Lab C Report - Vitis BLAS Library

110061638 呂政和 110061639 林致佑 110061641 鄧向凱

1. **Background introduction**

Basic Linear Algebra Subroutines (BLAS) is a specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication.

Although the BLAS specification is general, BLAS implementations are often optimized for speed on a particular machine, so using them can bring substantial performance benefits. BLAS implementations will take advantage of special floating point hardware such as vector registers or SIMD instructions.

As a result, Xilinx offers Vitis BLAS Library as a fast FPGA-accelerated implementation of the standard basic linear algebra subroutines which provides us the end-to-end (E2E) application acceleration.

s

Types	L1 primitives	L2 kernels	L3 software APIs
Leveraged by	FPGA hardware developers	Vitis host code developers	Pure software developers

L1 primitives:

L1 primitives' implementations are intended to be used by hardware developers to implement an application or algorithm specific FPGA logic in HLS which include computation and data mover modules. The computation modules always have **stream** interfaces. The data mover modules move data between vectors' and matrices' on-chip storage and the computation modules.

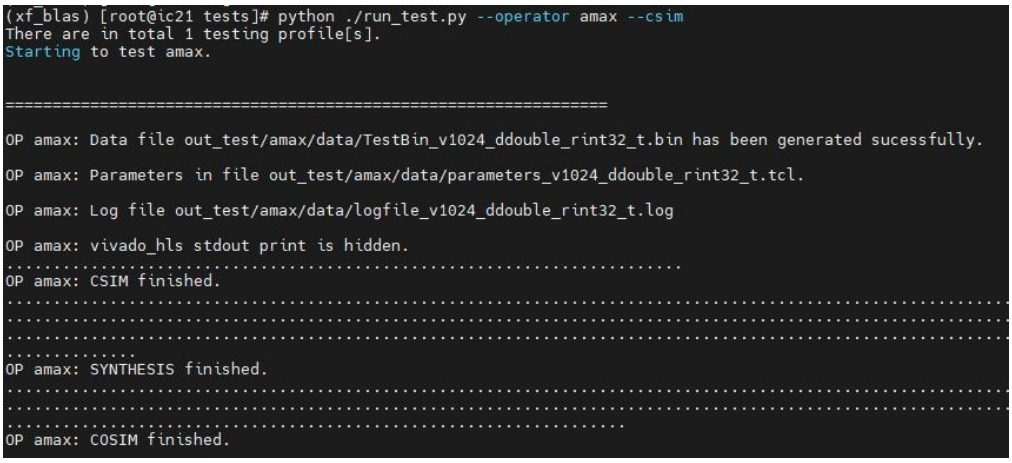


Fig 1 ：Result of Level 1 csim、csynth、cosim

L1 compute APIs:

1. amax(): returns the position of the vector element that has the maximum magnitude.

2. amin(): returns the position of the vector element that has the minimum magnitude.
3. asum(): returns the sum of the magnitude of vector elements.
4. axpy(): compute Y = alpha*X + Y.
5. copy(): compute Y = X
6. dot(): returns the dot product of vector x and y.
7. gbmv(): performs general banded matrix-vector multiplication matrix and a vector y = alpha * M * x + beta * y
8. gemv(): returns the result vector of the multiplication of a matrix and a vector y = alpha * M * x + beta * y
9. nrm2(): returns the Euclidean norm of the vector x.
10. scal(): compute X = alpha * X
11. swap(): swap vector x and y
12. symv(): returns the result vector of the multiplication of a symmetric matrix and a vector y = alpha * M * x + beta * y
13. trmv(): returns the result vector of the multiplication of a triangular matrix and a vector y = alpha * M * x + beta * y

L1 data mover:

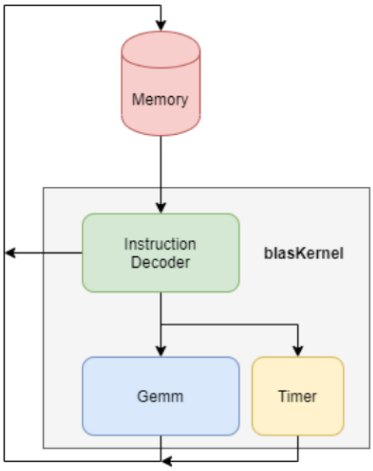
The L1 data mover modules are used to move matrix and vector data between their on-chip storage and the input/output streams of the computation modules. These data movers are intended to be used in conjunction with computation modules to form the HLS implementations for BLAS level 1 and 2 functions.

How to test L1 functionality?

All L1 primitive implementations have been tested **against numpy functions**. That is, a python based testing environment has been developed to generate random test inputs for each primitive and its corresponding function in numpy, compute the golden reference via the numpy function call, and finally compare the golden reference with the csim and cosim outputs of the primitive to verify the correctness of the implementation.

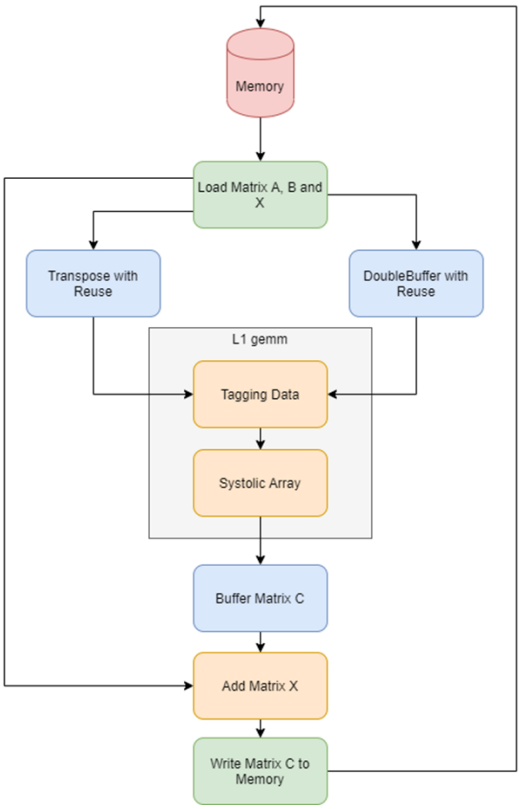
L2 Kerenls:

L2 Kernel includes datamovers and computation components composed by L1 primitive functions. Data from off-chip storage needs datamovers to transfer data into L1 function through stream interfaces. As shown in the figure below, the top function blasKernel is composed by instruction process unit, timer and operation functional unit e.g. GEMM. The functional unit can be a single blas function or more.When implementing Deep Learning, GEMM is also an important function.



GEMM Kernel:

GEMM kernel here is an implementation of the operation C = A * B + X, where A, B, X and C are matrices.There are three major parts in this kernel, including data movers, buffer and a systolic array. For systolic array, it is the main part to implement the matrix multiplication and do the partial sum; for data movers, it is responsible for moving data and results after computation. If we carefully arrange data movement, we won't need other registers for any temporary results. And what we must be careful with is the addresses. One wrong address may cause the function fails; for buffers, the reason we choose double buffers is that load in and load out speed of GEMM module is not same, we use double buffers to store other matrix and later transpose it.



L2 benchmark:

HW_emu(64 bit):

```
[root@ic21 memKernel]# ./build_dir.hw_emu.xilinx_u50_gen3x16_xdma_201920_3/gemm_bench.exe ./build_dir.hw_emu.xilinx_u50_gen3x16_xdma_201920_3/blas.xclbi
n ./build_dir.hw_emu.xilinx_u50_gen3x16_xdma_201920_3/config_info.dat 64 64 64 ./data/float/
Read custom sizes of matrix: (64, 64, 64)
Read custom data directory: ./data/float/
INFO: [HW-EMU 01] Hardware emulation runs simulation underneath. Using a large data set will result in long simulation times. It is recommended that a s
mall dataset is used for faster execution. The flow uses approximate models for Global memories and interconnect and hence the performance data generate
d is approximate.
configuring penguin scheduler mode
scheduler config ert(0), dataflow(1), slots(16), cudma(1), cuisr(0), cdma(0), cus(1)
xfblasCreate  21961.838285 msec
copyToFpga  24.701807 msec
copyFromFpga  136015.516622 msec
Api time is 136040.218429 msec
DATA_CSV:,Freq,M,K,N,TimeApiMs,EffApiPct,PerfApiTops
DATA_CSV:,200.000000,64,64,64,136040.218429,0.000004,0.000000
>> Kernel #0 << Test passed!
INFO: [HW-EMU 06-0] Waiting for the simulator process to exit
INFO: [HW-EMU 06-1] All the simulator processes exited successfully
```

HW(64 bit):

```
[root@ic21 memKernel]# ./build_dir.hw.xilinx_u50_gen3x16_xdma_201920_3/gemm_bench.exe ./build_dir.hw.xilinx_u50_gen3x16_xdma_201920_3/blas.xclbin ./buil
d_dir.hw.xilinx_u50_gen3x16_xdma_201920_3/config_info.dat 64 64 64 ./data/float/
Read custom sizes of matrix: (64, 64, 64)
Read custom data directory: ./data/float/
xfblasCreate  6273.250042 msec
copyToFpga  0.156553 msec
copyFromFpga  0.517170 msec
Api time is 0.673723 msec
DATA_CSV:,Freq,M,K,N,TimeApiMs,EffApiPct,PerfApiTops
DATA_CSV:,200.000000,64,64,64,0.673723,0.759956,0.000796
>> Kernel #0 << Test passed!
```

GEMM execution Result:

Kernel Route Utilization

Name	LUT	LUTAsMem	REG	BRAM	URAM	DSP
Platform	100406	8640	123812	178	0	4
User Budget	769610	393376	1619548	1166	640	5936
Used Resources	193902	23505	341414	40	24	1376
Unused Resources	575708	369871	1278134	1126	616	4560
blasKernel (1)	193902	23505	341414	40	24	1376
blasKernel_1	193902	23505	341414	40	24	1376

Problem we encountered and how we solved it in L2 kernel:

In Vitis User Guide, When we need to implement the kernel, it only shows the command below.

```
build_dir.hw.xilinx_u250_xdma_201830_2/host.exe build_dir.hw.xilinx_u250_xdma_201830_2/blas.xclbin 64 64 64
```

However, the system reports error:

```
[root@ic21 memKernel]# ./build_dir.hw.xilinx_u50_gen3x16_xdma_201920_3/gemm_bench.exe ./build_dir.hw.xilinx_u50_gen3x16_xdma_201920_3/blas.xclbin 64
xfblasCreate  0.011130 msec
Segmentation fault (core dumped)
```

To solve this issue, we first look into the usage in the figure below. It reveals that we also need configuration file, bin file to implement L2 kernel.

```
[root@ic21 memKernel]# ./build_dir.hw_emu.xilinx_u50_gen3x16_xdma_201920_3/gemm_bench.exe ./build_dir.hw_emu.xilinx_u50_gen3x16_xdma_201920_3/blas.xclbi
n
 usage:
 gemm_bench.exe gemx.xclbin config_info.dat m k n data_dir
 gemm_bench.exe gemx.xclbin config_info.dat
```

The command below is the correct one:

```
[root@ic21 memKernel]# ./build_dir.hw.xilinx_u50_gen3x16_xdma_201920_3/gemm_bench.exe ./build_dir.hw.xilinx_u50_gen3x16_xdma_201920_3/blas.xclbin ./buil
d_dir.hw.xilinx_u50_gen3x16_xdma_201920_3/config_info.dat 64 64 64 ./data/float/
```

L3 Software API:

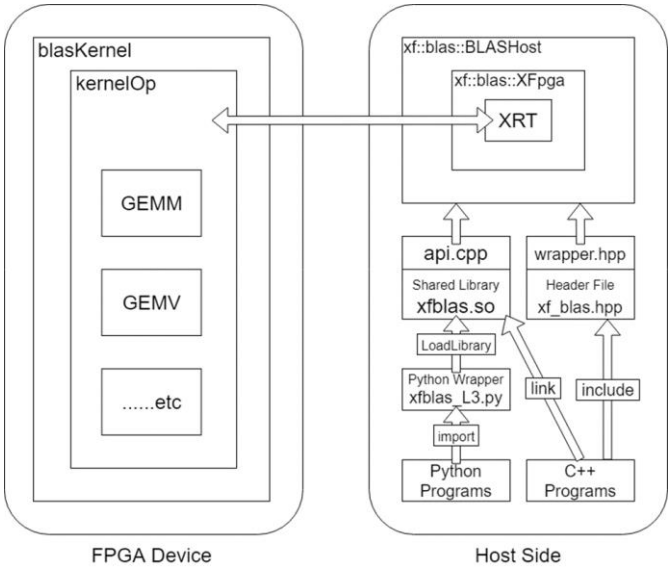
Vitis BLAS level 3 provides software API functions to offload BLAS operations to pre-built FPGA images. It is implemented on top of the XILINX runtime (XRT) and

allows software developers to use Vitis BLAS library without writing any runtime functions and hardware configurations.

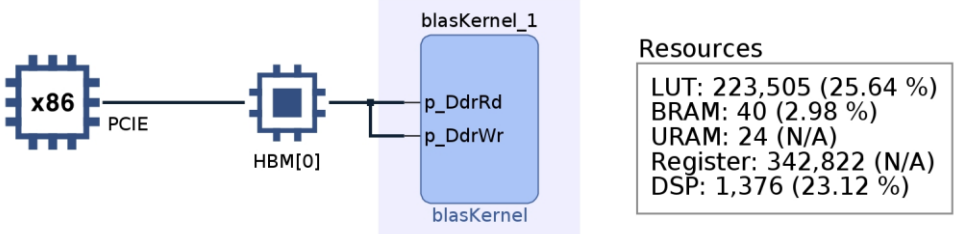
Vitis BLAS library uses row-major storage. The array index of a matrix element could be calculated by the following macro.
 # define IDX2R(i,j,ld) (((i)(ld))+(j))*

How to use L3 Vitis BLAS API?
All the helper functions and the operation functions provided by BLAS Software API have the prefix xfblas.
Software API forms:

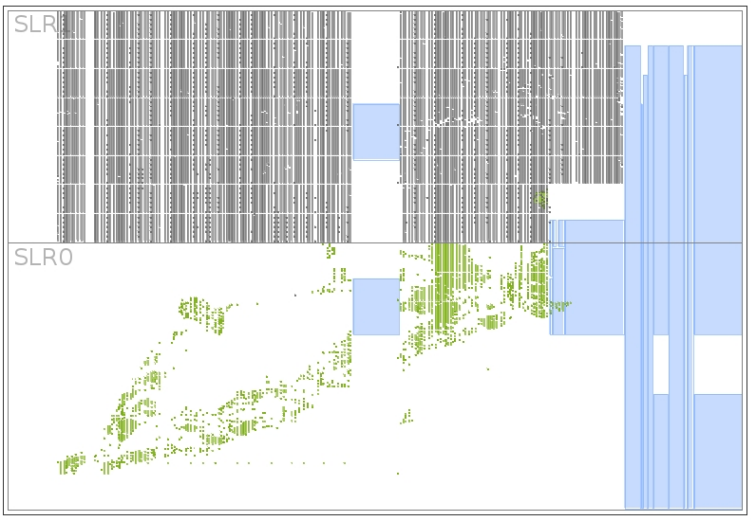
1. Included C++ header file
    - #include "xf_blas.hpp"
2. Python Binding library (Shared Object/ Dynamic Link Library)
    - build xfblas.so
    - import xfblas_L3 as xfblas



2. **Findings from the lab work**
   Vitis Analyser Summary (Hardware):
    - System diagram



    - Device map



- Utilization

Kernel Route Utilization

Name	LUT	LUTAsMem	REG	BRAM	URAM	DSP
Platform	100406	8640	123812	178	0	4
User Budget	769610	393376	1619548	1166	640	5936
Used Resources	193902	23505	341414	40	24	1376
Unused Resources	575708	369871	1278134	1126	616	4560
blasKernel (1)	193902	23505	341414	40	24	1376
blasKernel_1	193902	23505	341414	40	24	1376

3. Analysis

L3 API GEMM benchmark - memKernel

The benchmark performs the matrix-matrix multiplication (A * B = C), M is number of rows of matrix A/C, K is number of columns of matrix A/number of rows of matrix B, N is number of columns of matrix B/C.

GEMM Benchmark result

Software Emulation (sw_emu) - Having error however we don't know how to fix

```
[root@ic21 memKernel]# build_dir.sw_emu.xilinx_u50_gen3x16_xdma_201920_3/gemm_bench.exe build_dir.sw_emu.xilinx_u5
0_gen3x16_xdma_201920_3/blas.xclbin build_dir.sw_emu.xilinx_u50_gen3x16_xdma_201920_3/config_info.dat
xfblasCreate  17.337880 msec
copyToFpga  0.176141 msec
copyFromFpga  691.788477 msec
*** Error in `/opt/Xilinx/Vitis/2021.1/data/emulation/unified/cpu_em/generic_pcie/model/genericpciemodel': munmap_
chunk(): invalid pointer: 0x0000000000f2e000 ***
======= Backtrace: =========
/lib64/libc.so.6(+0x7f474)[0x7f2e78790474]
```

Hardware Emulaiton (hw_emu)

```
[root@ic21 memKernel]# export XCL_EMULATION_MODE=hw_emu
[root@ic21 memKernel]# build_dir.hw_emu.xilinx_u50_gen3x16_xdma_201920_3/gemm_bench.exe build_dir.hw_emu.xilinx_u5
0_gen3x16_xdma_201920_3/blas.xclbin build_dir.hw_emu.xilinx_u50_gen3x16_xdma_201920_3/config_info.dat
INFO: [HW-EMU 01] Hardware emulation runs simulation underneath. Using a large data set will result in long simula
tion times. It is recommended that a small dataset is used for faster execution. The flow uses approximate models
for Global memories and interconnect and hence the performance data generated is approximate.
configuring penguin scheduler mode
scheduler config ert(0), dataflow(1), slots(16), cudma(1), cuisr(0), cdma(0), cus(1)
xfblasCreate  22087.556253 msec
copyToFpga  26.275839 msec
copyFromFpga  139018.122705 msec
Api time is 139044.398544 msec
DATA_CSV:,Freq,M,K,N,TimeApiMs,EffApiPct,PerfApiTops
DATA_CSV:,200.000000,64,64,64,139044.398544,0.000004,0.000000
>> Kernel #0 << Test passed!
INFO: [HW-EMU 06-0] Waiting for the simulator process to exit
INFO: [HW-EMU 06-1] All the simulator processes exited successfully
[root@ic21 memKernel]#
```

HW

```
[root@ic21 memKernel]# export XCL_EMULATION_MODE=hw
[root@ic21 memKernel]# build_dir.hw.xilinx_u50_gen3x16_xdma_201920_3/gemm_bench.exe build_dir.hw.xilinx_u50_gen3x1
6_xdma_201920_3/blas.xclbin build_dir.hw.xilinx_u50_gen3x16_xdma_201920_3/config_info.dat
xfblasCreate  7520.088145 msec
copyToFpga  0.148569 msec
copyFromFpga  0.579887 msec
Api time is 0.728456 msec
DATA_CSV:,Freq,M,K,N,TimeApiMs,EffApiPct,PerfApiTops
DATA_CSV:,200.000000,64,64,64,0.728456,0.702856,0.000737
>> Kernel #0 << Test passed!
```

Benchmark results taken from user guide

M	N	K	api execution time [ms]	api Eff [%]	PerfApiTops
256	256	256	1.370527	19.127241	0.024626
512	512	512	4.517989	46.417820	0.059589
1024	1024	1024	29.500145	56.871639	0.072902
2048	2048	2048	217.555482	61.693563	0.079026
4096	4096	4096	1685.337895	63.710774	0.081580

We can see the API efficiency increased when the size of the multiplication matrix expanded. That may be because the lead time feeding into API and write back time contribute less ratio and thus making API execution time higher ratio.

4. Suggestion for improvement

As we had completed Lab B before, there are many type of pragma eg. pragma HLS pipeline, pragma HLS DATAFLOW, pragma UNROLL. In blas L1, we can understand how it utilize by adding pragma:

```
#pragma HLS PIPELINE
        WideType<t_DataType, t_ParEntries> l_x = p_x.read();
        WideType<t_DataType, t_ParEntries> l_y = p_y.read();
        WideType<t_DataType, t_ParEntries> l_r;
        for (t_IndexType j = 0; j < t_ParEntries; j++) {
#pragma HLS UNROLL
```

```
#pragma HLS stream variable = l_abs depth = 2
#pragma HLS DATAFLOW
```

We had discussed in lab B that using #pragma HLS DATAFLOW can enable task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the register transfer level (RTL) implementation, and increasing the overall throughput of the design.

We think there might be somewhere we can improve, but may not have a significant advantage like adding pragma DATAFLOW.

GitHub link:https://github.com/hank871116/HLS_LAB_C_blas