FLAMES Insight

Wuqiong Zhao

September 18, 2023

This document provides additional insights into the FLAMES (**F**lexible **L**inear **A**lgebra with **M**atrix-**E**mpowered **S**ynthesis) library for Vitis HLS [1]. The FLAMES library is open source at `https://github.com/autohdw/flames`, and the C++ API documentation is available at `https://flames.autohdw.com`.

1 Coverage Comparisons With Armadillo

Though we employ the concept of class-based interfaces from Armadillo [2], the syntax and coverage of FLAMES is not directly comparable with Armadillo.

1.1 Similarities and Differences

1.1.1 Similarities

1. Both the FLAMES library and the Armadillo library utilize modern C++ features in their code implementation;

2. Both the FLAMES library and the Armadillo library primarily focus on constructing classes and provide numerous member functions (methods) which significantly **simplify code** and **enhance readability**;

3. FLAMES has implemented matrix basics in Armadillo, including classes, operators, member functions, and other related matrix operations.

1.1.2 Differences

1. FLAMES is a C++ library for HLS implementation of hardware, while Armadillo is a C++ library for software run on CPU;

2. Though FLAMES is written in C++, it must be synthesizable, i.e., supported by the HLS tool (Vitis HLS [1]);

3. Writing HLS in C++ restricts the usage of class inheritance and virtual functions, and there is a lack of return value optimization (RVO) in Vitis HLS;
4. FLAMES does not include functions that potentially require dynamic memory allocation, such as the sparse matrix class (`SpMat`), as dynamic memory can not be synthesized by Vitis HLS.

1.2 HLS C++ Obstacles

Writing HLS in C++ is *substantially different* from C++ software programming with the CPU target. These differences hinder any HLS library implementation to cover most of the Armadillo library for software designs.

The library could have been more easily implemented with class inheritance and virtual functions (especially for `MatView`), however, the synthesis support for virtual functions has been removed in recent Vitis HLS releases. Another headache is the missing return value optimization (RVO) in Vitis HLS, therefore, returning a `Mat` object will inevitably lead to an unneeded copy process.

1.3 Coverage Comparisons

FLAMES has implemented almost all basics in Armadillo[1], including classes, operators, member functions, and other matrix operations. Nevertheless, FLAMES is still readily extendible and can work with existing algorithm implementations. The contents marked red in the following tables are later added, thanks to the Reviewer.

Table 1 shows the basic class implementations. The sparse matrix class (`SpMat`) is not implemented here, because it can potentially require dynamic memory (for example, change the number of non-zero elements), which is not synthesizable. It is recommended that users design the implementation to avoid use a sparse matrix representation. This feature may be added in the future should users make a reasonable request.

[1]Documentation website: https://arma.sourceforge.net/docs.html.

Table. 1. Coverage comparisons of matrix, vector, cube classes.

Armadillo [2]	FLAMES	Remarks
`Mat`	`Mat`	Dense matrix class.
`Col`	`Col`	Dense column vector class.
`Row`	`Row`	Dense row vector class.
`Cube`	`Row`	Dense cube class.
`SpMat`	N/A	Sparse matrix class.
`const Mat&`	`MatView`	Read-only access to a matrix.
`Mat&`	`MatRef`	Writable reference to a matrix.

Table 2 shows the defined operators. It is worth noting that `==` and `!=` are not used for the matrix equality check. Additionally, we also provided the `<<` operator to print the matrix (only in C++ simulation, not used in synthesis).

Table. 2. Coverage comparisons of operators.

Armadillo [2]	FLAMES	Remarks
`+` (Unary)	`+` (Unary)	Positive sign.
`+` (Binary)	`+` (Binary)	Addition of two objects.
`-` (Unary)	`-` (Unary)	Negative sign.
`-` (Binary)	`-` (Binary)	Subtraction of two objects.
`*`	`*`	Matrix multiplication.
`%`	`%`	Matrix element-wise (Hadamard) multiplication.
`/`	`/`	Matrix element-wise division.
`==`	`==`	Element-wise equality evaluation.
`!=`	`!=`	Element-wise inequality evaluation.
`>=`, `<=`, `>`, `<`	`>=`, `<=`, `>`, `<`	Element-wise comparisons.
`&&`, `||`	`&&`, `||`	Element-wise And and Or logic.

Methods for main matrix member functions are shown in Table 3. Since FLAMES is for hardware implementation, where dynamic memory allocation is not allowed, all operations that modify the size of a matrix are not supported.

Table. 3. Coverage comparisons of main matrix member functions.

Armadillo [2]	FLAMES	Remarks
`.zeros`	`.setZero`	Set all elements to zero.
`.fill`	`.setValue`	Set all elements to a specified value.
`.memptr`	`.rawDataPtr`	Raw pointer to memory.
`[]`	`[]`	Access an element (viewed on 1D).
`(,)`	`(,)`	Access an element (viewed on 2D).
`.col`	`.col/.col_`	Column (as a copy/view).
`.cols`	`.cols/.cols_`	Columns (as a copy/view).
`.row`	`.row/.row_`	Row (as a copy/view).
`.rows`	`.rows/.rows_`	Rows (as a copy/view).
`.t`	`.t/.t_`	Transpose (as a copy/view).
`.i`	`.invNSA`	Matrix inverse (NSA is hardware friendly)[2].
`.diagvec`	`.diagVec`	Take the diagonal and return a vector.
`.diagmat`	`.diagMat`	Generate a diagonal matrix.
`.norm(,"fro")`	`.power`	Power of a matrix (square of ℓ_2-norm).
`.print`	`.print`	Print the matrix (not for synthesis).

Basic matrix operations, including `mul`, `add`, `sub`, `innerProd` are also supported by FLAMES.

FLAMES does not currently cover other functions, including decomposition, factorization, and statics. However, an **interface** to existing algorithm IPs can be easily implemented, by getting the raw data pointer with the `.rawDataPtr` method.

2 FLAMES Simplicity

2.1 Code Length Comparisons

We use a design methodology for general-purpose DSP as a baseline [3] (the HLS book: https://kastner.ucsd.edu/hlsbook/) and compare it with our code implementation using the FLAMES library. There is a mapping relationship between the design methodology in the reference and our baseline (implementations of operations, e.g., matrix multiplication, and addition, can be found in [3]). It is worth noting that through our comparison, achieving the same hardware efficiency and performance as the FLAMES library often requires longer code implementation in the baseline approach.

2.1.1 Matrix-Vector Multiplication

For multiplication between matrix `A` and vector `b`, with the result as `c`, FLAMES has shorter code than the traditional HLS [3]. This example can also be accessed at https://github.com/autohdw/flames/tree/master/examples/mat-vec-multiplication.

With FLAMES.

```cpp
void top(const M& A, const V& b, V& c) { c = A * b; }
```

Without FLAMES. [3]

```cpp
void top(dtype A[4][4], dtype b[4], dtype c[4]) {
#pragma HLS ARRAY_PARTITION variable = A complete
#pragma HLS ARRAY_PARTITION variable = b complete
    for (size_t i = 0; i != 4; ++i) {
        for (size_t j = 0; j != 4; ++j) {
#pragma HLS UNROLL
            c[i] = A[i][j] * b[j];
        }
    }
}
```

2.1.2 Neumann Series Approximation (NSA) Inverse

This example can also be accessed at https://github.com/autohdw/flames/tree/master/examples/mat-inv-nsa.

FLAMES provides a built-in Neumann series approximation (NSA) inversion function `.invNSA()`.

With FLAMES.

```cpp
#include "flames/flames.hpp"

using dtype = FxP<8, 8>;
using M     = Mat<dtype, 4, 4>;

M top(const M& A) { return A.invNSA(); }

int main() {
    M A{ 10, -2, 1, 0, 1, -8, 2, 0, 0, 0, 11, -1, 0, 1, 2, 4 };
    A.print("A = ");
    M A_inv = top(A);
```

```
    A_inv.print("A_inv = ");
    return 0;
}
```

Even without using the .invNSA() function, the implementation of the algorithm can be done with only a few lines of code.

With FLAMES but not directly use .invNSA().

```
#include "flames/flames.hpp"

using dtype = FxP<8, 8>;
using M     = Mat<dtype, 4, 4>;

M top(const M& A) {
    M A_inv;
    const auto D = A.diagMat_(); // diagonal part
    const auto E = A.offDiag_(); // off-diagonal part
    Mat<dtype, 4, 4, MatType::DIAGONAL> D_inv;
    D_inv.invDiag(D); // inverse of diagonal part
    Mat<dtype, 4, 4, MatType::NORMAL> product = (-D_inv) * E;
    Mat<dtype, 4, 4, MatType::NORMAL> sum_tmp = A_inv = product; // the first
    ↪ iteration
    Mat<dtype, 4, 4, MatType::NORMAL> tmp;
    const size_t iter = 4;
MAT_INV_NSA:
    for (size_t i = 1; i < iter; ++i) {
        tmp.mul(A_inv, product);
        A_inv = tmp;
        sum_tmp += tmp;
    }
    A_inv.mul(sum_tmp, D_inv);
    return A_inv += D_inv;
}

int main() {
    M A{ 10, -2, 1, 0, 1, -8, 2, 0, 0, 0, 11, -1, 0, 1, 2, 4 };
    A.print("A = ");
    M A_inv = top(A);
    A_inv.print("A_inv = ");
    return 0;
}
```

Directly using built-in functions provided by Vitis HLS [1] and the design method in [3] without FLAMES, the required code is much longer, and the readability is much poorer.

Without FLAMES.

```
#include "ap_fixed.h"
#include <iostream>
#include <string>
```

```

using dtype = ap_fixed<17, 8>;

void print(dtype A[4][4], std::string s = "") {
#ifndef __SYNTHESIS__
    std::cout << s << "[";
    for (size_t i = 0; i + 1 < 4; ++i) {
        std::cout << "[";
        for (size_t j = 0; j + 1 < 4; ++j) std::cout << A[i][j] << ", ";
        std::cout << A[i][3] << "]," << std::endl;
    }
    std::cout << "[";
    for (size_t j = 0; j + 1 < 4; ++j) std::cout << A[3][j] << ", ";
    std::cout << A[3][3] << "]]" << std::endl;
#endif
}

void mat_copy(dtype from[4][4], dtype to[4][4]) {
#pragma HLS INLINE
#pragma HLS ARRAY_PARTITION variable = from complete
#pragma HLS ARRAY_PARTITION variable = to complete
    for (size_t i = 0; i != 4; ++i) {
#pragma HLS UNROLL
        for (size_t j = 0; j != 4; ++j) {
#pragma HLS LOOP_FLATTEN
            to[i][j] = from[i][j];
        }
    }
}

void top(dtype A[4][4], dtype A_inv[4][4]) {
#pragma HLS ARRAY_PARTITION variable = A complete
#pragma HLS ARRAY_PARTITION variable = A_inv complete
    dtype product[4][4];
    dtype D_inv[4];
    for (size_t i = 0; i != 4; ++i) {
#pragma HLS UNROLL
        D_inv[i] = static_cast<dtype>(1) / A[i][i];
    }
MAT_DIAG_TIMES_MAT_NORMAL:
    for (size_t j = 0; j != 4; ++j) {
#pragma HLS UNROLL
        for (size_t i = 0; i != 4; ++i) {
#pragma HLS LOOP_FLATTEN
            product[i][j] = i == j ? static_cast<dtype>(0) : static_cast<dtype>(-D_inv[i] * A[i][j]);
        }
    }
    dtype sum_tmp[4][4], tmp[4][4];
    mat_copy(product, sum_tmp);
```

```
    mat_copy(product, A_inv);
    const size_t iter = 4;
    for (size_t i = 1; i < iter; ++i) {
    // tmp = A_inv * product;
    GEMM:
        for (size_t _i = 0; _i != 4; ++_i) {
        GEMM_r:
            for (size_t r = 0; r != 4; ++r) {
#pragma HLS UNROLL
            GEMM_c:
                for (size_t c = 0; c != 4; ++c) {
#pragma HLS LOOP_FLATTEN
                    if (_i == 0) tmp[r][c] = static_cast<dtype>(0);
                    tmp[r][c] += A_inv[r][_i] * product[_i][c];
                }
            }
        }
        mat_copy(tmp, A_inv);
        // sum_tmp += tmp;
        for (size_t _i = 0; _i != 4; ++_i) {
#pragma HLS UNROLL
            for (size_t j = 0; j != 4; ++j) {
#pragma HLS LOOP_FLATTEN
                sum_tmp[_i][j] += tmp[_i][j];
            }
        }
    }
// A_inv = sum_tmp * D_inv;
MAT_NORMAL_TIMES_MAT_DIAG:
    for (size_t i = 0; i != 4; ++i) {
#pragma HLS UNROLL
        for (size_t j = 0; j != 4; ++j) {
#pragma HLS LOOP_FLATTEN
            A_inv[i][j] = sum_tmp[i][j] * D_inv[j];
        }
    }
    // A_inv += D_inv;
    for (size_t i = 0; i != 4; ++i) {
#pragma HLS UNROLL
        A_inv[i][i] += D_inv[i];
    }
}

int main() {
    dtype A[4][4] = { { 10, -2, 1, 0 }, { 1, -8, 2, 0 }, { 0, 0, 11, -1 }, { 0, 1, 2, 4 } };
    print(A, "A = ");
    dtype A_inv[4][4];
    top(A, A_inv);
    print(A_inv, "A_inv = ");
```

```
    return 0;
}
```

2.2 Simplicity Beyond Code Length

Writing HLS C++ code with FLAMES enhances code readability by providing modular and reusable components. FLAMES encapsulate complex functionalities into pre-built functions, allowing designers to focus on higher-level logic and abstraction. This makes the code **better organized**, and **less redundant**. It also facilitates *code maintainability* by making the code easier to comprehend and modify.

3 Task-Level Pipelining

Designs **can** leverage task-level pipelining with FLAMES. Auto pipelining is applied in the case study by HLS.

Since the task-level pipelining optimization is provided by the HLS tool (Vitis HLS, for example), there is no significant difference between using FLAMES or not in this term. If anything, FLAMES provides a clearer task flow, where common matrix operations are optimized and managed, which is conducive to task-level pipelining. For instance, the (false) data dependency issue can be avoided by using FLAMES.

To demonstrate the availability of task-level pipelining optimization, an additional example is provided here, which can also be accessed at https://github.com/autohdw/flames/tree/master/examples/task-level-pipelining. In this example, `main_task` can be pipelined by HLS. It is worth noting that the function `main_task` needs to be marked as an `INLINE` function so as to achieve a good pipeline result. The schedule viewer for the `top` function provided by Vitis HLS 2022.2 is shown in Fig. 1.

```cpp
#include "flames/flames.hpp"

using dtype = FxP<6, 2>;
using M = Mat<dtype, 4, 4>;
using V = Vec<dtype, 4>;

void main_task(const M& A, const V& b, V& c) {
    #pragma HLS INLINE
    M tmp1;
    V tmp2;
    tmp1 = A * A;
    tmp2 = tmp1 * b;
    c = tmp2 % tmp2;
}

void top(const M& A1, const M& A2, const M& A3, const V& b, V& c) {
    V c1, c2;
```

```
    M tmp;
    #pragma HLS PIPELINE
    main_task(A1, b, c1);
    main_task(A2, c1, c2);
    main_task(A3, c2, c);
}

int main() {
    M A1, A2, A3;
    V b, c;
    top(A1, A2, A3, b, c);
    return 0;
}
```

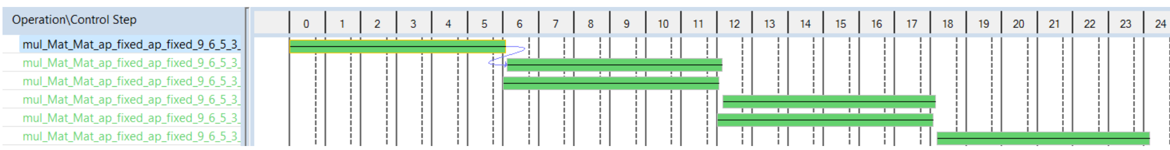

Fig. 1. Task-level pipelining result shown in the schedule viewer.

References

[1] Xilinx, "Vitis high-level synthesis user guide (UG1399)," Accessed: Feb. 27, 2023, 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls.

[2] C. Sanderson and R. Curtin, "Armadillo: A template-based C++ library for linear algebra," *J. Open Source Softw.*, vol. 1, no. 2, p. 26, 2016.

[3] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel programming for FPGAs," *arXiv:1805.03648*, 2018. [Online]. Available: https://arxiv.org/abs/1805.03648.