languages when targeting FPGAs, using familiar languages such as C, C++, or OpenCL to synthesize hardware circuits. Although this reduces the design and development cycle of FPGA programs, a new set of coding and optimization techniques must be employed that take the underlying hardware into account [8]. Generally, optimizing for FPGAs is more challenging than for LSA architectures, as *the addition of space utilization as a metric for code transformations and optimizations leads to a* ***space/time trade-off***, which must be considered by the programmer. Even with access to HLS, development of HPC codes is hampered further by the lack of maintained and publicly available high-level *libraries*, requiring most components to be implemented from scratch.

With a programming methodology that captures the complexity of FPGA programming, and the release of fundamental libraries to avoid tedious duplicate work, reconfigurable hardware might yet have a future in HPC systems. Our work addresses both directions.

We present FBLAS, a flexible and customizable implementation of the Basic Linear Algebra Subroutines (BLAS) for FPGAs. The library is implemented with a modern HLS tool to promote productivity, reusability, and maintainability. FBLAS enables the rapid development of numerical computation that targets reconfigurable architectures, giving the HLS programmer access to a set of customizable routines that can be re-

[1]FBLAS is open sourced at: https://github.com/spcl/FBLAS

used, composed, and integrated with other program logic. 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. The contributions of this paper are:

- FBLAS, the first portable and *open source* BLAS implementation on FPGA, realized entirely with state-of-the-art HLS tools;
- a characterization of HLS routines by the key characteristics that affect the performance, resource usage, and memory bandwidth consumption of the resulting design;
- models to enable the user to choose desirable combinations of parameters to optimize performance and/or resource usage of her design; and
- guidelines for composing routines that communicate through on-chip resources, avoiding costly reads from off-chip memory, providing significant performance improvements for I/O bound computations, and reducing the total communication volume.

Additionally, we offer insights obtained from developing FBLAS that we believe generalize to the development of future FPGA libraries. Although we focus on FPGAs as a platform for implementation specialized hardware in this work, the principles discussed generalize to any platform capable of implementing static dataflow architectures, such as coarse-grained reconfigurable array or ASICs.

II. Background

To utilize the massive spatial parallelism offered by FPGAs we must exploit **pipelining** of computations. In contrast to CPUs and GPUs, where parallelism is achieved by saturating multiple wide SIMD units, FPGAs primarily rely on deep pipelines to achieve *pipeline parallelism* in a "multiple instruction, single data"-fashion. This is true both within single compute kernels, such as individual BLAS routines, and between kernels, where data can be **streamed** to achieve more reuse and increase pipeline parallelism. Multiple aspects of FBLAS must be considered in this context

FPGA Layout. FPGAs are the most widely available implementation of *reconfigurable hardware*, allowing custom architectures to be implemented in terms of a fixed resource budget of logic, buffer, and interconnect components. The resources of contemporary FPGAs can be divided into three categories: *General purpose logic units,* consisting of *lookup tables* (LUTs) implementing combinatorial logic, and *flip-flops* (FFs) implementing registers, which are already sufficient to implement arbitrary functions; *Hardened arithmetic units*, such as *digital signal processing* units (DSPs), used to accelerate common arithmetic operations; and *Memory blocks,* specialized for bulk storage, exposing limited access ports (as opposed to registers), but large capacity – examples include block and Ultra RAM on Xilinx FPGAs, and M20K blocks on Intel FPGAs. The latter are our primary tool for data reuse, and will be the focus of communication avoiding optimizations. When considering raw throughput, we are primarily occupied with general purpose logic and arithmetic units, as these make up the computational circuits. However, for algorithms that expose a high potential for data reuse, memory blocks become significant for shifting algorithms into the compute bound domain by forming deep pipelines to increase pipeline parallelism. Targeting FPGA resources requires the program specified by the user to be mapped to concrete hardware components, and for these components to be wired up correctly (the *placement and routing* problem). Finding an *optimal* mapping is intractable, and real-world implementations rely on heuristics to find and optimize mappings [9]. If a large number of connections are required to/from the same location on the chip (fan-in and fan-out, respectively), or if the full resource capacity of the FPGA is approached, the automatic placement and routing process often fails to find good mappings. This can result in reduced clock rates, or failure to map the design altogether. Mitigating this often requires manual user invention that exploits domain knowledge.

High-level Synthesis Programming. HLS tools allow programmers to define FPGA architectures using a high-level language, typically C, C++ or OpenCL, which is transformed to hardware description languages such as Verilog or VHDL, which can be synthesized to hardware circuits. The architectures produced by the source code is influenced using *pragmas*, used to specify code transformations (see Sec. IV-A). The relationship between resource usage and performance and I/O is thus handled directly from the HLS code. Distinct computational kernels can exchange data either through off-chip memory (typically DRAM), or on-chip resources, such as registers and memory blocks. To express on-chip communication, HLS tools provide the *FIFO* abstraction (also referred to as *channels* or *streams*). FIFOs are typed and bounded single-producer/single-consumer queues implemented in hardware, and provide synchronized data exchange between end-points.

The BLAS Library. The *Basic Linear Algebra Subprograms* (BLAS) [10] are established as the standard dense linear algebra routines used in HPC programs. BLAS routines are organized into three levels: Level 1 BLAS routines offer scalar-vector and vector-vector operations with linear computational complexity; Level 2 deals with matrix-vector operations of squared complexity; and Level 3 exposes matrix-matrix operations with cubic complexity. Level 1 and 2 are generally *memory bound*, while operations in Level 3 are *computation bound*, which translate to different opportunities for parallelism and data reuse: compute bound kernels allow us to increase pipeline parallelism through internal reuse, whereas memory bound routines offer limited potential for increasing compute performance individually. However, some BLAS routines have matching rates of consuming input and producing output, allowing us to increase pipeline parallelism and reducing I/O by streaming between them (treated in Sec. VI).

We design the FBLAS library to expose BLAS routines in a way that will naturally expose opportunities for pipeline parallelism both within the library, and when used as blocks in other codes, using the tools offered by the HLS abstraction to specialize kernels to exploit the resources offered by the

hardware.

III. The fBLAS Library

fBLAS exposes two layers of functionality to the programmer (see Fig. 1): HLS *modules*, produced by a provided *code generator*, which can be integrated into existing hardware designs; and a high-level host API conforming to the classical BLAS interface. This distinction is a key contrast to software libraries, as it facilitates two ways of interacting with the library, depending on the use-case and level of abstraction required.

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. Modularity helps the development and maintainability of the software, and facilitates efficient exploitation of FPGA architecture: 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. In addition, HLS modules can be characterized by a precise performance and space/time model (see Sec. V), to optimize them for performance and/or resource consumption.

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. Library calls can be *synchronous* (return when the computation is done) or *asynchronous* (return immediately).

C. Code Generator

HLS modules in isolation work on streaming interfaces, which must be integrated with consumers and producers. Often these must be connected to DRAM, requiring 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* and *non-functional* parameters of the routine. Functional parameters affect the logic of the routine, by specifying, for example, if a Level 2 routine accepts a transposed or non-transposed matrix. Non-functional parameters are optional, and characterize the performances and resource occupation. They regulate 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.

IV. 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 (see Sec. V). This is facilitated by tweaking the parameters given to the employed HLS transformations, decribed below. This can in turn affect the streaming behavior of kernels depending on the tiling strategy employed, as detailed in Sec. IV-B.

A. Applied HLS Optimizations

While HLS tools reduce the complexity of FPGA programming, writing optimized code that results in efficient hardware design is still a challenging task. Traditional program optimizations are insufficient, as they do not consider pipelining and spatial replication of hardware circuits [8]. To optimize fBLAS circuits, we employ a set of FPGA-targeted optimizations, divided into three classes.

1) Pipeline-enabling Transformations: Achieving perfect pipelining is crucial for efficient hardware design. For a loop, this implies an *Initiation Interval* ("*II*", or just *I*) of 1, meaning that a new loop iteration is started every clock cycle. To allow this, the programmer must 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. To overcome these issues, we apply *iteration space transposition*, *loop strip-mining*, and *accumulation interleaving* [8]. This is particularly relevant when targeting double precision, as the target FPGA used in this work does not support double precision accumulation natively, requiring a two-stage circuit to fully pipeline.

2) Replication: Parallelism on FPGA is achieved by replication compute units, either "horizontally" (SIMD-style vectorization) or "vertically" (parallelism through data reuse). 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: Loop tiling is a well-known code optimization used to increase both spatial and temporal locality of computations. In HLS, we use the transformation to organize loop schedules such that reused data fits into fast on-chip memory,

saving costly accesses to DRAM. 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 buffers for the reused data using C-arrays. Tile sizes must be defined at compile-time, as they affect the number of memory blocks instantiated to hold the data.

Because FBLAS modules are implemented with streaming interfaces, tiling routines has implications for how data must be sent to, from, and between modules. To avoid mismatches, this must be treated explicitly by the framework.

B. Impact of Tiling on Streaming Kernels

The module streaming interface specifies how input data is received and how output data is produced. BLAS routines accept three classes of input data: *scalars*; *vectors*; and *matrices*. Scalars are passed once when invoking a module, while vectors and matrices are streamed between modules. As vectors are only tiled along a single dimension, the tile size, and optionally the number of repetitions, are the only interface parameters. Matrices, on the other hand, 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 across a matrix interface. We choose to adopt a 2D tiling schema also for Level 2 routines as this *i)* open the possibility to have different I/O complexities for the same routine, *ii)* favors module composition both between routines in the same BLAS Level (see Sec. VI) and across different Levels.

FBLAS routines must take into account that data may be streamed in different ways. Consider the GEMV routine that computes $y = \alpha Ax + \beta y$, where A is an $N \times M$ matrix and x and y are M and N elements vectors, and A is optionally transposed. If A is not transposed, the routine receives A by tiles (by rows or by columns), x and y, and pushes results to the output stream. This can implemented in two possible ways: In the first case (exemplified in Fig. 2), A is received in tiles of size $T_N \times T_M$, in a row-major fashion. For each tile of A, a range of x (of size T_M) is used to update a block of y (of size T_N). An entire row of tiles (i.e., MT_N elements) is needed to compute a block of y. This tiling scheme achieves reuse over y, but requires receiving the x-vector $\lceil N/T_N \rceil$ times. We say that this implementation requires that vector x is *replayed*. The number of I/O operations for this computation is $NM + MN/T_N + 2N$, where only the vertical tile size contributes to reducing the overall I/O, as it reduces the number of times x must be replayed.

Another possibility could be to stream the tiles of A by columns (see Fig. 3). With this solution, we use an entire column of tiles (i.e., NT_M elements) and a block of x to update all the blocks of y. The resulting y is produced only at the end of the computation. In this case, y must be replayed: since each block is updated multiple times, we need to output it and re-read it $\lceil M/T_M \rceil$ times. The number of I/O operations for this configuration is $NM + M + 2NM/T_M$, where T_M is now the primary factor affecting overall I/O.

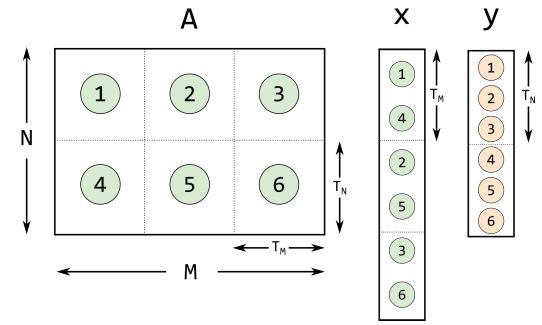


Fig. 2: GEMV where A is received in tiles by rows. Numbers indicates arrival order, green is read, orange is read/write.

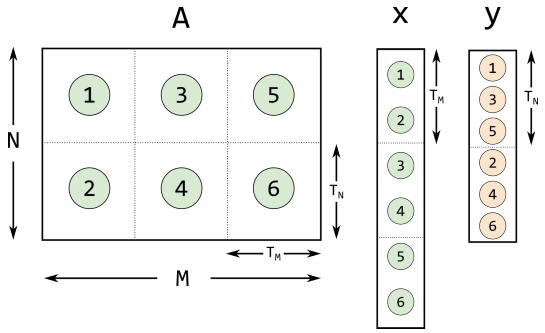


Fig. 3: GEMV where A is received in tiles by columns.

This example shows that different ways of streaming input data may result in different way of computing the result. Handling them in a single implementation would lead to the generation of large designs due to the presence of multiples branch in the control flow. Therefore, we offer different module specializations implementing the same routine. Each version handles a specific scheme affecting the order of input and output elements, and varies in I/O complexity. While the specialization with lowest I/O can be straightforwardly determined for a single module, additional constraints on feasible specializations can be introduced when integrating with existing work, or composing with other kernels (see Sec. VI).

V. Space and time trade-offs

Performance and space utilization are the two main metrics that must be considered while optimizing code for reconfigurable hardware. This *space/time* trade-off, i.e, the compromise between resource consumption and performance, must be understood to optimize the performance and/or resource usage of the resulting design. In FBLAS, HLS modules implement numerical routines by means of fully pipelined nested loops. Inner loops perform the actual processing: they are unrolled and synthesized as circuits that implement the desired computation. Outer loops are derived from tiling, and they can be considered as the *schedule* in which computations are performed (i.e., to order in which operands are sent to the circuit). Vectorization widths and tile sizes represent the two knobs on which a programmer can act to change module

performance and used resources. Having a greater unrolling factor increases the performances at the expense of higher *computational resource* consumption (i.e., LUTs, FFs, and DSPs). Similiarly, bigger tiles reduces the communication volume at the expense of higher *memory resource* usage (i.e., memory blocks).

To capture the space/time trade-off, we introduce models to analyze the interplay between parallelism, resource consumption, and performance of an FPGA design. For this discussion, we consider a single module, assuming that input data is always available in its input channels. Code is shown as written using a generic HLS tool, in which pragmas apply to the *following* loop, and FIFO buffers are channels accessible through *pop* and *push* operations.

A. Modeling the computational circuit

We model the computational resource consumption and performance of a circuit by using the *work and depth* model. The *work and depth* model is a popular methodology used to analyze the running time of parallel algorithms independent of the execution platform [11]. The cost of an algorithm is determined by considering the *work*, i.e., the total number of operations that are performed in a computation, and the *depth*, i.e., the length of the longest shortest path from any input to any output. In the following, we will refer to these quantities as the *application work* ($\mathcal{A}_W$) and *application depth* ($\mathcal{A}_D$). The application depth represents the minimum time needed to perform the computation with a sufficient number of resources. We introduce the additional concepts of *circuit work* ($\mathcal{C}_W$) and *circuit depth* ($\mathcal{C}_D$), to analyze the circuit implementing the inner loop of the module (where the computation is actually performed). The circuit work is linked to the number of resources required to implement the computation in hardware, while the circuit depth represents the *latency* of the circuit.

In FBLAS HLS modules, computations performed in the inner loops can be viewed either as a *map* (loop iterations work on different data) or as a *map-reduce* (intermediate results are accumulated) computation. Modules that implement routines such as SCAL, AXPY, GER, or SYR fall in the first category; modules that implement DOT, GEMV, TRSV, or GEMM belong to the second. In the following, we focus the analysis on SCAL and DOT, as representative for each of the two cases.

The SCAL routine takes an input vector of N elements and scales them by using a user-defined factor. Since each operation is independent from the others, the application work and depth are $\mathcal{A}_W = N$ and $\mathcal{A}_D = 1$. An excerpt of the HLS implementation code is shown in Listing 1, where *alpha* is the scaling factor. The computation loop has been strip-mined with a factor W, which is the vectorization width.

For the circuit work and depth analysis, we consider the body of the outer loop (Lines 3-7). At each iteration, it performs the scaling of W distinct input elements, implemented through the unrolled inner loop. Fig. 4 shows the computation performed with $W = 4$. Nodes represent operations. The circuit work $\mathcal{C}_W$ is equal to the vectorization width W, while the circuit depth $\mathcal{C}_D$ is equal to 1. In general, the number

```
1 void scal(float alpha, int N, chan ch_x, chan ch_out){
2     for(int it=0; it<N/W; it++){
3         #pragma unroll
4         for(int i=0;i<W;i++){
5             x[i]=alpha*pop(ch_x);
6             push(ch_out,x[i]);
7         }
8 }}
```

Listing 1: SCAL implementation: data is received/sent using channels ch_x/ch_out.

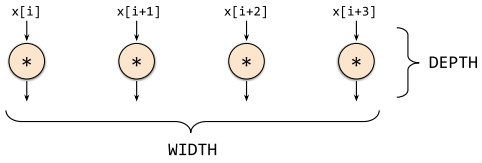


Fig. 4: SCAL: circuit work and depth analysis

of cycles C required to execute a pipeline with latency L, initiation interval I, and taking M elements as input, is:

$$C = L + IM$$

Given that in FBLAS, all modules have $I = 1$, and the latency is the circuit depth, this becomes $C = \mathcal{C}_D + M$, where M is the number of iterations of the inner loop. Therefore, for SCAL, we will have $C = 1 + N/W$: if we increase the vectorization width we will linearly reduce the number of loop iterations, and consequently reduce the time to completion.

The DOT routine performs the dot product between two N element vectors. An efficient implementation could be realized using a binary tree structure, resulting in an application work and depth of $\mathcal{A}_W = 2N - 1$ and $\mathcal{A}_D = \log_2(2N)$. The HLS implementation code is shown in Listing 2, where x and y are the two vectors received from two channels, and W is the vectorization width. If synthesized for the Intel Arria 10 or Stratix 10 FPGAs, the two loops will have $I = 1$, due to native support for single precision floating point accumulation in hardware. Fig. 5 shows the computation performed by the body of the outer loop (Line 6-10) with $W = 4$. Edges are data dependencies between operations. In this case, both circuit work and depth depend on the vectorization width, $\mathcal{C}_W = 2W$ and $\mathcal{C}_D = 2 + \log_2(W)$. This results in a computation times of $C = 2 + \log_2(W) + N/W$ cycles. In this case, if we increase

```
1  void dot(int N, chan ch_x, chan ch_y) {
2      float res=0;
3      for(int it=0; it<N/W; it++){
4          float acc=0;
5          #pragma unroll
6          for(int i=0;i<W;i++){
7              x[i]=pop(ch_x);
8              y[i]=pop(ch_y);
9              acc+=x[i]*y[i];
10         }
11         res+=acc;
12     }
13 push(ch_res,res);}
```

Listing 2: DOT implementation: data is read from channels ch_y and ch_y, result is sent to channel ch_res.

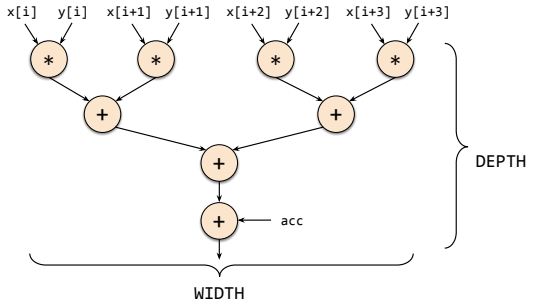


Fig. 5: Circuit work/depth analysis of DOT inner loop.

the vectorization width, we will linearly reduce the number of loop iterations and we will increase only logarithmically the depth $\mathcal{C}_D$.

To show how the circuit work and depth capture the characteristics of the circuit, we synthesized the two discussed modules. Table I reports empirical computational resource consumption and circuit latency obtained by varying the vectorization width. These figures are obtained from the Intel FPGA Offline Compiler targeting an Intel Stratix 10 GX 2800 FPGA. The DSPs of this FPGA are able to start one addition and one multiplication per clock cycle. For SCAL, the reported numbers exactly match our expectations. Computational resources (i.e. LUTs, FFs and DSPs) linearly increase with respect to the vectorization width, while latency remains constant. In particular we have that $LUT = 49\mathcal{C}_W$, $FF = 96\mathcal{C}_W$, and $DSP = \mathcal{C}_W$ For DOT, resource consumption also grows proportionally with respect to the width, and the latency increases linearly when the vectorization width is doubled. The compiler introduced some optimization in laying down the circuit design, but we can still see a linear relation between circuit work and computational resources, resulting in $LUT \simeq 17\mathcal{C}_W$, $FF \simeq 40\mathcal{C}_W$, and $DSP = \mathcal{C}_W/2$. These empirical results show how the work and depth model can be used to correlate used resources and achieved computation times.

	SCAL				DOT			
W	LUTs	FFs	DSPs	Lat	LUTs	FFs	DSPs	Lat
2	98	192	2	78	96	192	2	85
4	196	384	4	78	160	320	4	88
8	392	768	8	78	288	640	8	92
16	784	1,536	16	78	544	1,280	16	96
32	1,568	3,072	32	78	1,056	2,560	32	100
64	3,136	6,144	64	78	2,112	5,120	64	106

TABLE I: Resource consumption and latency.

B. Modeling Memory Resources

Modules that implement Level 2 and Level 3 routines use tiling to reduce the communication volume. Tiling is expressed by outer loops. To model memory resources we have to take into account the applied tiling and how the computation circuit accesses memory.

Consider the case of the tiled version of GEMV that works on A and receives the matrix in tiles by row. From Sec. **??** we know that the communication volume for this computation is $NM + MN/T_N + 2N$. The HLS implementation is shown in Listing 3. The code exploits reuse over x and y. To implement

```
1  void gemv (int N, int M, float alpha, float beta,
2             chan ch_x, chan ch_y, chan ch_A){
3    float local_y[TILE_N], local_x[TILE_M];
4    for(int ti=0;ti<N/TILE_N;ti++){
5      for(int tj=0;tj<M/TILE_M;tj++){
6        for(int i=0;i<TILE_N;i++){
7          float acc_o=0, acc_i=0, prev;
8          for(int jj=0;jj<TILE_M/W;jj++){
9            if(tj==0 && jj==0) prev=beta*pop(ch_y);
10           if(tj!=0) prev=local_y[i];
11           if(i==0)          //receive x
12             #pragma unroll
13             for(int j=0;j<W;j++)
14               local_x[jj*W+j]=pop(ch_x);
15           #pragma unroll
16           for(int j=0;j<W;j++) //compute
17             acc_i+=pop(ch_A)*local_x[jj*W+j];
18           acc_o+=alpha*acc_i;
19         }
20         local_y[i] = prev+acc_o; acc_i=0;
21         if(tj==BlocksX-1) push(ch_out,local_y[i]);
22       }
23 } } }
```

Listing 3: Implementation of GEMV: TILE_N and TILE_M are the tile sizes (i.e., T_N and T_M, respectively). ch_x and ch_y are used for vectors, ch_A for the input matrix.

the memory area for storing a block of x (local_x in the code) and y (local_y) we need a memory buffer of size S bytes, sufficient to store store T_N and T_M single precision numbers, respectively. This memory can be seen as a 2D area, characterized by a *memory width* $\mathcal{M}_W$ and a *memory depth* $\mathcal{M}_D$. The width is given by the number of bytes that must be read or written in a single cycle in the circuit. The depth is given by the ratio $S/\mathcal{M}_W$. In the computational circuit (lines 7-23), a single element of local_y is read and written, while W elements of local_x are accessed. The number of memory blocks that are required to implement such memory areas depends on the architecture. Assuming that a memory block has 1 read and 1 write port, a given capacity of R bytes and and port width of P bits, we have that the number of blocks B is equal to $\lceil 8\mathcal{M}_W/P \rceil \lceil \mathcal{M}_D/R \rceil$.

Table II shows the number of memory blocks that are required to implement the GEMV on an Intel Stratix 10. The FPGA has memory blocks (M20K) of 20K bits and a port width of 40 bits. These numbers match the expected number of blocks B.

	Tiles: 256×256			Tiles: 1024×1024			Tiles: 4096×4096		
Var	W:4	W:32	W:128	W:4	W:32	W:128	W:4	W:32	W:128
x	4	26	103	4	26	103	8	26	103
y	1	1	1	2	2	2	7	7	7

TABLE II: Memory block used in GEMV to implement buffers for x and y for different tile sizes and widths (W).

C. Pareto Optimal Configurations

The models introduced in this section allow the user to optimize the resulting design according to her needs, such as lowest computation time within a computational resource

budget. In this case, the circuit work and depth is analyzed. Fig. 6a shows how circuit work (DSPs) and computation time changes with different vectorization width in the DOT module working on a vector with 1 K elements. All the points except one are on the Pareto frontier. Indeed, the cases with vectorization width 1024 and 512 have equal computation time, but the former has a higher circuit work (i.e., usage of DSPs).

Similarly, the programmer could be interested in restricting the number of used memory blocks, or in limiting the communication volume. The plot in Fig. 6b shows the trade-off between these two metrics for different tile sizes in the GEMV example. In this case, the matrix is $8K \times 8K$. For this relation, all points are on the Pareto frontier. However, it should be noticed that for high value of the tile sizes, the benefit of the reduced communication volume with respect to the number of used memory blocks is negligible.

VI. Streaming composition

Numerical computations may involve two or more modules that share or reuse data. The input required for one module may be produced from another module, or two modules may accept the same input data. When the order in which such data is produced and consumed is the same, the streaming interface introduced in Sec. III-A enables modules to communicate through on-chip memory, rather than through off-chip DRAM. This has two key advantages: *1)* it reduces costly off-chip memory accesses, as data is streamed directly by the producer module to the consumer module, rather than storing and loading it to DRAM; and *2)* it allows pipeline parallel execution of different modules that are configured simultaneously on the FPGA. Avoiding off-chip communication is key for I/O bound computations, such as BLAS Level 1 and Level 2 routines. In this section we analyze the benefit of streaming linear algebra using FBLAS routines.

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*, that are 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 configuration (e.g., GEMV in Sec. **??**). 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 module A and B 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.

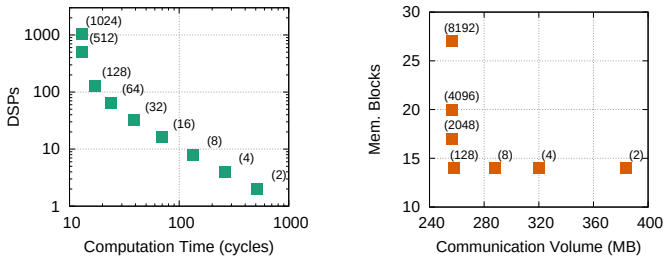
For example, if B is the GEMV module discussed in Sec. **??**, and A produces the input vector x, we can compose them only if B operates on a matrix received in tiles by columns, as it will otherwise need to replay the reading of x, thus violating (1) above. In general, tiling schemes must be compatible, i.e., tiles must have the same size and must be streamed in the same way between consecutive modules.

In the following, we will evaluate different module compositions patterns that can be found in real computations. To this end, we target the updated set of BLAS subprograms introduced by Blackford et al. [10]. These routines 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 based on these examples. 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

This simplest streaming FBLAS composition is a linear sequence of compute modules, where each module receives data from one or more interface modules, and (at most) one other computational module. 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 copy is needed because AXPY overwrites the input vector y).

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. 7). This also allows omitting the first copy of w. The number of I/O operations is then equal to $3N+1$, the minimal



(a) DOT (b) GEMV

Fig. 6: Trade-offs between (a) computation time and used DSPs in DOT with varying W, (b) communication volume and memory blocks in GEMV by varying (square) tile sizes.

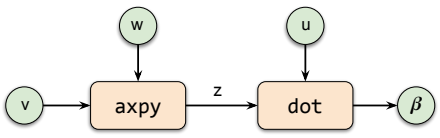


Fig. 7: AXPYDOT streaming implementation.

number required to perform the computation. 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 $L_{\text{copy}} + L_{\texttt{axpy}} + L_{\texttt{dot}} + N$, under the assumption that all memory interfaces are fully saturated during execution (where N is adjusted for the vectorization width W). If N is sufficiently large, the computation time is reduced from $3N$ to just N. Such a composition will always be valid, assuming all edges are independently valid.

In many cases, the output of a computational or interface module is shared between two (or more) computational modules. Consider the BICG computation, used in the *biconjugate gradient stabilized method*. Given matrix A of size $N \times M$, BICG computes $q = Ap$ and $s = A^T r$, where p and s are vectors of size M, and q and r are vectors of size N. The computation is implemented with two independent GEMV routines, that can be executed in parallel. Both routines read A, but with different access patterns. Using a streaming composition we can read A only once (see Fig. 8), assuming that the two GEMV modules accept data *streamed in the same way*. Although one GEMV expects A^T, we can achieve the same access pattern by setting their schedule accordingly through tiling patterns. The two modules compute in parallel and the results are sent to interface modules that write them in memory. In this case, we reduce the number of I/O operations related to the matrix A from $2NM$ to NM, but do not affect the number of cycles to completion NM under the assumption of fully saturated memory ports.

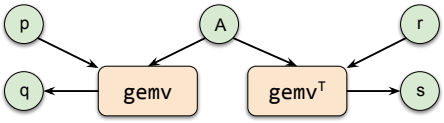


Fig. 8: BICG streaming implementation

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. Consider the case of ATAX, that computes $y = A^T Ax$, where A is an $M \times N$ matrix, x and y are vectors of size N. A streaming implementation similar to the previous example is shown in Fig. 9. In this case, the two computational modules

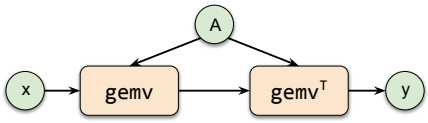


Fig. 9: ATAX: invalid streaming implementation.

share one interface module and the first GEMV module streams its results to the second one. Given that replaying data is not allowed between two computational modules, the right GEMV, must receive A in tiles by rows. However, the left GEMV produces a block of results only after it receives an entire row of tiles of A, i.e., NT_N elements. Therefore, the composition would stall forever, unless the channel between the A interface module and the second GEMV has a size $\geq NT_N$. Unless N is known a priori, this quantity is not fixed at compilation time, and the composition is invalid. In general, a similar situation occurs in all the cases in which, given two vertices of the MDAG v_i and v_j, there are at least two vertex-disjoint paths (except v_i and v_j) from v_i to v_j.

C. Complex compositions

In complex computations, we can compose modules in different ways. Choosing the most suitable implementation is critical for both validity and performance. For example, GEMVER computes $B = A + u_1 v_1^T + u_2 v_2^T$, $x = \beta B^T y + z$, and $w = \alpha Bx$,

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. This leads to a non-multitree composition similar to the one of ATAX, the two GEMV routines read B and B^T, respectively, while one streams to the other, which we know to be an invalid configuration for dynamic values of N. Although this prevents a full streaming implementation, we can resort to multiple sequential multitree streaming composition (Fig. 10). The first component streams

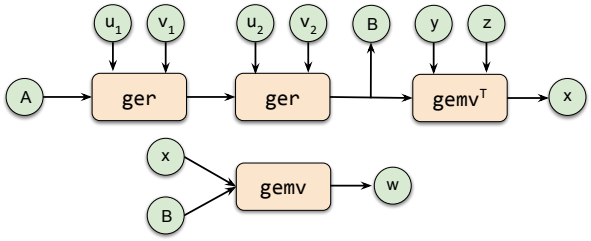


Fig. 10: GEMVER: a possible streaming implementation.

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 $2N^2$; a significant improvement, despite resorting to sequentializing the two components.

As a final example, we consider the *Conjugate Gradient* (CG) method. This iterative method can be implemented by using one matrix-vector product, three vector updates, and two inner products per iteration. A possible streaming implementation is shown in Fig. 11. Although all modules are connected in a streaming fashion, the two calls to DOT effectively sequentialize the implementation into three parts, since they need to receive all input data to produce the result. Going from

a sequence of BLAS operations to the streaming composition shown reduces the communication volume from N^2+14N to N^2+11N, and number of cycles to completion from N^2+5N to N^2+2N, which shows limit benefit from streaming for this application. Indeed, it becomes negligible for large N. We will evaluate the performance of this composition along will all other examples shown in this section to show the varying benefit of streaming composition with different scenarios.

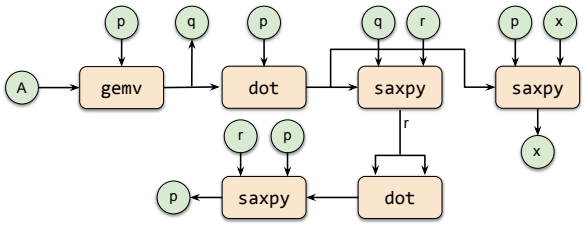


Fig. 11: A Conj. Gradient streaming implementation.

VII. 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 with single and double precision support. To evaluate FBLAS, we show the scaling of a representative set of single HLS modules, and the behavior and benefits of streaming composition. We also include comparison to a state-of-the-art BLAS implementation on CPU.

A. Experimental Setup

We performed experiments on two Nallatech boards equipped with two different FPGAs, described in Tab. III. In both cases, the card is connected to the host machine via a PCIe bus. The host has a 10 cores Intel Xeon E5-2630 v4, operating at 2.2GHz (no Hyper-Threading), and 64 GB of 4-channel DDR4 memory. For the Stratix testbed, we target an early access version of the *board support package* (BSP) firmware for Stratix 10 provided by Nallatech. Approximately 25% of the FPGA resources are reserved by the BSP.

FPGA	Board	ALUs	FFs	M20Ks	DSP	DRAM
Arria 10 GX 1150	510T	748 K	1.5 M	2.3 K	1444	2×8GB
Stratix 10 GX 2800	520N	1.8 M	3.7 M	11.7 K	5760	4×8GB

TABLE III: FPGA boards used for evaluation.

For synthesizing FPGA kernels, we use the Intel FPGA SDK for OpenCL v17.1 (Arria) and v18.0.1 (Stratix). In the Stratix FPGA, automatic memory interleaving is disabled (per advice of the vendor), data must be manually allocated to one of the DDR banks. The peak bandwidth of a single bank is 19.2 GB/s. All designs are compiled with the `-no-interleaving=default`, `-fp-relaxed`, and `-fpc` flags. CPU code is compiled using gcc 7.2, with optimization flag `-O3`, and we use Intel MKL 2018 (update 4), a highly tuned BLAS implementation specialized for Intel processors. For measuring power on the FPGA, we use the `aocl` utility provided by Intel, that reports the power drain of the entire board (not only the FPGA chip). For the Arria testbed, we removed the power consumed by unused FPGA. For CPU measurements, we use Mammut [12], and consider the power consumed by the processor and by the DRAM only.

B. Individual Module Evaluation

In this section, we evaluate the impact of vectorization and tiling on the performance of individual FBLAS modules. To capture different computational and communication complexities, we show 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, in order to test the scaling behavior of the memory bound applications `DOT` and `GEMV`.

Experiments were repeated 50 times and averaged computation times have been considered for producing the reported performance figures. In all cases the 99% confidence interval is within 5% of the measured mean. 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, implying maximum throughput of the circuit.

Fig. 12 (left) shows the evaluation for the `DOT` modules that operate on single and double precision. The vectorization width spans from 4 to 64, and the input data size is fixed at 100M elements. For both testbeds, synthesized designs are able to achieve the expected performance, implying that the instantiated compute is running at full throughput. The evaluation for `GEMV` is shown in Fig. 12 (middle). In this case, we used square tiles of size 2048×2048. The running frequencies differ slightly between designs with the same vectorization width, but different precision. For the Stratix testbed, measured performance starts to be lower than expected for larger vectorization width (up to 35% lower). Finally, Fig. 12 (right) shows the results obtained for `GEMM` with matrices 2048×2048. In `GEMM` module we exploit both horizontal and vertical replication (see Sec. IV-A2). Due to the different number of available resources, in the evaluation, we used a width of 16×16 (single precision) and 8×8 (double precision) for the Arria testbed, and a width of 32×16 (single precision) and 16×8 (double precision) for the Stratix FPGA. These are the highest values for which the compiler is able to generate the design without failing placement or routing. As this implementation is based on unrolling inner loops of `GEMM`, a different approach would be using a processing element/systolic array-based architecture. By increasing the tiles size we were able to approach the expected performance given by the number of compute units instantiated. The performance differences between the two architectures are due to different running frequencies and replication factors.

Table IV shows the used resources for modules with highest performance (i.e, width 64 for `DOT` and `GEMM`, biggest tiles size for `GEMM`), the frequency of the synthesized design, and

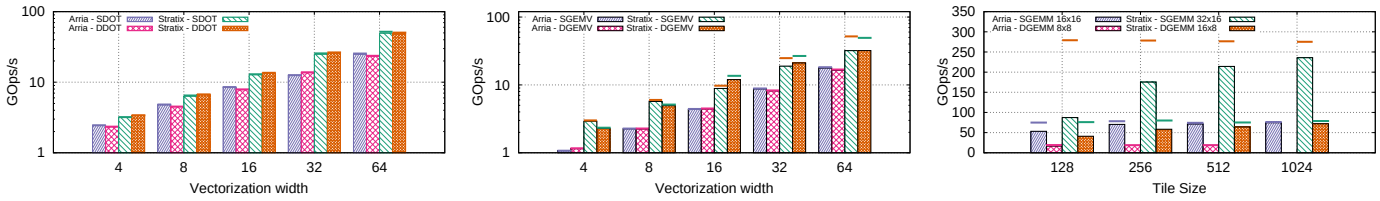


Fig. 12: Performance of modules implementing DOT, GEMV, and GEMM. Horizontal bars indicate expected performance.

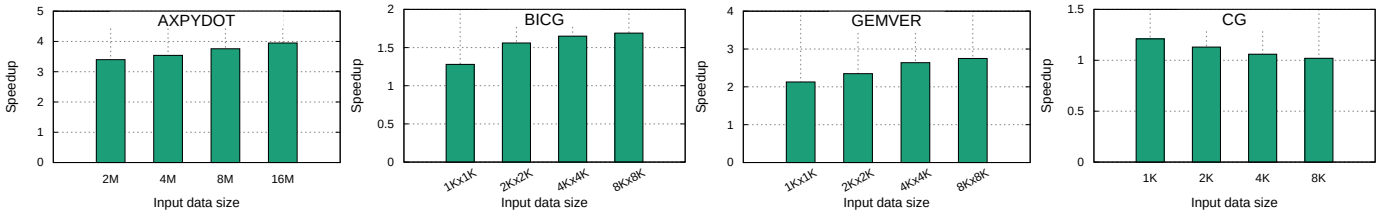


Fig. 13: Speedup of streaming composition kernels over individual kernels.

the power consumed. For the Stratix architecture, the table also includes MLAB consumption. These are shallow memory areas used for control logic, implemented with the *Adaptive Logic Module* (ALM) of the FPGA, so their number affects the global logic utilization. Double precision modules use 4 DSPs per operation, as well as more logic (one order of magnitude higher) to guarantee a loop initiation interval of one.

C. Streaming Composition Evaluation

We used the applications discussed in Sec. VI 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. Due to BSP limitation, designs have no automatic memory interleaving. For all the used modules we fixed the vectorization width to 16 and, when relevant, tiles of size 1024×1024. The used width is sufficient to saturate the memory bandwidth of the single DDR module. Fig. 13 reports the *speedups* obtained with different input data sizes on the Stratix testbed, computed as the ratio between the execution time of the host layer version over the execution time of the streaming composition. Similar results hold for the Arria testbed. According to the analysis done in Sec. VI-A, for AXPYDOT we expected a speedup of 3. However, given the limitations of the BSP, 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. For BICG, the expected performance gain is due to the reduced number of memory accesses that are performed by the streaming version. Considering the frequency of the synthesized design (290MHz), the interface module of the streaming version is able to saturate the 92% of the memory bandwidth of the module. This gives an expected speedup of 1.84. The measured speedup is at most 1.7. Speedups for GEMVER and the conjugate gradient (CG) confirm the analysis performed in Sec. VI-B.

Overall, these experiments validate our performance analysis and highlight the performance benefits of pipelining computational modules exploiting on-chip FIFO buffers, in particular in cases where the chained kernels can execute completely in parallel, yielding maximum pipeline parallelism. Additionally, thanks to the reduction of interface modules, module composition allows to use a comparable (e.g., in CG) or lower amount of resources (up to -40% in AXPYDOT) with respect to the non-streamed implementation.

		LUTs [K]	FFs [K]	M20Ks	DSPs	MLABs	F [MHz]	P [W]
ARRIA	SDOT	5.738 (0.8%)	4.296 (0.3%)	0 (0.0%)	65 (4.5%)	-	199	56.3
	DDOT	241.9 (32.3%)	121.5 (8.1%)	3 (0.1%)	256 (17.7%)	-	185	56.8
	SGEMV	10.38 (1.4%)	7.645 (0.5%)	58 (2.5%)	67 (4.6%)	-	142	55.6
	DGEMV	346.1 (46.3%)	219.7 (14.7%)	365 (15.6%)	264 (18.3%)	-	131	57.2
	SGEMM	27.62 (3.7%)	80.02 (5.3%)	1639 (70.0%)	272 (18.8%)	-	149	62.2
	DGEMM	225.7 (30.2%)	151.4 (10.1%)	205 (8.8%)	288 (19.9%)	-	147	62
STRATIX	SDOT	5.36 (0.3%)	15 (0.4%)	34 (0.3%)	65 (1.1%)	196 (51.7%)	405	58.5
	DDOT	138.1 (7.4%)	280.6 (7.5%)	4 (0.1%)	256 (4.4%)	367 (66.0%)	394	62.8
	SGEMV	7.2 (0.4%)	20.4 (0.5%)	119 (1.0%)	67 (1.2%)	167 (41.4%)	405	59.1
	DGEMV	166.6 (8.9%)	387.3 (10.4%)	755 (6.4%)	264 (4.6%)	1,640 (73.8%)	385	67.5
	SGEMM	58.3 (3.1%)	237.2 (6.4%)	1639 (14.0%)	544 (9.4%)	2,067 (56.6%)	269	61.5
	DGEMM	267.2 (14.3%)	505.9 (13.6%)	4,096 (34.9%)	576 (10.0%)	1,959 (68.9%)	307	67.8

TABLE IV: Resource consumption for the different modules. The percentages of maximum resources are in brackets.

D. Comparison with CPU

In this section we compare the performance of the Host CPU and of the Stratix FPGA in executing the single routines (Sec. VII-B) and the streaming applications (Sec. VII-C), using the single precision format. In this case, to overcome BSP limitation, FPGA implementations have been realized by manually interleaving the data across different memory modules. In CPU tests, we considered the best parallel execution time.

Tab. V reports the reports the execution time for the individual routines. FPGA designs have been compiled using a vectorization width of 64 for DOT, width 64 and squared tile of size 2048 for GEMV, and width 32×16, squared tile of size 1024 for GEMM. For the memory bound routines (DOT and GEMV), FBLAS is able to achieve an execution time comparable to the CPU version when using larger size of the input data, with an higher energy efficiency. For GEMM, the FBLAS implementation provides lower performance with respect to the CPU implementation. As pointed

Rout.	N	CPU		FPGA		
		Time [usec]	P [W]	Time [usec]	F [MHz]	P [W]
DOT	4M	146.9	97.5	713	213.3	54.8
	16M	2,050	97.9	2,484		
GEMV	2Kx2K	92.5	93.7	605	210	55.3
	8Kx8K	5,402	98.6	7,170		
GEMM	1Kx1K	2,972	103.6	13,411	198.5	61.5
	4Kx4K	168,890	105.3	831,403		

TABLE V: Comparison to CPU for single routines.

out in Sec. VII-B, different implementations could be used for compute bound routines. Thanks to its interface, FBLAS can be easily extended, favoring the explorations of different variants of the same routine.

Table VI reports the execution time for the streaming applications, with different input sizes. For the FPGA, we considered modules with vectorization width 32 and tiles size 2048×2048, with BICG as the only exception, being compiled with a width of 64. This allows it to exploit the memory bandwidth of the 4 DDR modules. FBLAS is able to obtain lower executions times for AXPYDOT and GEMVER, and slightly higher for BICG. The FPGA board generally uses $\sim 40\%$ less power for the measured workloads with respect to the CPU (we note that the reported power drain for FPGA consider the full board). Conjugate gradient does not gain from streaming composition due to its sequential nature. In general, the best fit for the FPGA are kernels that benefit from streaming composition, where the computation of many or all stages can be executed in a fully pipeline parallel fashion.

Appl.	N	CPU		FPGA		
		Time [usec]	**P** [W]	**Time** [usec]	**F** [MHz]	**P** [W]
AXPYDOT	4M	1,376	96.9	1,228	282.5	55.5
	16M	8,556	97.1	3,970		
BICG	2Kx2K	218	98.4	777	225	59
	8Kx8K	5,796	98.5	9,929		
GEMVER	2Kx2K	895	99.2	3,362	228.3	60.8
	8Kx8K	43,291	99.5	38,783		
CG	2Kx2K	6,771	98.8	175,850	228.3	59.6
	8Kx8K	477,396	99.1	1,918,180		

TABLE VI: Comparison to CPU for composed kernels.

VIII. Related work

There has been significant interest in implementing dense numerical routines for reconfigurable hardware. In most cases, hardware description languages are used. Zhuo and Prasanna [13] and Kestur et al. [14] propose implementations for several linear algebra operations, including dot product, matrix-vector multiplication, matrix multiplication, and matrix factorization. Work by Jovanovic and Milutinovic [15] addresses the implementation of a double precision matrix multiplication by using blocking. Moss et al. [5] present a matrix multiplication SystemVerilog template, able to achieve close to peak performance for GEMM on the target FPGA (800 GOPs/s on an Arria 10 1150). More recently, there has been a wider adoption of HLS tools for implementing linear algebra. D'Hollander [16] proposes a blocked matrix-matrix multiplication, in which block summation is done over the host CPU while the block multiplication is performed on FPGA. It achieved 4.77 GOPs/s on a Zynq XC7Z020. de Fine Licht et al. [8] apply code transformations to GEMM, by scaling up computational kernels until constrained by the resource on the device. This design is able to achieve 184.1 GOps/s on a Xilinx UltraScale KU115 FPGA. Yinger et al. [17] propose a matrix multiplication OpenCL template, based on a systolic array design. The implementation details are not discussed, but the design is reported to yield 790 GOps/s on an Intel Arria 10 1150. In all the aforementioned cases, code is not made available to users. Furthermore, these works address one or a few numerical routines, and does not treat composition between routines. In contrast, FBLAS is open source, offers the full set of BLAS routines, and exposes native streaming composition to the user.

Some previous work focuses on *design space exploration* (DSE), where HLS programmers are assisted to find good combinations of pipelining, vectorization, initiation interval and memory usage, to achieve a given resource/performance goal. DSE tools are based on analytical models coupled with static analysis [18], estimated provided by HLS tools [19], or machine learning methods [20]. Usually these tools require code instrumentation and output hints to drive the programmer optimizations. Similar to this work, Zhuo and Prasanna [13] analyze the design trade-off between used resources and performances. In FBLAS, HLS modules can be tuned by acting on only two aspects: vectorization and tiling. To guide this, we propose models to analyze the space/time trade-offs that arise when selecting vectorization widths and tile size.

Works spanning in different application domains have benefited from pipeline parallelism exploiting on-chip resources for streaming between modules [21], [22], [23]. In these works, applications are written to take advantage of pipelining, addressing the problems that arise from a streaming implementation. Vasiljevic et al. [24] propose a library for OpenCL streaming components, used to decouple kernel-kernel and memory-kernel communications. In FBLAS, all routines communicate via streaming interfaces, enabling the benefits of composing HLS modules.

IX. Conclusion

In this paper 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 are designed such that resource usage and performance is *tunable* according to a model of their *space/time trade-off*, allowing them to be specialized to the user's application; and expose *streaming interfaces*, enabling pipelined composition by exploiting on-chip data movement. These two aspects allow exploiting the pipeline parallelism offered by reconfigurable hardware, and are key considerations in the design of libraries for spatial architectures. 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.

ACKNOWLEDGMENTS

This work has been supported from the European Research Council (ERC) under the European Union's Horizon 2020 programme, Grant Agreement No. 678880 (DAPP), and Grant Agreement No. 801039 (EPiGRAM-HS). We also would like to thank the Swiss National Supercomputing Center (CSCS) for providing the computing resources and for their excellent technical support.

REFERENCES

[1] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *IEEE Micro*, vol. 32, no. 3, pp. 122–134, May 2012.

[2] M. M. Waldrop, "The chips are down for moore's law," *Nature News*, vol. 530, no. 7589, p. 144, 2016.

[3] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb 2014, pp. 10–14.

[4] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021740

[5] D. J. Moss, S. Krishnan, E. Nurvitadhi, P. Ratuszniak, C. Johnson, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. Leong, "A customizable matrix multiplication framework for the intel harpv2 xeon+fpga platform: A deep learning case study," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: ACM, 2018, pp. 107–116. [Online]. Available: http://doi.acm.org/10.1145/3174243.3174258

[6] Intel, "Intel fpga sdk for opencl," https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html, 2018.

[7] Xilinx, "Vivado hls," https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, 2018.

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

[9] S. Chen and Y. Chang, "Fpga placement and routing," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2017, pp. 914–921.

[10] S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and C. Whaley, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151, Jun. 2002.

[11] G. E. Blelloch and B. M. Maggs, "Algorithms and theory of computation handbook," M. J. Atallah and M. Blanton, Eds. Chapman & Hall/CRC, 2010, ch. Parallel Algorithms, pp. 25–25.

[12] D. D. Sensi, M. Torquati, and M. Danelutto, "Mammut: High-level management of system knobs and sensors," *SoftwareX*, vol. 6, pp. 150 – 154, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352711017300225

[13] L. Zhuo and V. K. Prasanna, "High-performance designs for linear algebra operations on reconfigurable hardware," *IEEE Transactions on Computers*, vol. 57, no. 8, pp. 1057–1071, Aug 2008.

[14] S. Kestur, J. D. Davis, and O. Williams, "Blas comparison on fpga, cpu and gpu," in *Proceedings of the 2010 IEEE Annual Symposium on VLSI*, ser. ISVLSI '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 288–293. [Online]. Available: http://dx.doi.org/10.1109/ISVLSI.2010.84

[15] Z. Jovanovic and V. Milutinovic, "Fpga accelerator for floating-point matrix multiplication," *IET Computers Digital Techniques*, vol. 6, no. 4, pp. 249–256, July 2012.

[16] E. H. D'Hollander, "High-level synthesis optimization for blocked floating-point matrix multiplication," *SIGARCH Comput. Archit. News*, vol. 44, no. 4, pp. 74–79, Jan. 2017. [Online]. Available: http://doi.acm.org/10.1145/3039902.3039916

[17] J. Yinger, E. Nurvitadhi, D. Capalija, A. Ling, D. Marr, S. Krishnan, D. Moss, and S. Subhaschandra, "Customizable fpga opencl matrix multiply design template for deep neural networks," in *2017 International Conference on Field Programmable Technology (ICFPT)*, Dec 2017, pp. 259–262.

[18] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: A high-level performance analysis tool for fpga-based accelerators," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2016, pp. 1–6.

[19] Y. Choi, P. Zhang, P. Li, and J. Cong, "Hlscope+,: Fast and accurate performance estimation for fpga hls," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2017, pp. 691–698.

[20] K. O'Neal, M. Liu, H. Tang, A. Kalantar, K. DeRenard, and P. Brisk, "Hlspredict: Cross platform performance prediction for fpga high-level synthesis," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2018, pp. 1–8.

[21] T. Kenter, J. Förstner, and C. Plessl, "Flexible fpga design for fdtd using opencl," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–7.

[22] Z. Wang, J. Paul, B. He, and W. Zhang, "Multikernel data partitioning with channel on opencl-based fpgas," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 6, pp. 1906–1918, June 2017.

[23] H. R. Zohouri, A. Podobas, and S. Matsuoka, "Combined spatial and temporal blocking for high-performance stencil computation on fpgas using opencl," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: ACM, 2018, pp. 153–162. [Online]. Available: http://doi.acm.org/10.1145/3174243.3174248

[24] J. Vasiljevic, R. Wittig, P. Schumacher, J. Fifield, F. M. Vallina, H. Styles, and P. Chow, "Opencl library of stream memory components targeting fpgas," in *2015 International Conference on Field Programmable Technology (FPT)*, Dec 2015, pp. 104–111.