Deliverable D3.5

1st Version of Open-Source Libraries

Work package:	VP3 Development of 1st Version of Applications	
Author(s):	Panagiotis Miliadis	ICCS
	Chloi Alverti	ICCS
	Albert Kahira	FZJ
	Dimitris Theodoropoulos	ICCS
	Dionisios Pnevmatikatos	ICCS
Reviewer #1	Giovanni Isotton	M3E
Reviewer #2	Albert Kahira	FZJ
Dissemination Level	Public	
Nature	Report	

Confidentiality

This document may contain proprietary material of certain OPTIMA contractors. The commercial use of any information contained in this document may require a license from the proprietor of that information.

This project has received funding from the European High-Performance Computing Joint Undertaking Joint Undertaking (JU) under grant agreement No 955739. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Greece, Germany, Italy, Netherlands,Spain,Switzerland.

Executive Summary

This deliverable provides the details of the OOPS library 1.0 hardware accelerated implementation. The current version supports L1, L2 and L3 routines from the BLAS library, multiplication between sparse matrix and vector, as well as left – upper (LU) matrix decomposition. All kernel implementations were annotated with High-Level Synthesis (HLS) directives to enable hardware mapping onto Xilinx Alveo FPGA cards. For each kernel, D3.5 lists its input / output parameters, HLS implementation, resource utilization, as well as comparison against optimized and non-optimized software versions. This report concludes with a set of future steps to be taken towards enhancing performance, and support for additional software routines.

Table of Contents

List of Abbreviations

HPC	High Performance Computing
FPGA	Field Programmable Gate Array
CSR	Compressed Sparse Row
CUDA	Compute Unified Device Architecture
MPI	Message Passing Interface
PCG	Preconditioned Conjugate Gradient
OpenMP	Open Multi-Processing.
OOPS	OPTIMA OPen Source

1. Introduction

As described in the project proposal, a large set of scientific and industrial applications is based on vector operations, linear / differential equations, and matrix multiplications. Consequently, towards enhancing performance, the project will provide the Optima OPen Source (OOPS) library as an optimized set of software routines that may be used by industrial / scientific software and applications, which will take advantage of the OPTIMA hardware platforms.

This deliverable focuses on the first implementation of the OOPS library kernels. As discussed in D2.2 [6], OOPS will support the kernels listed on the following table:

Table 1 - Kernels supported by the OOPS library.

BLAS	L1	ROT, ROTM, SWAP, SCAL, COPY, AXPY, DOT, DSDOT, NRM2, ASUM, AMIN, AMAX
	L2	GEMV, GBMV, SYMV, SBMV, SPMV, TRMV, TBMV, TPMV, TRSV, TBSV, TPSV
	L3	GEMM, SYMM, TRMM, TRSM
Sparse Linear Algebra	SpMV	
General Computer-Aided Engineering (CAE) solvers	PETSc	Preconditioners: Jacobi Direct solvers: LU factorization / decomposition Krylov: CG

The current version of the OOPS library was developed with the Xilinx Vitis 2020.2 unified platform [7], and the target platform was an Alveo U280 card that hosts a chip with UltraScale+™ architecture, PCIe 4.0 support, and high-bandwidth memory. The card host machine is based on an Intel Xeon Gold 5118 @ 2.30GHz., with up to 24 threads and 16.5 MB L3 cache.

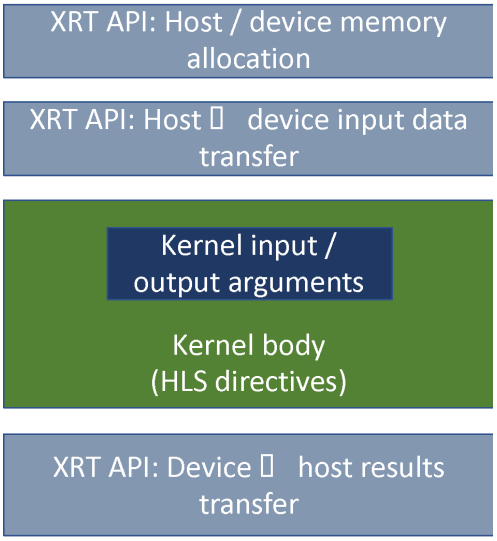


Figure 1 - General data flow and computation of the OOPS kernels.

The OOPS library set exposes high-level function prototypes that applications executed on the host processor can call to enable data processing onto hardware. Figure 1 depicts the overall data flow and computation steps:

1. On the host side, each kernel uses a set of Xilinx Runtime (XRT) system calls to allocate space on the memory located to the FPGA card (either DDR or HBM).
2. On the host side, the software copies input data to the allocated memory space on the FPGA side.
3. On the FPGA side, the hardware IP leverages (when possible) burst data access using the AXI Stream protocol. Using dedicated HLS directives, the IP body processes input data and / or updates intermediates status (e.g., accumulators), and writes back the results.
4. When done, on the host side, the software copies back results to the host memory.

The rest of the deliverable presents the OOPS kernels that are implemented up to now. For each kernel, its corresponding section provides the following information:

- input / output arguments
- HLS implementation
- resource utilization on the selected FPGA platform
- performance charts

D3.5 is organized as follows: Section 2 describes the implementation of the BLAS routines, Section 3 describes the kernel for Sparse Matrix – Vector multiplications (SpMV), whereas Section 4 describes the kernel for left – upper (LU) matrix decomposition. Section 5 concludes this deliverable by listing our next steps towards supporting remaining kernels, as well as enhancing performance of the currently implemented ones.

2. BLAS kernels

This section focuses on the implementation of all BLAS kernels [1], divided into three levels, namely L1, L2 and L3, presented in Sections 2.1, 2.2 and 2.3 respectively.

2.1 Level 1

2.1.1 OOPS_iamax

parameter	direction	type	description
N	input	int	number of elements in input vector
&X	input / output	float	array of floats
incX	input	int	storage spacing between elements of X

This kernel returns the index of the first element having maximum absolute value.

The code snippet below provides its implementation based on HLS directives.

```
int iamax(hls::stream< v_dt>& Xin,const  int N) {
  unsigned int vSize = ((N - 1) / VDATA_SIZE) + 1;
  v_dt temp;
  int i_max[VDATA_SIZE];
  float maxi[VDATA_SIZE];
  float temp_number[VDATA_SIZE];
  #pragma HLS ARRAY_PARTITION variablei_max dim=1 complete
  #pragma HLS ARRAY_PARTITION variable=maxi dim=1 complete
  #pragma HLS ARRAY_PARTITION variable=temp_number dim=1 complete
  init_iamax:
  #pragma HLS pipeline II=1
  temp=Xin.read();
  for(int j=0;j<VDATA_SIZE;j++){
    #pragma HLS unroll
    maxi[j]=abs_float(temp.data[j]);
    i_max[j]=j;
  }
  execute_iamax:
  for (int i = 1; i < vSize; i++) {
    #pragma HLS pipeline II=1
    temp=Xin.read();
    for(int j=0;j<VDATA_SIZE;j++){
      #pragma HLS unroll
      temp_number[j]=abs_float(temp.data[j]);
      if(temp_number[j]>maxi[j]){
        maxi[j]=temp_number[j];
      i_max[j]=i*VDATA_SIZE+j;
    }
```

```
    }
  }

  for (int i=1;i<VDATA_SIZE;i++){
    #pragma HLS pipeline II=1
    if(maxi[i]>maxi[0]){
      maxi[0]=maxi[i];
      i_max[0]=i_max[i];
    }
  }
   return i_max[0];
}
```

In order to fully exploit the data width of the HBM channels, we implement a vectorization technique where data is fetched in batches. Memory controllers are capable of fetching 256 bits of sequential data every clock cycle, while their frequency has been set on 450 MHz. The frequency of the IAMAX kernel is almost the half, so in every clock cycle the memory controllers can read and/or write two batches of 256 bits.

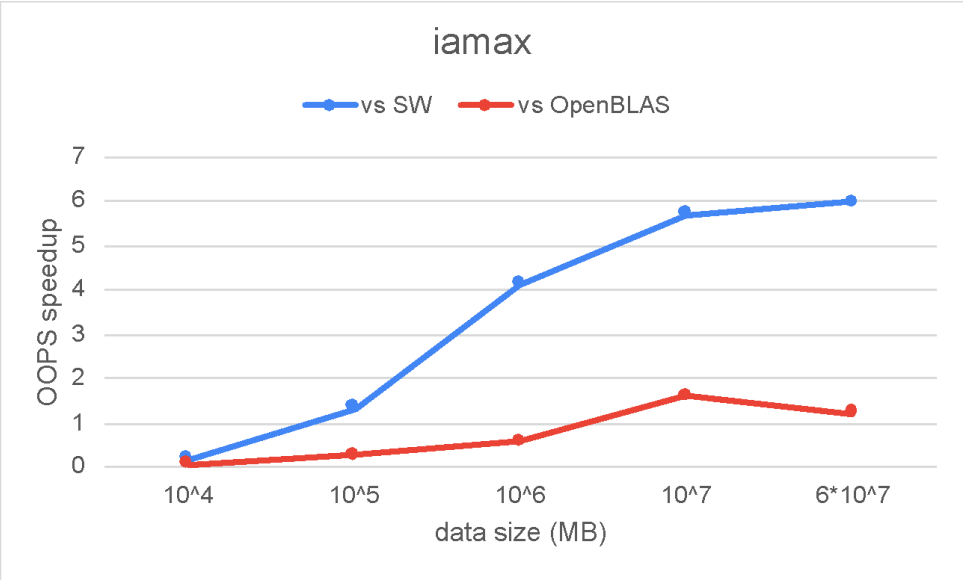
Two functions are implemented in this kernel. Both of them are operating in parallel using the dataflow directive, while data between these functions are passed by using the AXI-stream protocol. The first function implements the vectorization technique, while the second function focuses on the computational part of the kernel. The above snippet presents the latter part of the kernel.

IAMAX function uses three basic HLS directives. Due to the vectorization technique, the unroll primitive is used in order to process 16 float or 8 double elements in parallel. Because the IAMAX kernel returns the index of the element with the max absolute value, temporal arrays are created to store the partial results obtained due to the vectorization technique. Because each cell of the array must be accessed concurrently, we integrated the array_partition primitive, which alleviates the performance bottleneck introduced from parallel accesses. Finally, our kernel uses the pipeline primitive, in order to initiate an iteration of the loop in every clock cycle.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.06	0.11	0.79	0

The chart below compares the IP performance against the designated software implementations:



A single compute unit of our kernel exceeds both the performance of an unoptimized software as well as the OpenBLAS library when a single thread is used. The chart shows that the IAMAX kernel performs worse when a small dataset is used, but the performance significantly increases as the user provides a bigger dataset. This is due to the fact our kernel is fully-pipelined, and each iteration of the loop starts at every clock cycle.

Currently, our kernel uses only one of the thirty-two available channels, or 3.125% of the available memory bandwidth. Along with the low resource consumption on the selected FPGA chip, we can instantiate multiple instances of the kernel which will provide tremendous improvements compared to the initial performance.

2.1.2 OOPS_asum

parameter	direction	type	description
N	input	int	number of elements in input vector
&X	input / output	float	array of floats
incX	input	int	storage spacing between elements of X

This kernel returns the sum of the absolute values.

The code snippet below provides its implementation based on HLS directives.

```
float asum(hls::stream< v_dt>& Xin, int N) {
  unsigned int vSize = ((N - 1) / VDATA_SIZE) + 1;
  float result[VDATA_SIZE];
  float final_result=0;
  #pragma HLS ARRAY_PARTITION result dim=1 complete
```

```
  v_dt temp;
  int count =0;
  init_asum:
  for(int i=0;i<VDATA_SIZE;i++){
    #pragma HLS unroll
    result[i]=0;
  }
  excecute_asum:
  for (int i = 1; i < vSize; i++) {
    #pragma HLS pipeline II=1
    temp=Xin.read();
    for(int j=0;j<VDATA_SIZE;j++){
      #pragma HLS unroll
      count++;
      if (count>N)
        temp_result[j] +=0;
      else
        result[j] +=abs_float(temp.data[j]);
    }
  }
  excecute_final_of_sum:
  for (int i=0;i<VDATA_SIZE;i++){
    #pragma HLS pipeline II=1
    final_result+=result[i];
  }
  return final_result;
}
```

In order to fully exploit the data width of the HBM channels, we implement a vectorization technique where data is fetched in batches. Memory controllers are capable of fetching 256 bits of sequential data every clock cycle, while their frequency has been set on 450 MHz. The frequency of the ASUM kernel is almost the half, so in every clock cycle the memory controllers can read or write two batches of 256 bits.

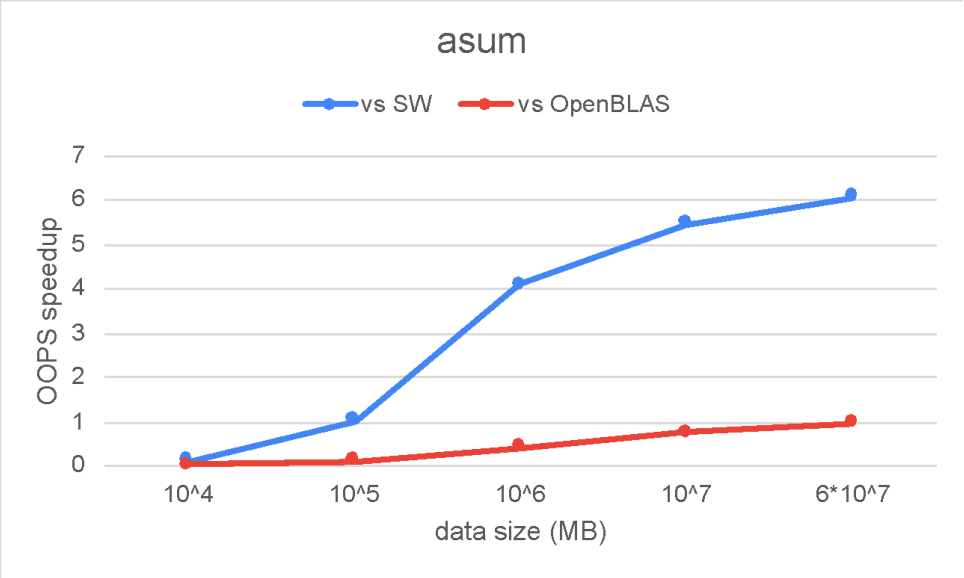
Two functions are implemented in this kernel. Both of them are operating in parallel using the dataflow directive, while data between these functions are passed by using the AXI-stream protocol. The first function implements the vectorization technique, while the second function focuses on the computational part of the kernel. The above snippet presents the latter part of the kernel.

ASUM function uses three basic HLS directives. Due to the vectorization technique, the unroll primitive is used in order to process 16 float or 8 double elements in parallel. Because the ASUM kernel returns the sum of the absolute values of a vector, a temporal array is created to store the partial results obtained due to the vectorization technique. Because each cell of the array must be accessed in parallel, we integrated the array_partition primitive, which alleviates the performance bottleneck introduced from parallel accesses. Finally, our kernel uses the pipeline primitive, in order to initiate an iteration of the loop in every clock cycle.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.12	0.18	0.05	0

The chart below compares the IP performance against the designated software implementations:



A single compute unit of our kernel exceeds both the performance of an unoptimized software as well as the OpenBLAS library when a single thread is used. The chart shows that the ASUM kernel performs worse when a small dataset is used, but the performance significantly increases as the user provides a bigger dataset. This is due to the fact our kernel is fully-pipelined, and in every clock cycle an iteration of the loop starts.

Currently, our kernel uses only one of the thirty-two available channels, or 3.125% of the available memory bandwidth. Along with the low resource consumption on the selected FPGA chip, we can instantiate multiple instances of the kernel which will provide tremendous improvements compared to the initial performance.

2.1.3 OOPS_axpy

parameter	direction	type	description
N	input	int	number of elements in input vector
&X	input	float	array of floats multiplied by scalar
&Y	input/output	float	array of floats added to scaled X
alpha	input	float	scalar applied to X
incX	input	int	storage spacing between elements of X

incY	input	int	storage spacing between elements of Y

This kernel multiples a vector X by constant alpha and adds the sum to a second vector Y.

$$Y \leftarrow alpha * X + Y$$

The output is on vector Y.

The code snippet below provides its implementation based on HLS directives.

```
inline void write_vector_wide_axpy(v_dt* out, hls::stream<v_dt>& Xtemp, hls::stream<v_dt>&
Ytemp ,const int N,const int incy,const float alpha) {
  unsigned int vSize = ((N - 1) / VDATA_SIZE) + 1;

  v_dt X;
  v_dt Y;
  v_dt temp;

  mem_wr:
    for (int i = 0; i < vSize; i++) {
      #pragma HLS pipeline II=1
       X=Xtemp.read();
        Y=Ytemp.read();
        for(int j=0;j<VDATA_SIZE;j++){
          #pragma HLS unroll
            temp.data[j]=Y.data[j] + alpha*X.data[j];
       }
       out[i] = temp;
    }
}
```

In order to fully exploit the data width of the HBM channels, we implement a vectorization technique where data is fetched in batches. Memory controllers are capable of fetching 256 bits of sequential data every clock cycle, while their frequency has been set on 450 MHz. The operating frequency of the AXPY kernel is almost the half, so in every clock cycle the memory controllers can read or write two batches of 256 bits.

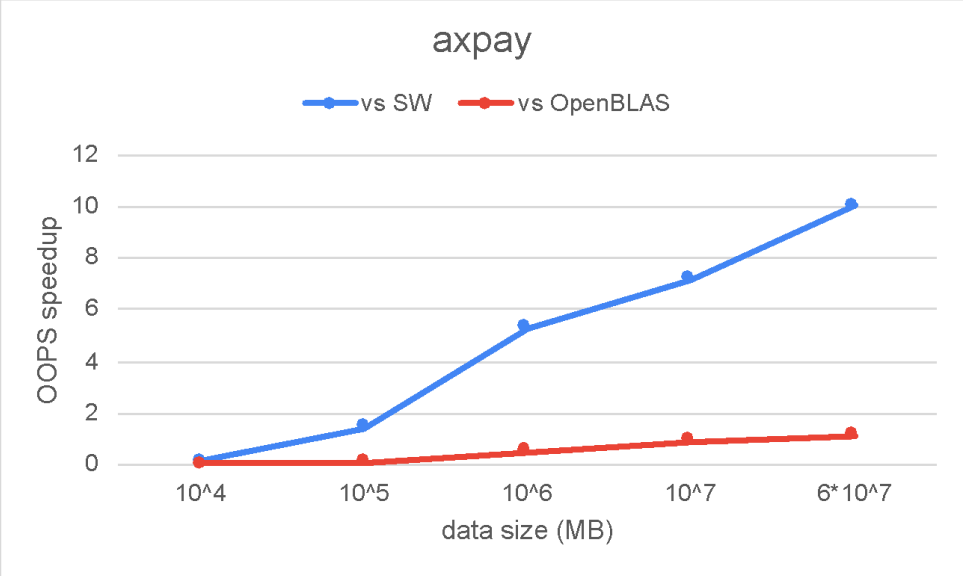
Three functions are implemented in this kernel. All functions are operating in parallel using the dataflow directive, while data between these functions is passed by using the AXI-stream protocol. The first two functions implement the vectorization technique, where batches of elements from both X and Y vectors are read. Meanwhile, the third function focuses on the computational part of the kernel and on storing the result of vector Y. The above snippet presents the latter function of the kernel.

AXPY function uses two basic HLS directives. Due to the vectorization technique, the unroll primitive is used in order to process 16 float or 8 double elements in parallel. Finally, our kernel uses the pipeline primitive, in order to initiate an iteration of the loop in every clock cycle.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.84	0.92	1.88	0.89

The chart below compares the IP performance against the designated software implementations:



A single compute unit of our kernel exceeds both the performance of an unoptimized software as well as the OpenBLAS library when a single thread is used. The chart shows that the AXPY kernel performs worse when a small dataset is used, but the performance significantly increases as the user provides a bigger dataset. This is due to the fact our kernel is fully-pipelined, and in every clock cycle an iteration of the loop starts.

Currently, our kernel uses only two of the thirty-two available channels, or 6.25% of the available memory bandwidth. Along with the low resource consumption on the selected FPGA chip, we can instantiate multiple instances of the kernel which will provide tremendous improvements compared to the initial performance.

2.1.4 OOPS_copy

parameter	direction	type	description
N	input	int	number of elements in input vector
&X	input	float	array of floats
&Y	output	float	array of floats
incX	input	int	storage spacing between elements of X

incY	input	int	storage spacing between elements of Y

This kernel copies a vector X to a second vector Y.

$$Y \leftarrow X$$

The output is on vector Y.

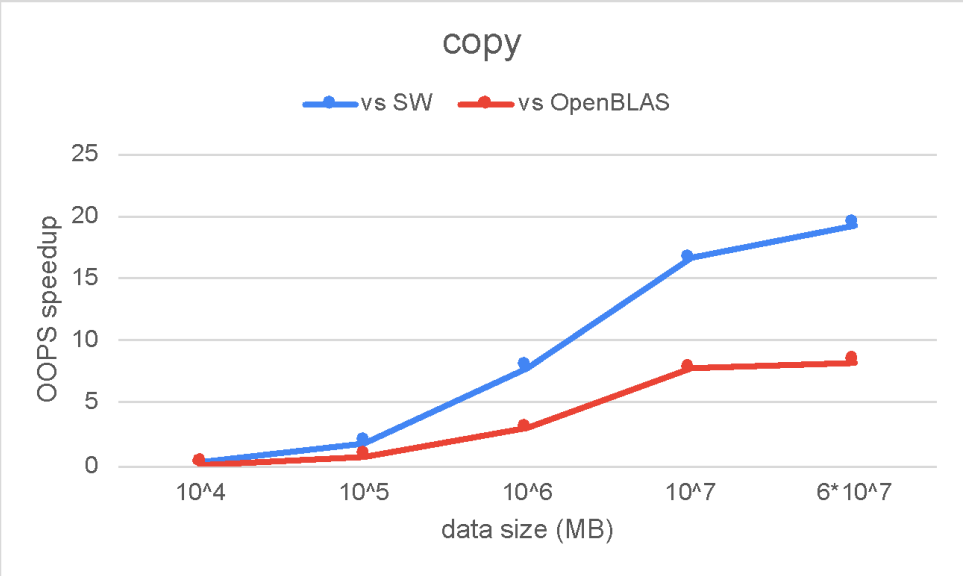
In order to fully exploit the data width of the HBM channels, we implement a vectorization technique where data is fetched in batches. Memory controllers are capable of fetching 256 bits of sequential data every clock cycle, while their frequency has been set on 450 MHz. The operating frequency of the COPY kernel is almost the half, so in every clock cycle the memory controllers can read or write two batches of 256 bits.

Two functions are implemented in this kernel. Both of them are operating in parallel using the dataflow directive, while data between these functions are passed by using the AXI-stream protocol. The first function implements the vectorization technique, where batches of elements from the X vector are read, while the second function stores the elements to vector Y.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.37	0.31	1.14	0.53

The chart below compares the IP performance against the designated software implementations:



A single compute unit of our kernel exceeds both the performance of an unoptimized software as well as the OpenBLAS library when a single thread is used. The chart shows that the COPY kernel performs worse when a small dataset is used, but the performance significantly increases as the user

provides a bigger dataset. This is due to the fact our kernel is fully-pipelined, and in every clock cycle an iteration of the loop starts.

Currently, our kernel uses only two of the thirty-two available channels, or 6.25% of the available memory bandwidth. Along with the low resource consumption on the selected FPGA chip, we can instantiate multiple instances of the kernel which will provide tremendous improvements compared to the initial performance.

2.1.5 OOPS_dot

parameter	direction	type	description
N	input	int	number of elements in input vector
&X	input	float	array of floats
&Y	inpu	float	array of floats
incX	input	int	storage spacing between elements of X
incY	input/output	int	storage spacing between elements of Y

This kernel calculates the dot product between a vector X and vector Y.

$$result = \sum_{i=1}^{N} Xi * Yi$$

The kernel returns the output as a float.

The code snippet below provides its implementation based on HLS directives.

```
static void dot(hls::stream< v_dt>& Xin, hls::stream< v_dt>& Yin, int N,float result) {
  unsigned int vSize = ((N - 1) / VDATA_SIZE) + 1;

  v_dt Xtemp;
  v_dt Ytemp;
  int count=0;
  float temp_result[VDATA_SIZE];
  result=0;
  #pragma HLS ARRAY_PARTITION variable=temp_result dim=1 complete

  for(int i=0;i<VDATA_SIZE;i++){
    #pragma HLS unroll
    temp_result[i]=0;
  }

  execute:
    for (int i = 0; i < vSize; i++) {
```

```
      #pragma HLS pipeline II=1
      Xtemp=Xin.read();
      Ytemp=Yin.read();
      for (int j=0;j<VDATA_SIZE;j++){
         #pragma HLS unroll
         count++;
         if (count>N)
           temp_result[j] +=0;
         else
           temp_result[j]+=Xtemp.data[j]*Ytemp.data[j];
       }
   }

   accum:
     for (int j=0;j<VDATA_SIZE;j++){
      #pragma HLS pipeline II=1
     result+=temp_result[j];
     }
}
```

In order to fully exploit the data width of the HBM channels, we implement a vectorization technique where data is fetched in batches. Memory controllers are capable of fetching 256 bits of sequential data every clock cycle, while their frequency has been set on 450 MHz. The frequency of the ASUM kernel is almost the half, so in every clock cycle the memory controllers can read or write two batches of 256 bits.

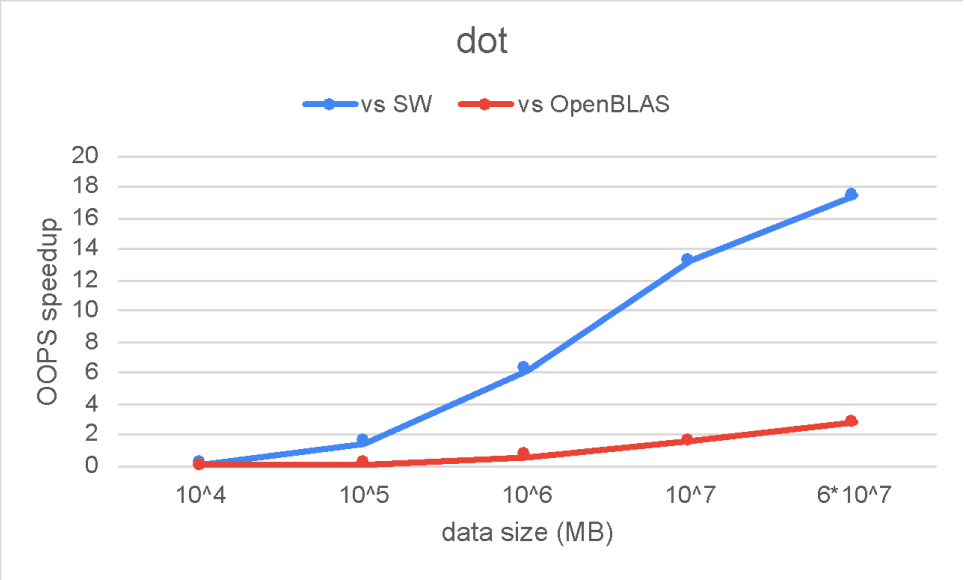
Three functions are implemented in this kernel. All functions are operating in parallel using the dataflow directive, while data between these functions are passed by using the AXI-stream protocol. The first two functions implement the vectorization technique, for reading both X and Y vectors, while the third function focuses on the computational part of the kernel. The above snippet presents the latter part of the kernel.

DOT function uses three basic HLS directives. Due to the vectorization technique, the unroll primitive is used in order to process 16 float or 8 double elements in parallel. Because the DOT kernel returns the dot product between a vector X and vector Y, a temporal array is created to store the partial results obtained due to the vectorization technique. Because each cell of the array must be accessed in parallel, we integrated the array_partition primitive, which alleviates the performance bottleneck introduced from parallel accesses. Finally, our kernel uses the pipeline primitive, in order to initiate an iteration of the loop in every clock cycle.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.12	0.2	0.05	0

The chart below compares the IP performance against the designated software implementations:



A single compute unit of our kernel exceeds both the performance of an unoptimized software as well as the OpenBLAS library when a single thread is used. The chart shows that the DOT kernel performs worse when a small dataset is used, but the performance significantly increases as the user provides a bigger dataset. This is due to the fact our kernel is fully-pipelined, and in every clock cycle an iteration of the loop starts.

Currently, our kernel uses only two of the thirty-two available channels, or 6.25% of the available memory bandwidth. Along with the low resource consumption on the selected FPGA chip, we can instantiate multiple instances of the kernel which will provide tremendous improvements compared to the initial performance.

2.1.6 OOPS_sddot

parameter	direction	type	description
N	input	int	number of elements in input vector
&X	input	float	array of floats multiplied by scalar
&Y	input	float	array of floats added to scaled X
alpha	input	float	scalar applied to the dot product
incX	input	int	storage spacing between elements of X
incY	input	int	storage spacing between elements of Y

This kernel computes the inner product of two vectors X and Y with extended precision accumulation.

$$result = alpha + \sum_{i=1}^{N} Xi * Yi$$

The kernel returns the output as a double.

The code snippet below provides its implementation based on HLS directives.

```
static void ddot(hls::stream< v_dt>& Xin, hls::stream< v_dt>& Yin,const int N,double result) {
   unsigned int vSize = ((N - 1) / VDATA_SIZE) + 1;
  double temp_result[VDATA_SIZE];
  #pragma HLS ARRAY_PARTITION temp_result dim=1 complete
   v_dt x,y;
  init_ddot:
  for(int i=0;i<VDATA_SIZE;i++){
    #pragma HLS unroll
    temp_result[i]=0;
  }
  execute:
  for (int i = 0; i < vSize; i++) {
    #pragma HLS pipeline II=1
    x=Xin.read();
    y=Yin.read();
    for(int j=0;j<VDATA_SIZE;j++){
      #pragma HLS unroll
      temp_result[j] +=(x.data[j]*y.data[j]);
    }
  }
  excecute_final_of_ddot:
    for (int i=0;i<VDATA_SIZE;i++){
      #pragma HLS pipeline II=1
      result+=temp_result[i];
    }

}
```

In order to fully exploit the data width of the HBM channels, we implement a vectorization technique where data is fetched in batches. Memory controllers are capable of fetching 256 bits of sequential data every clock cycle, while their frequency has been set on 450 MHz. The frequency of the SDDOT kernel is almost the half, so in every clock cycle the memory controllers can read or write two batches of 256 bits.

Three functions are implemented in this kernel. All functions are operating in parallel using the dataflow directive, while data between these functions are passed by using the AXI-stream protocol. The first two functions implement the vectorization technique, for reading both X and Y vectors, while the third function focuses on the computational part of the kernel. The above snippet presents the latter part of the kernel.

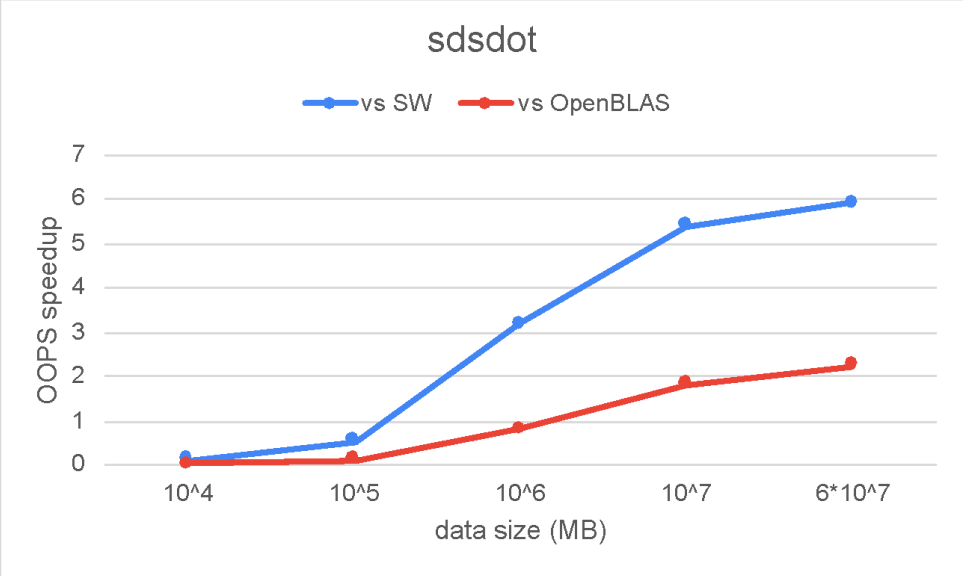
The SDDOT function uses three basic HLS directives. Due to the vectorization technique, the unroll primitive is used in order to process 16 float or 8 double elements in parallel. Because the SDDOT

kernel returns the inner product of two vectors X and Y with extended precision accumulation, a temporal array is created to store the partial result obtained due to the vectorization technique. Because each cell of the array must be accessed in parallel, we integrated the array_partition primitive, which alleviates the performance bottleneck introduced from parallel accesses. Finally, our kernel uses the pipeline primitive, in order to initiate an iteration of the loop in every clock cycle.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.2	0.28	0.05	0.02

The chart below compares the IP performance against the designated software implementations:



A single compute unit of our kernel exceeds both the performance of an unoptimized software as well as the OpenBLAS library when a single thread is used. The chart shows that SDDOT kernel performs worse when a small dataset is used, but the performance significantly increases as the user provides a bigger dataset. This is due to the fact our kernel is fully-pipelined, and in every clock cycle an iteration of the loop starts.

Currently, our kernel uses only two of the thirty-two available channels, or 6.25% of the available memory bandwidth. Along with the low resource consumption on the selected FPGA chip, we can instantiate multiple instances of the kernel which will provide tremendous improvements compared to the initial performance.

2.1.7 OOPS_nrm2

parameter	direction	type	description

N	input	int	number of elements in input vector
&X	input	float	array of floats
incX	input	int	storage spacing between elements of X

This kernel returns the Euclidean norm of a vector X.

$$result \leftarrow \|X\|$$

The code snippet below provides its implementation based on HLS directives.

```
static float nrm2(hls::stream< v_dt>& Xin, const int N) {
  unsigned int vSize = ((N - 1) / VDATA_SIZE) + 1;
  float result[VDATA_SIZE];
  float final_result=0;
  #pragma HLS ARRAY_PARTITION result dim=1 complete
  v_dt temp;
  init_nrm2:
  for(int i=0;i<VDATA_SIZE;i++){
    #pragma HLS unroll
    result[i]=0;
  }
  excecute_nrm2:
  for (int i = 1; i < vSize; i++) {
    #pragma HLS pipeline II=1
    temp=Xin.read();
    for(int j=0;j<VDATA_SIZE;j++){
      #pragma HLS unroll
      result[j] +=(temp.data[j]*temp.data[j]);
    }
  }
  excecute_final_of_nrm2:
  for (int i=0;i<VDATA_SIZE;i++){
    #pragma HLS pipeline II=1
    final_result+=result[i];
  }

   final_result =sqrtf(final_result);
   return final_result;
}
```

In order to fully exploit the data width of the HBM channels, we implement a vectorization technique where data is fetched in batches. Memory controllers are capable of fetching 256 bits of sequential data every clock cycle, while their frequency has been set on 450 MHz. The frequency of the NRM2 kernel is almost the half, so in every clock cycle the memory controllers can read or write two batches of 256 bits.

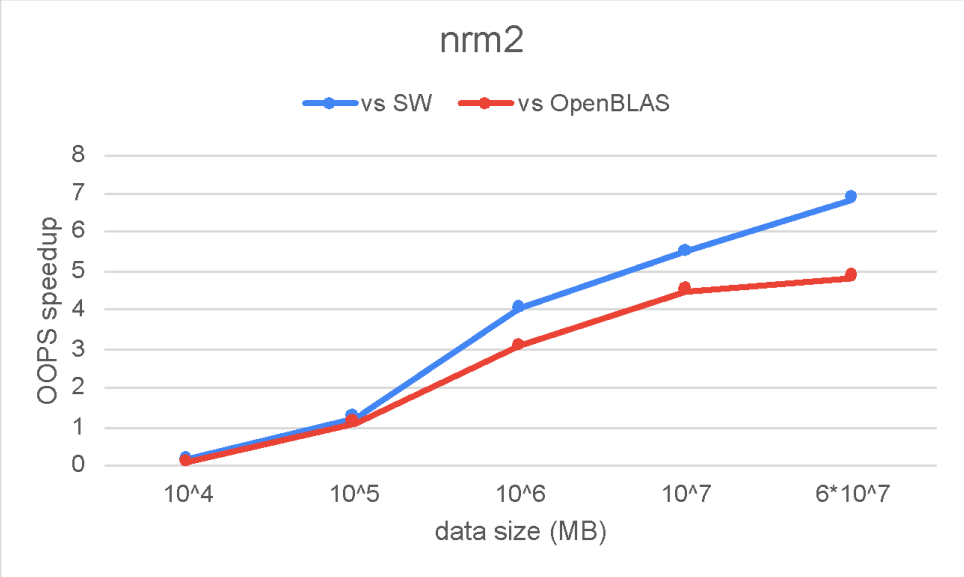
Two functions are implemented in this kernel. Both functions are operating in parallel using the dataflow directive, while data between these functions are passed by using the AXI-stream protocol. The first function implements the vectorization technique, for reading the X vector, while the second function focuses on the computational part of the kernel. The above snippet presents the latter part of the kernel.

NRM2 function uses three basic HLS directives. Due to the vectorization technique, the unroll primitive is used in order to process 16 float or 8 double elements in parallel. Because the NRM2 kernel returns the Euclidean norm of a vector X, a temporal array is created to store the partial results obtained due to the vectorization technique. Because each cell of the array must be accessed in parallel, we integrated the array_partition primitive, which alleviates the performance bottleneck introduced from parallel accesses. Finally, our kernel uses the pipeline primitive, in order to initiate an iteration of the loop in every clock cycle.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.12	0.18	0.05	0

The chart below compares the IP performance against the designated software implementations:



A single compute unit of our kernel exceeds both the performance of an unoptimized software as well as the OpenBLAS library when a single thread is used. The chart shows that the NRM2 kernel performs worse when a small dataset is used, but the performance significantly increases as the user provides a bigger dataset. This is due to the fact our kernel is fully-pipelined, and in every clock cycle an iteration of the loop starts.

Currently, our kernel uses only one of the thirty-two available channels, or 3.125% of the available memory bandwidth. Along with the low resource consumption on the selected FPGA chip, we can

instantiate multiple instances of the kernel which will provide tremendous improvements compared to the initial performance.

2.1.8 OOPS_rot

parameter	direction	type	description
N	input	int	number of elements in input vector
&X	input/output	float	array of floats
incX	input	int	storage spacing between elements of X
&Y	input/output	float	array of floats
incY	input	int	storage spacing between elements of Y
c	input	float	specifies the scalar c
s	input	float	specifies the scalar s

This kernel applies a plane rotation to vectors X and Y, based on the following calculations:

$$[X\ Y\ ] \leftarrow [c\ *\ X\ +\ S\ *\ Y\ \ -\ s\ *\ X\ +\ C\ *\ Y\ ]$$

The code snippet below provides its implementation based on HLS directives.

```
static void rot (hls::stream< v_dt>& Xin, hls::stream< v_dt>& Yin,hls::stream< v_dt>& Xout, hls::stream< v_dt>& Yout, const int N,const float C,const float S) {
   unsigned int vSize = ((N - 1) / VDATA_SIZE) + 1;
  float temp_x[VDATA_SIZE];
  float temp_y[VDATA_SIZE];
  #pragma HLS ARRAY_PARTITION temp_x dim=1 complete
  #pragma HLS ARRAY_PARTITION temp_y dim=1 complete
  v_dt x,y;
  execute:
  for (int i = 0; i < vSize; i++) {
    #pragma HLS pipeline II=1
    x=Xin.read();
    y=Yin.read();
    for(int j=0;j<VDATA_SIZE;j++){
      #pragma HLS unroll
      temp_x[j]=x.data[j];
      temp_y[j]=y.data[j];
      x.data[j]= (C*temp_x[j]+S*temp_y[j]);
      y.data[j]= (-S*temp_x[j]+C*temp_y[j]);
    }
    Xout << x;
    Yout <<y;
```

```
  }

}
```

In order to fully exploit the data width of the HBM channels, we implement a vectorization technique where data is fetched in batches. Memory controllers are capable of fetching 256 bits of sequential data every clock cycle, while their frequency has been set on 450 MHz. The frequency of the ROT kernel is almost the half, so in every clock cycle the memory controllers can read or write two batches of 256 bits.

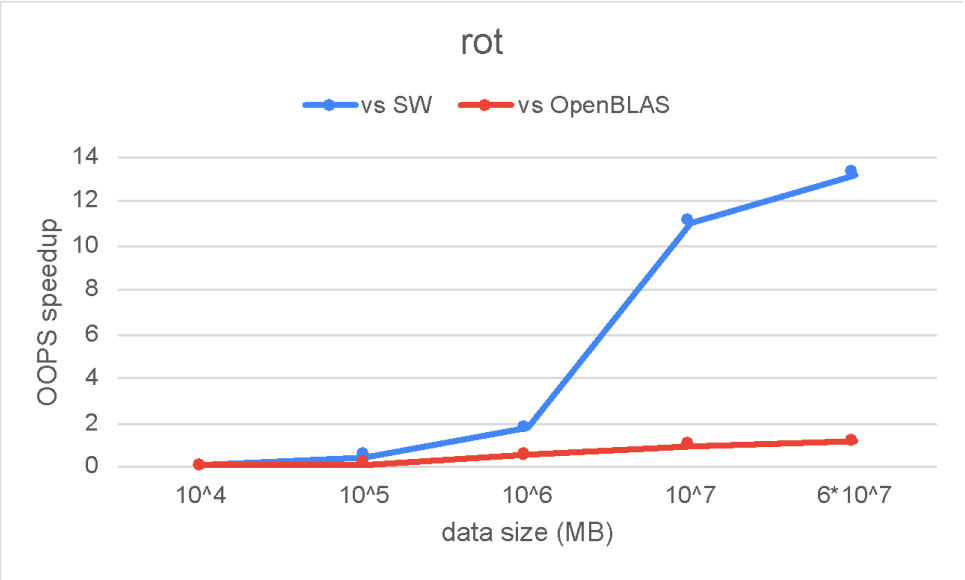
Five functions are implemented in this kernel. All functions are operating in parallel using the dataflow directive, while data between these functions are passed by using the AXI-stream protocol. Four out of five functions implement the vectorization technique, for reading and writing both vectors X and Y, while the fifth function focuses on the computational part of the kernel. The above snippet presents the latter part of the kernel.

ROT function uses three basic HLS directives. Due to the vectorization technique, the unroll primitive is used in order to process 16 float or 8 double elements in parallel. Because the ROT kernel applies a plane rotation to vectors X and Y, two temporal arrays are created to temporarily store the initial elements from X and Y vectors. Because each cell of the array must be accessed in parallel, we integrated the array_partition primitive, which alleviates the performance bottleneck introduced from parallel accesses. Finally, our kernel uses the pipeline primitive, in order to initiate an iteration of the loop in every clock cycle.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
1.65	1.57	3.03	2.84

The chart below compares the IP performance against the designated software implementations:



A single compute unit of our kernel exceeds both the performance of an unoptimized software as well as the OpenBLAS library when a single thread is used. The chart shows that ROT kernel performs worse when a small dataset is used, but the performance significantly increases as the user provides a bigger dataset. This is due to the fact our kernel is fully-pipelined, and in every clock cycle an iteration of the loop starts.

Currently, our kernel uses only two of the thirty-two available channels, or 6.25% of the available memory bandwidth. Along with the low resource consumption on the selected FPGA chip, we can instantiate multiple instances of the kernel which will provide tremendous improvements compared to the initial performance.

2.1.9 OOPS_rotm

parameter	direction	type	description
N	input	int	number of elements in input vector
&X	input/output	float	array of floats
incX	input	int	storage spacing between elements of X
&Y	input/output	float	array of floats
incY	input	int	storage spacing between elements of Y
&P	input	float	a float parameter array with dimension 5, param (1) contains a switch, flag. param (2-5) contain h11, h21, h12, and h22, respectively

This kernel performs modified Givens rotation of points in the plane as follows:

$[xi\ yi\ ] \leftarrow H[xi\ yi\ ]$, where i=1…N, and H is a modified Givens transformation matrix.

The code snippet below provides its implementation based on HLS directives.

```
static void rotm (hls::stream< v_dt>& Xin, hls::stream< v_dt>& Yin,hls::stream< v_dt>& Xout,
hls::stream< v_dt>& Yout, const int N,const float *P) {
   unsigned int vSize = ((N - 1) / VDATA_SIZE) + 1;
  float pis[4];
  for (int i=0;i<5;i++){
    pis[i]=P[i];
  }
  float temp_x[VDATA_SIZE];
  float temp_y[VDATA_SIZE];
  #pragma HLS ARRAY_PARTITION temp_x dim=1 complete
  #pragma HLS ARRAY_PARTITION temp_y dim=1 complete
  #pragma HLS ARRAY_PARTITION pis dim=1 complete
  v_dt x,y;

  execute:
    if(p0>0){
      pis[2]=1;
      pis[1]=-1;
    }
    else if(p0==0){
      pis[0]=1;
      pis[3]=1;
    }
    for (int i = 0; i < vSize; i++) {
      #pragma HLS pipeline II=1
      x=Xin.read();
      y=Yin.read();
      for(int j=0;j<VDATA_SIZE;j++){
        #pragma HLS unroll
        temp_x[j]=x.data[j];
        temp_y[j]=y.data[j];
        x.data[j]= (pis[0]*temp_x[j]+pis[2]*temp_y[j]);
        y.data[j]= (pis[1]*temp_x[j]+pis[3]*temp_y[j]);
      }
      Xout <<x;
      Yout<<y;
    }
}
```

In order to fully exploit the data width of the HBM channels, we implement a vectorization technique where data is fetched in batches. Memory controllers are capable of fetching 256 bits of sequential data every clock cycle, while their frequency has been set on 450 MHz. The frequency of the ROTM

kernel is almost the half, so in every clock cycle the memory controllers can read or write two batches of 256 bits.

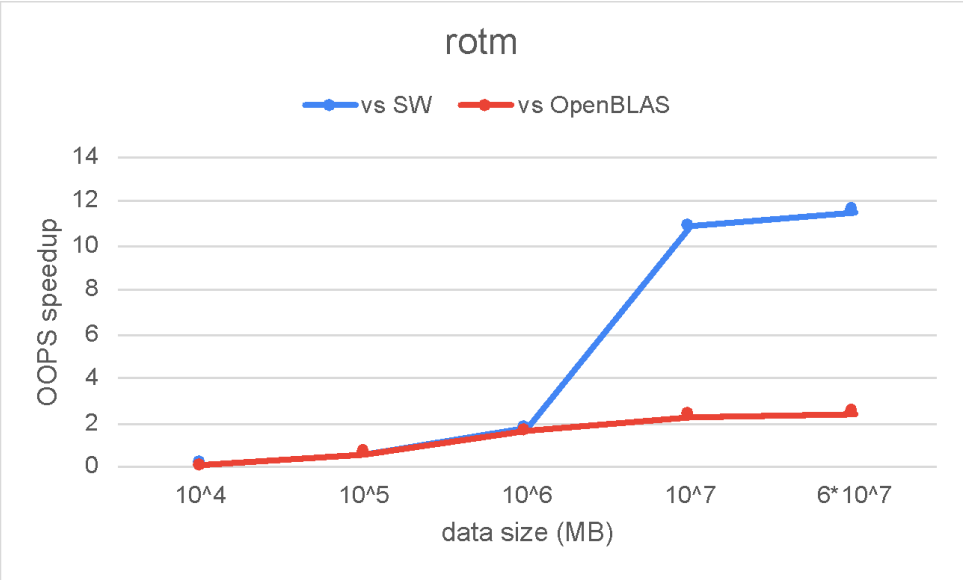
Five functions are implemented in this kernel. All functions are operating in parallel using the dataflow directive, while data between these functions are passed by using the AXI-stream protocol. Four out of five functions implement the vectorization technique, for reading and writing both on vectors X and Y, while the fifth function focuses on the computational part of the kernel. The above snippet presents the latter part of the kernel.

ROTM function uses three basic HLS directives. Due to the vectorization technique, the unroll primitive is used in order to process 16 float or 8 double elements in parallel. Because the ROTM kernel performs modified Givens rotation, two temporal arrays are created to temporarily store the initial elements from X and Y vectors. Because each cell of the array must be accessed in parallel, we integrated the array_partition primitive, which alleviates the performance bottleneck introduced from parallel accesses. Finally, our kernel uses the pipeline primitive, in order to initiate an iteration of the loop in every clock cycle.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
1.57	1.45	3.08	1.77

The chart below compares the IP performance against the designated software implementations:



A single compute unit of our kernel exceeds both the performance of an unoptimized software as well as the OpenBLAS library when a single thread is used. The chart shows that the ROTM kernel performs worse when a small dataset is used, but the performance significantly increases as the user

provides a bigger dataset. This is due to the fact our kernel is fully-pipelined, and in every clock cycle an iteration of the loop starts.

Currently, our kernel uses only two of the thirty-two available channels, or 6.25% of the available memory bandwidth. Along with the low resource consumption on the selected FPGA chip, we can instantiate multiple instances of the kernel which will provide tremendous improvements compared to the initial performance.

2.1.10 OOPS_scal

parameter	direction	type	description
N	input	int	number of elements in input vector
&X	input/output	float	array of floats
incX	input	int	storage spacing between elements of X
alpha	input	float	specifies the scalar alpha

This kernel scales a vector by a constant alpha.

The code snippet below provides its implementation based on HLS directives.

```
inline void write_vector_wide_scal(v_dt* out, hls::stream<v_dt>& outStream,const int N,const int
incy,const float alpha) {
unsigned int vSize = ((N - 1) / VDATA_SIZE) + 1;

mem_wr:
   for (int i = 0; i < vSize; i++) {
#pragma HLS pipeline II=1
          v_dt temp=outStream.read();
          for(int j=0;j<VDATA_SIZE;j++){
                    #pragma HLS unroll
                    temp.data[j]=temp.data[j]*alpha;
          }
      out[i] = temp;
   }
}
```

In order to fully exploit the data width of the HBM channels, we implement a vectorization technique where data is fetched in batches. Memory controllers are capable of fetching 256 bits of sequential data every clock cycle, while their frequency has been set on 450 MHz. The frequency of the SCAL kernel is almost the half, so in every clock cycle the memory controllers can read or write two batches of 256 bits.

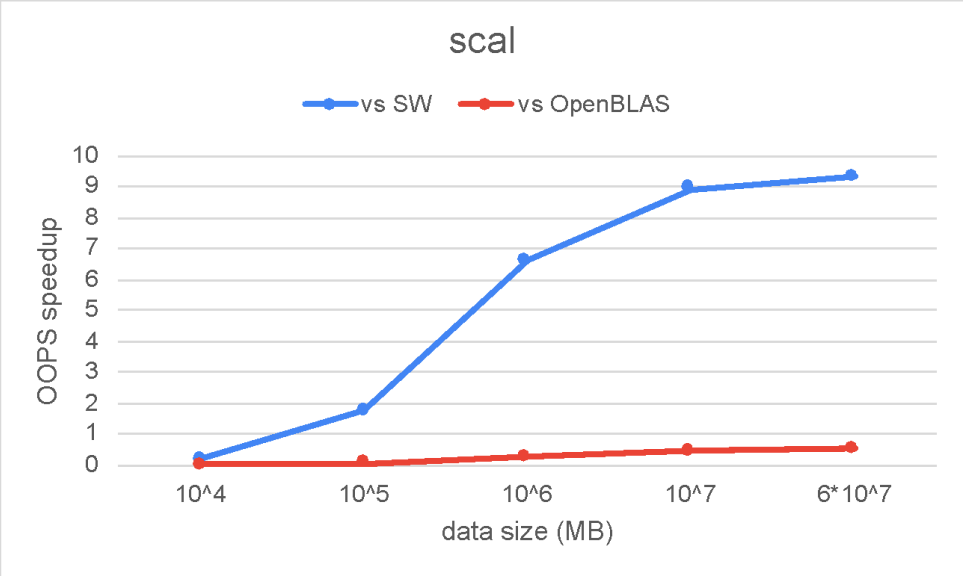
Two functions are implemented in this kernel. Both of them are operating in parallel using the dataflow directive, while data between these functions are passed by using the AXI-stream protocol. The first function implements the vectorization technique, for reading vector X, while the second function focuses on the computational part of the kernel and writing the results back in memory. The above snippet presents the latter part of the kernel.

SCAL function uses two basic HLS directives. Due to the vectorization technique, the unroll primitive is used in order to process 16 float or 8 double elements in parallel. Finally, our kernel uses the pipeline primitive, in order to initiate an iteration of the loop in every clock cycle.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.49	0.49	0.74	0.53

The chart below compares the IP performance against the designated software implementations:



SCAL is the only kernel where the OOPS library has lower performance compared to OpenBLAS when a single thread is used. However, the performance gap closes as the user provides a bigger dataset. This is due to the fact our kernel is fully-pipelined, and in every clock cycle an iteration of the loop starts.

However, our kernel uses only one of the thirty-two available channels, or 3.125% of the available memory bandwidth. Along with the low resource consumption on the selected FPGA chip, we can instantiate multiple instances of the kernel which will provide tremendous improvements compared to the initial performance.

2.1.11 OOPS_swap

parameter	direction	type	description
N	input	int	number of elements in input vector
&X	input/output	float	array of floats
incX	input	int	storage spacing between elements of X
&Y	input/output	float	array of floats
incY	input	int	storage spacing between elements of Y

This kernel interchanges two vectors X and Y.

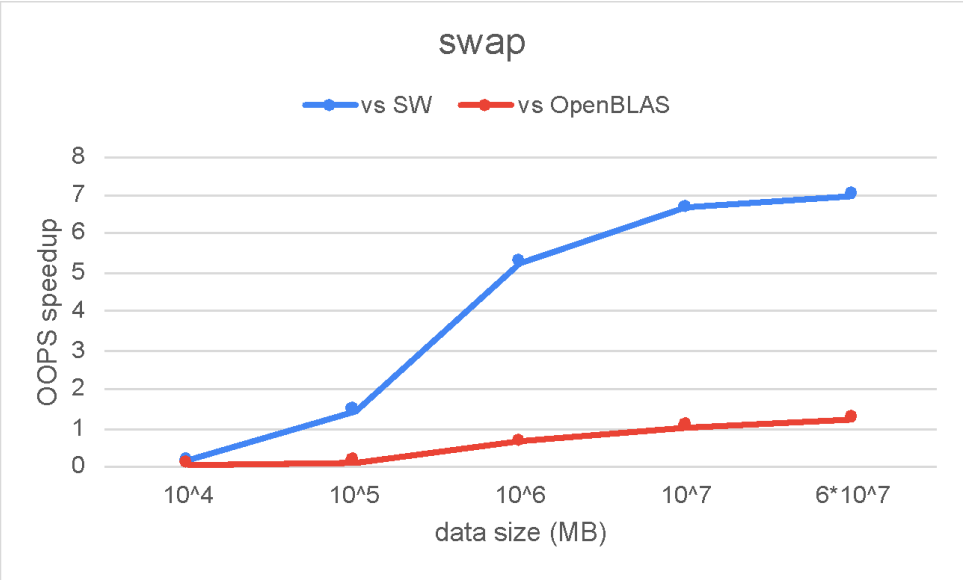
In order to fully exploit the data width of the HBM channels, we implement a vectorization technique where data is fetched in batches. Memory controllers are capable of fetching 256 bits of sequential data every clock cycle, while their frequency has been set on 450 MHz. The operating frequency of the SWAP kernel is almost the half, so in every clock cycle the memory controllers can read or write two batches of 256 bits.

Four functions are implemented in this kernel. All functions are operating in parallel using the dataflow directive, while data between these functions are passed by using the AXI-stream protocol. The first two functions implement the vectorization technique, where multiple elements of the X and Y vector are read, while the other two functions store the elements into Y and X vector respectively.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.74	0.61	2.28	0

The chart below compares the IP performance against the designated software implementations:



A single compute unit of our kernel exceeds both the performance of an unoptimized software as well as the OpenBLAS library when a single thread is used. The chart shows that the SWAP kernel performs worse when a small dataset is used, but the performance significantly increases as the user provides a bigger dataset. This is due to the fact our kernel is fully-pipelined, and in every clock cycle an iteration of the loop starts.

Currently, our kernel uses only two of the thirty-two available channels, or 6.25% of the available memory bandwidth. Along with the low resource consumption on the selected FPGA chip, we can instantiate multiple instances of the kernel which will provide tremendous improvements compared to the initial performance.

2.1.12 OOPS_iamin

parameter	direction	type	description
N	input	int	number of elements in input vector
&X	input / output	float	array of floats
incX	input	int	storage spacing between elements of X

This kernel returns the index of the first element having minimum absolute value.

The code snippet below provides its implementation based on HLS directives.

```
int iamin(hls::stream< v_dt>& Xin,const  int N) {
  unsigned int vSize = ((N - 1) / VDATA_SIZE) + 1;
  v_dt temp;
  int i_min[VDATA_SIZE];
```

```
  float mini[VDATA_SIZE];
  float temp_number[VDATA_SIZE];
  #pragma HLS ARRAY_PARTITION variable=i_min dim=1 complete
  #pragma HLS ARRAY_PARTITION variable=mini dim=1 complete
  #pragma HLS ARRAY_PARTITION variable=temp_number dim=1 complete
  init_iamin:
  #pragma HLS pipeline II=1
  temp=Xin.read();
  for(int j=0;j<VDATA_SIZE;j++){
    #pragma HLS unroll
    mini[j]=abs_float(temp.data[j]);
    i_min[j]=j;
  }

  execute_iamin:
    for (int i = 1; i < vSize; i++) {
      #pragma HLS pipeline II=1
     temp=Xin.read();
      for(int j=0;j<VDATA_SIZE;j++){
        #pragma HLS unroll
        temp_number[j]=abs_float(temp.data[j]);
        if(temp_number[j]>mini[j]){
          mini[j]=temp_number[j];
          i_min[j]=i*VDATA_SIZE+j;
        }
      }
    }

  for (int i=1;i<VDATA_SIZE;i++){
    #pragma HLS pipeline II=1
    If (mini[i]>mini[0]){
      mini[0]=mini[i];
      i_min[0]=i_min[i];
    }
  }
   return i_min[0];
}
```

In order to fully exploit the data width of the HBM channels, we implement a vectorization technique where data is fetched in batches. Memory controllers are capable of fetching 256 bits of sequential data every clock cycle, while their frequency has been set on 450 MHz. The frequency of the IAMIN kernel is almost the half, so in every clock cycle the memory controllers can read or write two batches of 256 bits.

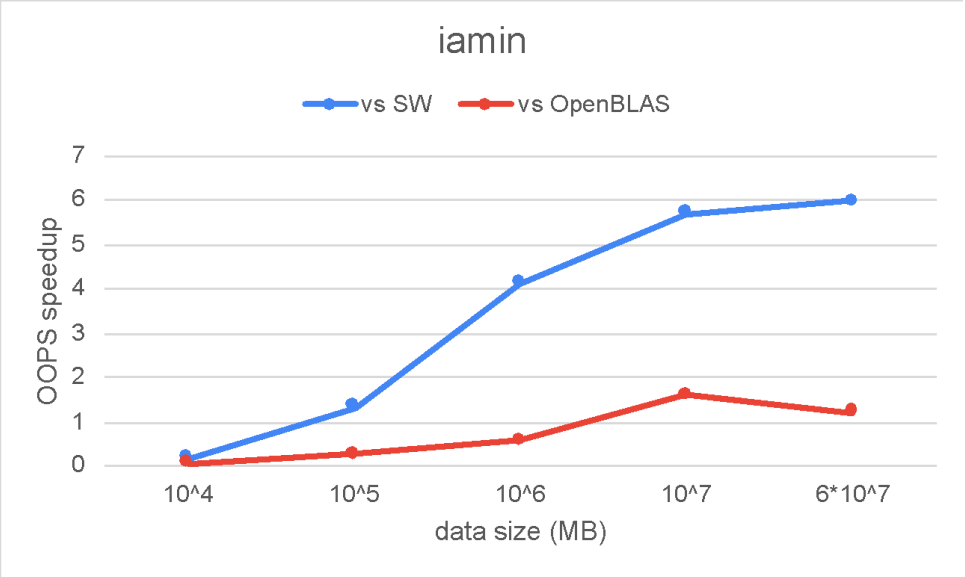
Two functions are implemented in this kernel. Both of them are operating in parallel using the dataflow directive, while data between these functions are passed by using the AXI-stream protocol. The first function implements the vectorization technique, while the second function focuses on the computational part of the kernel. The above snippet presents the latter part of the kernel.

The IAMIN function uses three basic HLS directives. Due to the vectorization technique, the unroll primitive is used in order to process 16 float or 8 double elements in parallel. Because the IAMIN kernel returns the index of the element with the min absolute value, temporal arrays are created to store the partial results obtained due to the vectorization technique. Because each cell of the array must be accessed concurrently, we integrated the array_partition primitive, which alleviates the performance bottleneck introduced from parallel accesses. Finally, our kernel uses the pipeline primitive, in order to initiate an iteration of the loop in every clock cycle.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.06	0.11	0.79	0

The chart below compares the IP performance against the designated software implementations:



A single compute unit of our kernel exceeds both the performance of an unoptimized software as well as the OpenBLAS library when a single thread is used. The chart shows that the IAMIN kernel performs worse when a small dataset is used, but the performance significantly increases as the user provides a bigger dataset. This is due to the fact our kernel is fully-pipelined, and in every clock cycle an iteration of the loop starts.

Currently, our kernel uses only one of the thirty-two available channels, or 3.125% of the available memory bandwidth. Along with the low resource consumption on the selected FPGA chip, we can instantiate multiple instances of the kernel which will provide tremendous improvements compared to the initial performance.

2.2 Level 2

2.2.1 OOPS_gemv

parameter	direction	type	description
Trans	input	char	operation to be performed (A or A^T)
M	input	int	number of rows of matrix A
N	input	int	number of columns of matrix A
KL	input	int	number of sub-diagonals of matrix A
KU	input	int	number of super-diagonals of matrix A
alpha	input	float	specifies the scalar alpha
beta	input	float	specifies the scalar beta; when beta is 0.0f then Y need not be set on input
lda	input	int	specifies the first dimension of A
&A	input	float	float array of dimension lda, N
&X	input	float	array of floats
incX	input	int	storage spacing between elements of X
&Y	input/output	float	array of floats
incY	input	int	storage spacing between elements of Y

This kernel calculates a scalar-matrix-vector product, and then adds the result to a scalar-vector product, with a general matrix.

$$Y \leftarrow alpha * A * X + beta * Y \; or \; Y \leftarrow alpha * A^T * X + beta * Y$$

The code snippet below provides its implementation based on HLS directive

```
loop_over_cols:
   for (int j=0;j<N;j+=VDATA_SIZE) {
          #pragma HLS dataflow
          wide_read_x(Xup1, Xup2, Xlow1, Xlow2, X, j/VDATA_SIZE);

          read_y(Yup1_in,Yupper1,M/4);
          read_y(Yup2_in,Yupper2,M/4);
          read_y(Ylow1_in,Ylower1,M/4);
          read_y(Ylow2_in,Ylower2,M/4);
```

```
        wide_read_matrix( Aupper1, Aup1, M/4, N/VDATA_SIZE, j/VDATA_SIZE);
        wide_read_matrix( Aupper2, Aup2, M/4, N/VDATA_SIZE, j/VDATA_SIZE);
        wide_read_matrix( Alower1, Alow1, M/4, N/VDATA_SIZE, j/VDATA_SIZE);
        wide_read_matrix( Alower2, Alow2, M/4, N/VDATA_SIZE, j/VDATA_SIZE);


        gemv(Aup1,Yup1_temp,Xup1,alpha,M/4);
        gemv(Aup2,Yup2_temp,Xup2,alpha,M/4);
        gemv(Alow1,Ylow1_temp,Xlow1,alpha,M/4);
        gemv(Alow2,Ylow2_temp,Xlow2,alpha,M/4);

        accum(Yup1_in,Yup1_temp,Yup1_out,beta,j/VDATA_SIZE,M/4);
        accum(Yup2_in,Yup2_temp,Yup2_out,beta,j/VDATA_SIZE,M/4);
        accum(Ylow1_in,Ylow1_temp,Ylow1_out,beta,j/VDATA_SIZE,M/4);
        accum(Ylow2_in,Ylow2_temp,Ylow2_out,beta,j/VDATA_SIZE,M/4);

        write_y(Yup1_out,Yupper1,M/4);
        write_y(Yup2_out,Yupper2,M/4);
        write_y(Ylow1_out,Ylower1,M/4);
        write_y(Ylow2_out,Ylower2,M/4);

}
```

The implementation of the GEMV kernel is based on a column-based approach, where we exploit the data reuse on the X vector. All functions inside the outer loop are executed with a dataflow directive. To fully exploit the dataflow implementation, a streaming interface should be implemented between functions(gemv, accum). Furthermore, functions that fetch data to streams(wide_read_x, read_y, wide_read_matrix) and write data on memory from streams (write_y) must also be implemented.

HBM memory controllers are capable of fetching 256 bits of sequential data within a single clock cycle. Meanwhile, the frequency has been set at 450 MHz. On the contrary, the running frequency of the GEMV kernel is 230 MHz, which is almost half compared to the memory controllers. So, in every cycle of the GEMV kernel, two batches of 256 bits can be fetched from the memory controllers. On wide_read_x and wide_read_matrix functions, we implement a vectorization technique which exploits the previous statements and performs burst read in order to fetch either 16 float or 8 double elements. On the other hand, due to the column-based approach, read_y and write_y perform a single float transaction with the memory, as all elements from the burst accesses correspond to a single element of the final Y vector.

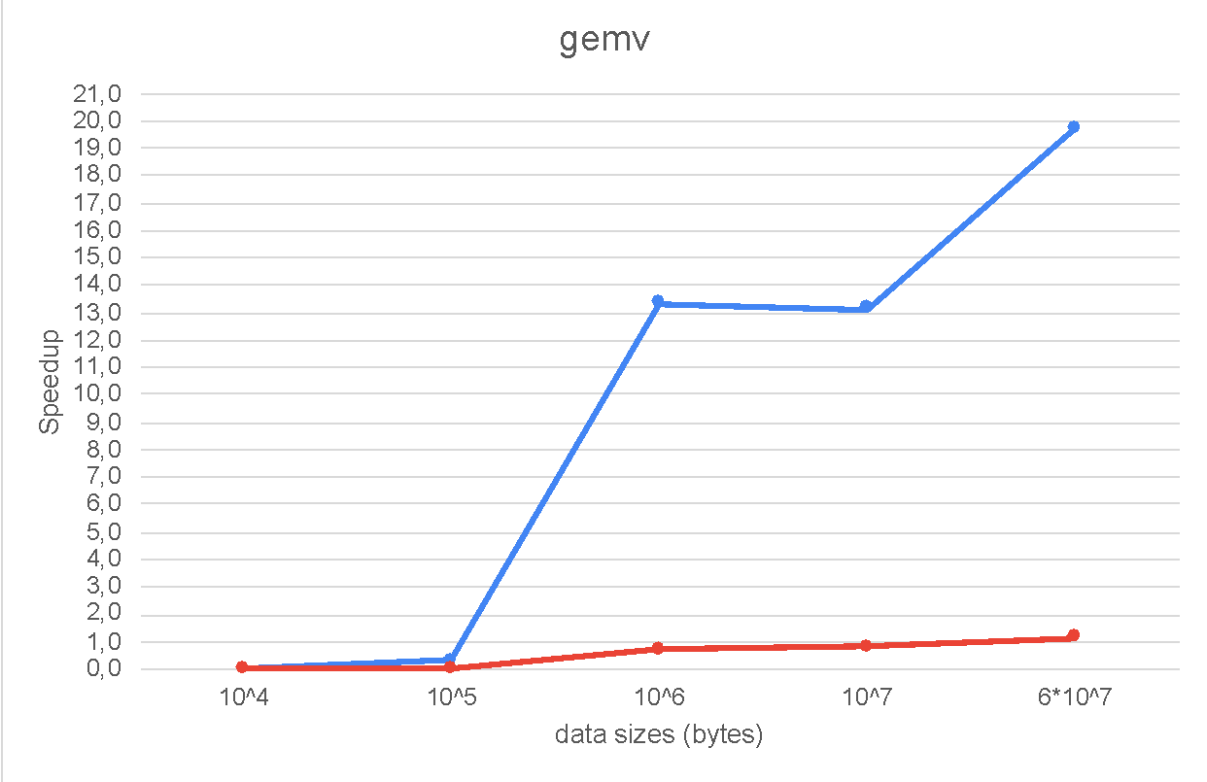
Gemv and accum functions are oriented on the computational part of the kernel. Gemv is responsible for the matrix multiplication between the matrix and vector, while accum adds the product of the previous function on the y vector. Both functions are implemented to use the pipeline primitive, in order to initiate an iteration of the inner loop in every clock cycle. Both functions achieve initiation interval equal to 1, offering the maximum possible performance of the current design.

To further exploit the data reuse on x vector, we create four instances of each function, except from wide_read_x which distributes the same elements across all instances. Each function operates on a different row of the matrix A as well as a different element of the vector Y, making them data independent.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
4.73	4.06	2.83	5.81

The chart below compares the IP performance against the designated software implementations:



A single compute unit of our kernel exceeds both the performance of an unoptimized software as well as the OpenBLAS library when a single thread is used. The chart shows that the GEMV kernel performs worse when a small dataset is used, but the performance significantly increases as the user provides a bigger dataset. This is due to the fact our kernel is fully-pipelined, and in every clock cycle an iteration of the loop starts.

Currently, our kernel uses only five of the thirty-two available channels, or 15.625% of the available memory bandwidth. Along with the low resource consumption on the selected FPGA chip, we can instantiate multiple instances of the kernel which will provide tremendous improvements compared to the initial performance.

2.2.2 OOPS_gbmv

parameter	direction	type	description
Trans	input	char	operation to be performed (A or A^T)
M	input	int	number of rows of matrix A
N	input	int	number of columns of matrix A
alpha	input	float	specifies the scalar alpha
beta	input	float	specifies the scalar beta; when beta is 0.0f then Y need not be set on input
lda	input	int	specifies the first dimension of A
&A	input	float	float array of dimension lda, N
&X	input	float	array of floats
incX	input	int	storage spacing between elements of X
&Y	input/output	float	array of floats
incY	input	int	storage spacing between elements of Y

This kernel calculates a scalar-matrix-vector product, and then adds the result to a scalar-vector product, with a general band matrix.

$$Y \leftarrow alpha * A * X + beta * Y \; or \; Y \leftarrow alpha * A^T * X + beta * Y$$

Currently, the implementation of the GBMV kernel is based on a simple software version without any significant optimizations. The AXI-Stream protocol is used as an interface between functions for data transfers. Furthermore, the pipeline primitive is used to initiate an iteration of the inner loop in every clock cycle. However, we could not achieve initiation interval equal to 1 yet, which significantly slows down the performance of our kernel. Nevertheless, the implementation of the GBMV will be based on the GEMV kernel, which already shows speedup against its software counterparts.

As a result, initial tests showed that the current implementation is slower compared to non-optimized and the OpenBLAS software counterparts. In the final section, we list our plans to improve its performance.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.3	0.43	0.35	0.2

2.2.3 OOPS_sbmv

parameter	direction	type	description
Uplo	input	char	specifies whether the upper or lower triangular part of the band matrix A is being supplied
K	input	int	specifies the number of super-diagonals of matrix A
N	input	int	number of columns of matrix A
alpha	input	float	specifies the scalar alpha
beta	input	float	specifies the scalar beta
lda	input	int	specifies the first dimension of A
&A	input	float	float array of dimension lda, N
&X	input	float	array of floats
incX	input	int	storage spacing between elements of X
&Y	input/output	float	array of floats
incY	input	int	storage spacing between elements of Y

This kernel calculates a scalar-matrix-vector product, and then adds the result to a scalar-vector product, with a symmetric band matrix.

$$Y \leftarrow alpha * A * X + beta * Y$$

Currently, the implementation of the SBMV kernel is based on a simple software version without any significant optimizations. The AXI-Stream protocol is used as an interface between functions for data transfers. Furthermore, the pipeline primitive is used to initiate an iteration of the inner loop in every clock cycle. However, we could not achieve an initiation interval equal to 1 yet, which significantly slows down the performance of our kernel. Nevertheless, the implementation of the SBMV will be based on the GEMV kernel, which already shows speedup against its software counterparts.

As a result, initial tests showed that the current implementation is slower compared to non-optimized and the OpenBLAS software counterparts. In the final section, we list our plans to improve its performance.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.38	0.57	0.64	0.24

2.2.4 OOPS_symv

parameter	direction	type	description
Uplo	input	char	specifies whether the upper or lower triangular part of the band matrix A is being supplied
N	input	int	number of columns of matrix A
alpha	input	float	specifies the scalar alpha
beta	input	float	specifies the scalar beta, when beta is zero then Y need not be set on input
lda	input	int	specifies the first dimension of A
&A	input	float	float array of dimension lda, N
&X	input	float	array of floats
incX	input	int	storage spacing between elements of X
&Y	input/output	float	array of floats
incY	input	int	storage spacing between elements of Y

This kernel calculates a scalar-matrix-vector product, and then adds the result to a scalar-vector product, with a symmetric matrix.

$$Y \leftarrow alpha * A * X + beta * Y$$

Currently, the implementation of the SYMV kernel is based on a simple software version without any significant optimizations. The AXI-Stream protocol is used as an interface between functions for data transfers. Furthermore, the pipeline primitive is used to initiate an iteration of the inner loop in every clock cycle. However, we could not achieve initiation interval equal to 1 yet, which significantly slows down the performance of our kernel. Nevertheless, the implementation of the SYMV will be based on the GEMV kernel, which already shows speedup against its software counterparts.

As a result, initial tests showed that the current implementation is slower compared to non-optimized and the OpenBLAS software counterparts. In the final section, we list our plans to improve its performance.

2.2.5 OOPS_spmv

parameter	direction	type	description
layout	input	enum	ColMajor → column-major, RowMajor → row-major format

Uplo	input	char	specifies whether the upper or lower triangular part of the band matrix A is being supplied
N	input	int	number of columns of matrix A
alpha	input	float	specifies the scalar alpha
beta	input	float	specifies the scalar beta
&AP	input	float	float array of dimension at least " ((n*(n + 1))/2) "
&X	input	float	array of floats
incX	input	int	storage spacing between elements of X
&Y	input/output	float	array of floats
incY	input	int	storage spacing between elements of Y

This kernel calculates a scalar-matrix-vector product, and then adds the result to a scalar-vector product, with a symmetric packed matrix.

$$Y \leftarrow alpha * AP * X + beta * Y$$

Currently, the implementation of the SPMV kernel is based on a simple software version without any significant optimizations. The AXI-Stream protocol is used as interface between functions for data transfers. Furthermore, the pipeline primitive is used to initiate an iteration of the inner loop in every clock cycle. However, we could not achieve initiation interval equal to 1 yet, which significantly slows down the performance of our kernel. Nevertheless, the implementation of the SPMV will be based on the GEMV kernel, which already shows speedup against its software counterparts.

As a result, initial tests showed that the current implementation is slower compared to non-optimized and the OpenBLAS software counterparts. In the final section, we list our plans to improve its performance.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.41	0.62	0.69	0.22

2.2.6 OOPS_tbsv

parameter	direction	type	description
Uplo	input	char	specifies whether the upper or lower triangular part of the band matrix A is being supplied

Trans	input	char	specifies the equation to be solved
Diag	input	char	specifies whether or not A is unit triangular
K	input	int	specifies the number of super-diagonals if Uplo is 'U', else the number of sub-diagonals
N	input	int	number of columns of matrix A
lda	input	int	specifies the first dimension of A
&A	input	float	float array of dimension lda, N
&X	input/output	float	array of floats
incX	input	int	storage spacing between elements of X

This kernel solves a system of linear equations of the following form, whose coefficients are in a triangular band matrix:

$$A * X = B \; or \; A^T * X = B$$

Currently, the implementation of the TBSV kernel is based on a simple software version without any significant optimizations. The AXI-Stream protocol is used as interface between functions for data transfers. Furthermore, the pipeline primitive is used to initiate an iteration of the inner loop in every clock cycle. However, we could not achieve initiation interval equal to 1 yet, which significantly slows down the performance of our kernel. Nevertheless, the implementation of the TBSV will be based on the GEMV kernel, which already shows speedup against its software counterparts.

As a result, initial tests showed that the current implementation is slower compared to non-optimized and the OpenBLAS software counterparts. In the final section, we list our plans to improve its performance.

2.2.7 OOPS_tbmv

parameter	direction	type	description
Uplo	input	char	specifies whether the upper or lower triangular part of the band matrix A is being supplied
Trans	input	char	specifies the equation to be solved
Diag	input	char	specifies whether or not A is unit triangular
K	input	int	specifies the number of super-diagonals if Uplo is 'U', else the number of sub-diagonals
N	input	int	number of columns of matrix A

lda	input	int	specifies the first dimension of A
&A	input	float	float array of dimension lda, N
&X	input/output	float	array of floats
incX	input	int	storage spacing between elements of X

This kernel calculates a matrix-vector product with a triangular band matrix.

$$X \leftarrow A * X \; or \; X \leftarrow A^T * X$$

Currently, the implementation of the TBMV kernel is based on a simple software version without any significant optimizations. The AXI-Stream protocol is used as interface between functions for data transfers. Furthermore, the pipeline primitive is used to initiate an iteration of the inner loop in every clock cycle. However, we could not achieve initiation interval equal to 1 yet, which significantly slows down the performance of our kernel. Nevertheless, the implementation of the TBMV will be based on the GEMV kernel, which already shows speedup against its software counterparts.

As a result, initial tests showed that the current implementation is slower compared to non-optimized and the OpenBLAS software counterparts. In the final section, we list our plans to improve its performance.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.3	0.43	0.35	0.19

2.2.8 OOPS_tpmv

parameter	direction	type	description
Uplo	input	char	specifies whether the upper or lower triangular part of the band matrix A is being supplied
Trans	input	char	specifies the equation to be solved
Diag	input	char	specifies whether or not A is unit triangular
&AP	input	float	float array of dimension at least " ((n*(n + 1))/2) "
N	input	int	number of columns of matrix A
&X	input/output	float	array of floats
incX	input	int	storage spacing between elements of X

This kernel calculates a matrix-vector product with a triangular packed matrix.

$$X \leftarrow AP * X \; or \; X \leftarrow AP^T * X$$

Currently, the implementation of the TPMV kernel is based on a simple software version without any significant optimizations. The AXI-Stream protocol is used as interface between functions for data transfers. Furthermore, the pipeline primitive is used to initiate an iteration of the inner loop in every clock cycle. However, we could not achieve initiation interval equal to 1 yet, which significantly slows down the performance of our kernel. Nevertheless, the implementation of the TPMV will be based on the GEMV kernel, which already shows speedup against its software counterparts.

As a result, initial tests showed that the current implementation is slower compared to non-optimized and the OpenBLAS software counterparts. In the final section, we list our plans to improve its performance.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.32	0.51	0.4	0.19

2.2.9 OOPS_tpsv

parameter	direction	type	description
Uplo	input	char	specifies whether the upper or lower triangular part of the band matrix A is being supplied
Trans	input	char	specifies the equation to be solved
Diag	input	char	specifies whether or not A is unit triangular
&AP	input	float	float array of dimension at least " ((n*(n + 1))/2) "
N	input	int	number of columns of matrix A
&X	input/output	float	array of floats
incX	input	int	storage spacing between elements of X

This kernel calculates a system of linear equations of the following form, whose coefficients are in a triangular packed matrix.

$$AP * X = B \; or \; AP^T * X = B$$

Currently, the implementation of the TPSV kernel is based on a simple software version without any significant optimizations. The AXI-Stream protocol is used as interface between functions for data transfers. Furthermore, the pipeline primitive is used to initiate an iteration of the inner loop in every clock cycle. However, we could not achieve initiation interval equal to 1 yet, which significantly slows down the performance of our kernel. Nevertheless, the implementation of the TPSV will be based on the GEMV kernel, which already shows speedup against its software counterparts.

As a result, initial tests showed that the current implementation is slower compared to non-optimized and the OpenBLAS software counterparts. In the final section, we list our plans to improve its performance.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.35	0.47	0.35	0.09

2.2.10 OOPS_trmv

parameter	direction	type	description
Uplo	input	char	specifies whether the upper or lower triangular part of the band matrix A is being supplied
Trans	input	char	specifies the equation to be solved
Diag	input	char	specifies whether or not A is unit triangular
N	input	int	number of columns of matrix A
lda	input	int	specifies the first dimension of A
&A	input	float	float array of dimension lda, N
&X	input/output	float	array of floats
incX	input	int	storage spacing between elements of X

This kernel calculates a matrix-vector product with a triangular matrix.

$$X \leftarrow A * X \; or \; X \leftarrow A^T * X$$

Currently, the implementation of the TRMV kernel is based on a simple software version without any significant optimizations. The AXI-Stream protocol is used as interface between functions for data transfers. Furthermore, the pipeline primitive is used to initiate an iteration of the inner loop in every clock cycle. However, we could not achieve initiation interval equal to 1 yet, which significantly

slows down the performance of our kernel. Nevertheless, the implementation of the TRMV will be based on the GEMV kernel, which already shows speedup against its software counterparts.

As a result, initial tests showed that the current implementation is slower compared to non-optimized and the OpenBLAS software counterparts. In the final section, we list our plans to improve its performance.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.77	1.13	0.15	1.81

2.2.11 OOPS_trsv

parameter	direction	type	description
Uplo	input	char	specifies whether the upper or lower triangular part of the band matrix A is being supplied
Trans	input	char	specifies the equation to be solved
Diag	input	char	specifies whether or not A is unit triangular
N	input	int	number of columns of matrix A
lda	input	int	specifies the first dimension of A
&A	input	float	float array of dimension lda, N
&X	input/output	float	array of floats
incX	input	int	storage spacing between elements of X

This kernel calculates a matrix-vector product with a triangular band matrix.

$$X \leftarrow A * X \; or \; X \leftarrow A^T * X$$

Currently, the implementation of the TRSV kernel is based on a simple software version without any significant optimizations. The AXI-Stream protocol is used as interface between functions for data transfers. Furthermore, the pipeline primitive is used to initiate an iteration of the inner loop in every clock cycle. However, we could not achieve initiation interval equal to 1 yet, which significantly slows down the performance of our kernel. Nevertheless, the implementation of the TRSV will be based on the GEMV kernel, which already shows speedup against its software counterparts.

As a result, initial tests showed that the current implementation is slower compared to non-optimized and the OpenBLAS software counterparts. In the final section, we list our plans to improve its performance.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.27	0.42	0.05	0.19

2.3 Level 3

2.3.1 OOPS_gemm

parameter	direction	type	description
TransA	input	char	specifies the form of operation A to be used in the matrix multiplication
TransB	input	char	Specifies the form of operation B to be used in the matrix multiplication
M	input	int	specifies the number of rows of the matrix operation A and of the matrix C
N	input	int	specifies the number of columns of the matrix operation B and of the matrix C
K	input	int	specifies the number of columns of the matrix operation A and the number of rows of the matrix operation B
alpha	input	float	specifies the scalar alpha
beta	input	float	specifies the scalar beta; when beta is 0.0f then C need not be set on input
lda	input	int	specifies the first dimension of A
&A	input	float	float array of dimension lda, K or M
ldb	input	int	specifies the first dimension of B
&B	input	float	float array of dimension ldb, N or K
ldc	input	int	specifies the first dimension of C
&C	input/output	float	float array of dimension ldc, N

This kernel calculates a scalar-matrix-matrix product, and then adds the result to a scalar-matrix product, with general matrices.

$$C \leftarrow alpha * op(A) * op(B) + beta * C$$, where op(X) = X or op(X) = X^T

The code snippet below provides its implementation based on HLS directives.

```
loop_over_CrowsNN:
  for (int j=0;j<M;j++){
    #pragma HLS dataflow
    wide_read_memC((v_dt*)C,CinNN,j*N,(j+1)*N);
    loop_over_CcolumnsNN:
    for (int i=0;i<vSize;i++){
      #pragma HLS dataflow
      wide_read_memA(A,AinNN,j*K,(j+1)*K,1);
      wide_read_memB((v_dt*)B,BinNN,i,vSize*K+i,vSize);
      gemm(alpha,beta,AinNN,BinNN,CinNN,CoutNN,K);
    }
  wide_write_mem((v_dt*)C,CoutNN,j*N,(j+1)*N);
}
```

The implementation of the GEMM kernel is based on HLS primitives and techniques, already discussed during the previous kernels.

In order to fully exploit the data width of the HBM channels, we implement a vectorization technique where data is fetched in batches. Memory controllers are capable of fetching 256 bits of sequential data every clock cycle, while their frequency has been set on 450 MHz. The frequency of the GEMM kernel is 250 MHz, so in every clock cycle almost two memory transactions can be accomplished through memory controllers.

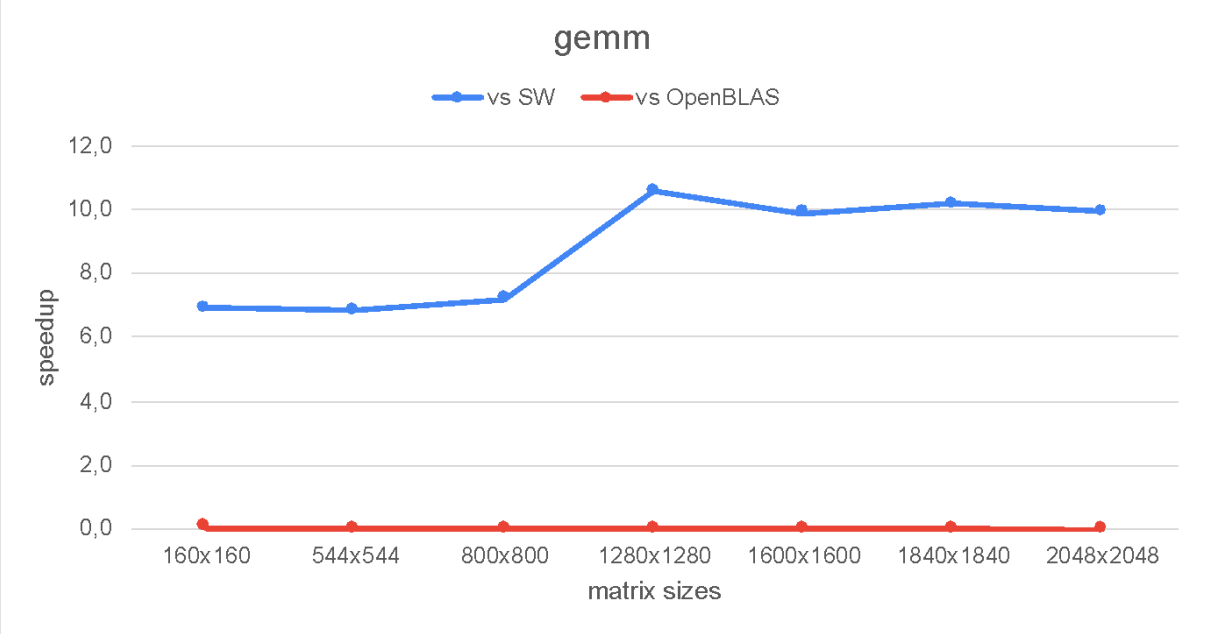
The dataflow primitive is used on both the outer loop (loop_over_CrowsNN) and second loop (loop_over_CcolumnsNN) in order to let the different functions execute in parallel. To fully exploit the dataflow primitive, a streaming interface should be implemented between the functions, as well functions that fetch data to streams (wide_read_mem) and write data on memory from streams (wide_write_mem).

The gemm function calculates the scalar-matrix-matrix product, and then adds the result to a scalar-matrix product. This function implements the pipeline primitive, in order to initiate an iteration of the inner loop in every clock cycle. However, the loop does not achieve an initiation interval equal to 1, which significantly slows down the performance.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
1.02	1.17	1.09	0.76

The chart below compares the IP performance against the designated software implementations:



Our initial results show a significant speedup over a naive implementation on software. Furthermore, by providing a bigger dataset, the performance of our kernel keeps increasing due to the pipeline primitive on our functions. On the other hand, our kernel could not keep at this moment in terms of raw performance with an optimized library like OpenBLAS. We figure out that the bottleneck comes from our approach in the kernel's design. In the final section, we list our plans to improve its performance.

2.3.2 OOPS_symm

parameter	direction	type	description
Side	input	char	specifies whether the symmetric matrix A appears on the left or right in the operation
Uplo	input	char	specifies whether the symmetric matrix A is an upper or lower triangular
M	input	int	specifies the number of rows of of the matrix C
N	input	int	specifies the number of columns of the matrix C
alpha	input	float	specifies the scalar alpha
beta	input	float	specifies the scalar beta; when beta is 0.0f then C need not be set on input
lda	input	int	specifies the first dimension of A
&A	input	float	float array of dimension lda, M or N

ldb	input	int	specifies the first dimension of B
&B	input	float	float array of dimension ldb, N
ldc	input	int	specifies the first dimension of C
&C	input/output	float	float array of dimension ldc, N

This kernel calculates a scalar-matrix-matrix product, and then adds the result to a scalar-matrix product, where one of the matrices in the multiplication is symmetric.

If Side='L', then it performs

$$C \leftarrow alpha * A * B + beta * C$$

Else

$$C \leftarrow alpha * B * A + beta * C$$

The code snippet below provides its implementation based on HLS directives.

```
loop_over_CrowsLU:
  for (int j=0;j<M;j++){
    #pragma HLS dataflow
    wide_read_memC((v_dt*)C,wide_Cin,j*N,(j+1)*N);
    loop_over_CcolumnsLU:
    for (int i=0;i<N;i+=VDATA_SIZE){
      #pragma HLS dataflow
      wide_read_memA(A,wide_Ain,j,M);
      wide_read_memB((v_dt*)B,wide_Bin,i/VDATA_SIZE,(M*N+i)/VDATA_SIZE,M/VDATA_SIZE);
      wide_symm(alpha, beta,wide_Ain,wide_Bin,wide_Cin,wide_Cout,M);
    }
    wide_write_memC((v_dt*)C,wide_Cout,j*N,(j+1)*N);
  }
```

The implementation of the SYMM kernel is based on HLS primitives and techniques, already discussed during the previous kernels.

In order to fully exploit the data width of the HBM channels, we implement a vectorization technique where data is fetched in batches. Memory controllers are capable of fetching 256 bits of sequential data every clock cycle, while their frequency has been set on 450 MHz. The frequency of the SYMM kernel is 250 MHz, so in every clock cycle almost two memory transactions can be accomplished through memory controllers.

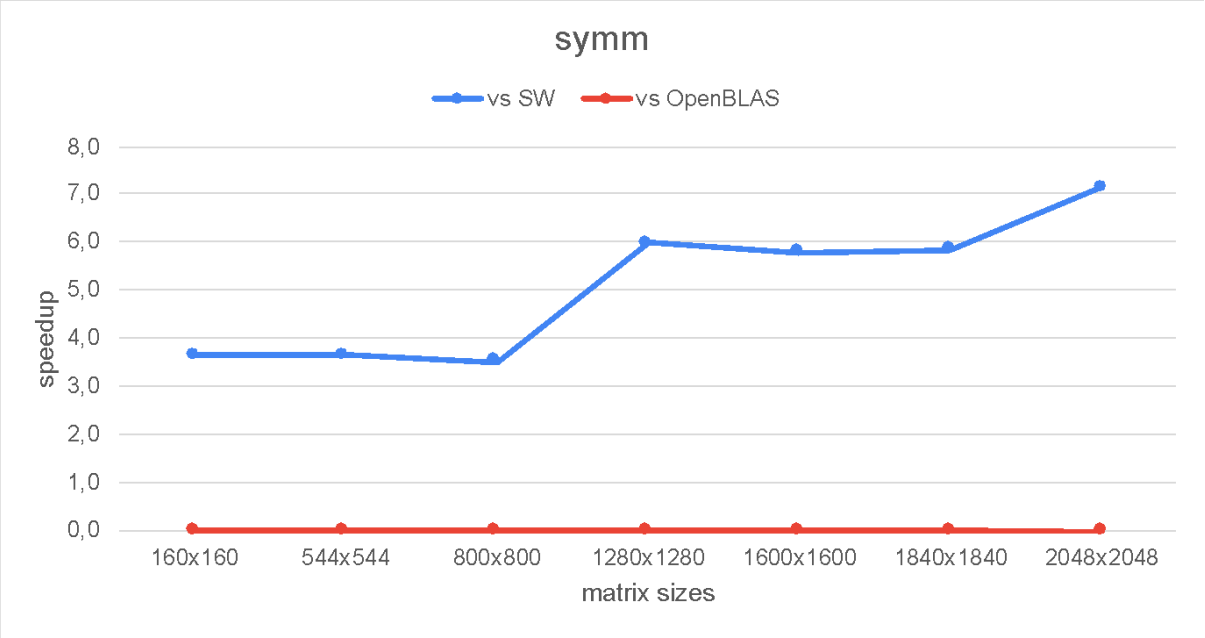
The dataflow primitive is used on both the outer loop (loop_over_CrowsLU) and second loop (loop_over_CcolumnsLU) in order to let the different functions execute in parallel. To fully exploit the dataflow primitive, a streaming interface should be implemented between the functions, as well functions that fetch data to streams (wide_read_mem) and write data on memory from streams (wide_write_mem).

The wide_symm function calculates the scalar-matrix-matrix product, and then adds the result to a scalar-matrix product. This function implements the pipeline primitive, in order to initiate an iteration of the inner loop in every clock cycle. However, the loop does not achieve initiation interval equal to 1, which significantly slows down the performance.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
1	1.14	1.14	0.81

The chart below compares the IP performance against the designated software implementations:



Our initial results show a significant speedup over a naive implementation on software. Furthermore, by providing a bigger dataset, the performance of our kernel keeps increasing due to the pipeline primitive on our functions. On the other hand, our kernel could not keep at this moment in terms of raw performance with an optimized library like OpenBLAS. We figure out that the bottleneck comes from our approach in the kernel's design. In the final section, we list our plans to improve its performance.

2.3.3 OOPS_trmm

parameter	direction	type	description
Side	input	char	specifies whether the symmetric matrix A appears on the left or right in the operation

Uplo	input	char	specifies whether the symmetric matrix A is an upper or lower triangular
TransA	input	char	specifies the form of operation A to be used in the matrix multiplication
Diag	input	char	specifies whether or not A is unit triangular
M	input	int	specifies the number of columns of the matrix B
N	input	int	specifies the number of rows of the matrix B
alpha	input	float	specifies the scalar alpha
lda	input	int	specifies the first dimension of A
&A	input	float	float array of dimension lda, M or N
ldb	input	int	specifies the first dimension of B
&B	input/output	float	float array of dimension ldb, N

This kernel calculates a scalar-matrix-matrix product where one of the matrices in the multiplication is triangular.

If Side='L', then it performs

$$B \leftarrow alpha * op(A) * B$$

Else

$$B \leftarrow alpha * B * op(A)$$, where op(X) = X or op(X) = X^T

The code snippet below provides its implementation based on HLS directives.

```
loop_over_CrowsNN:
  for (int j=0;j<M;j++){
    offset = j;

    loop_over_Ccolumns_wide:
    for (int i=offset+((N-j)%VDATA_SIZE);i<(N-VDATA_SIZE+1);i+=VDATA_SIZE){
      #pragma HLS dataflow
      wide_read_memA(A,wide_AinNN,j*M+j,(j+1)*M,1);
     wide_read_memB((v_dt*)B,wide_BinNN,(j*N+i)/VDATA_SIZE,(M*N+i)/VDATA_SIZE,M/
VDATA_SIZE);
      wide_trmm(j,alpha,wide_AinNN,wide_BinNN,wide_BoutNN,M);
      wide_write_memC((v_dt*)C,wide_BoutNN,(j*M+i)/VDATA_SIZE);
    }

loop_over_Ccolumns_normal:
  for (int k=offset;k<offset+((N-j)%VDATA_SIZE);k++){
    #pragma HLS dataflow
    read_memA(A,AinNN,j*M+j,(j+1)*M,1);
```

```
    read_memB(B,BinNN,j*N+k,M*N+k,M);
    trmm(j,alpha,AinNN,BinNN,BoutNN,M);
    write_memC(C,BoutNN,j*N+k);
  }
}
```

The implementation of the TRMM kernel is based on HLS primitives and techniques, already discussed during the previous kernels.

In order to fully exploit the data width of the HBM channels, we implement a vectorization technique where data are fetched on batches. Memory controllers are capable to fetch 256 bits of sequential data every clock cycle, while their frequency has been set on 450 MHz. The frequency of the TRMM kernel is 250 MHz, so in every clock cycle almost two memory transactions can be accomplished through memory controllers. However, because each row of matrix A contains a different number of elements, a separate for loop (loop_over_Ccolumns_normal) must be implemented in order to capture the cases where vectorization cannot be applied, due to the imbalance between the rows.

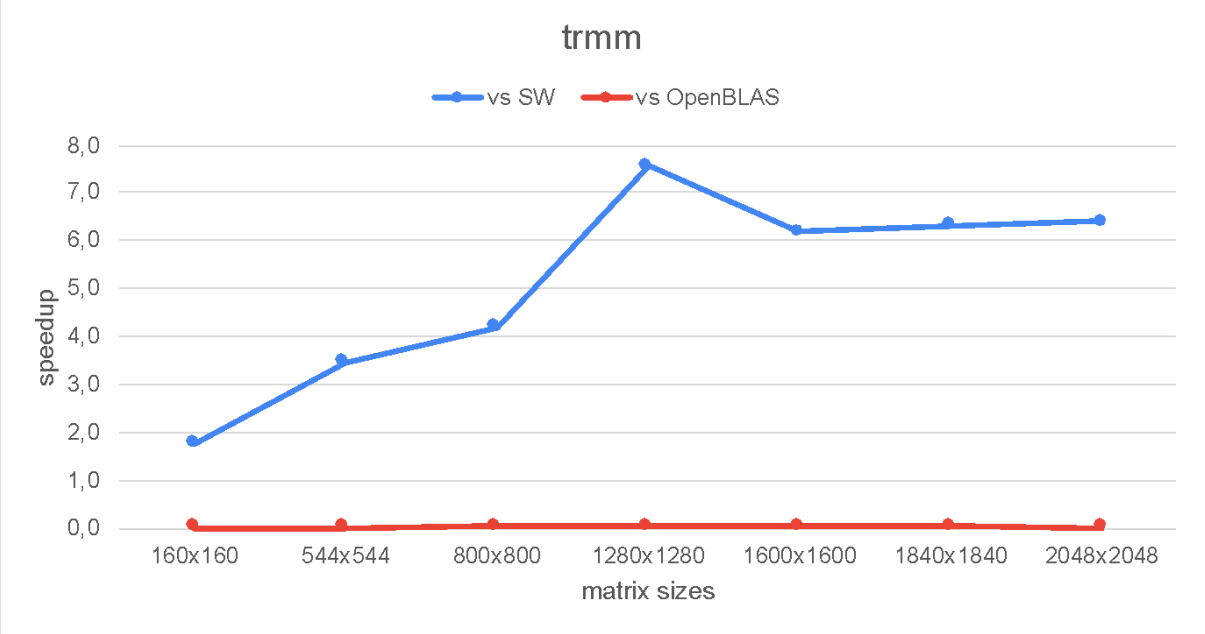
The dataflow primitive is used on the loop of each previously described case, (loop_over_Ccolumns_wide & loop_over_Ccolumns_normal) in order to let the different functions execute in parallel. To fully exploit the dataflow primitive, a streaming interface should be implemented between the functions, as well functions that fetch data to streams (wide_read_mem & read_mem) and write data on memory from streams (wide_write_mem & write_mem).

The wide_trmm and trmm functions calculate the scalar-matrix-matrix product, when vectorization can be applied and when not respectively. Both functions implement the pipeline primitive, in order to initiate an iteration of the inner loop in every clock cycle. However, the loop does not achieve initiation interval equal to 1, which significantly slows down the performance.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.85	1.09	1.09	0.53

The chart below compares the IP performance against the designated software implementations:



Our initial results show a significant speedup over a naive implementation on software. Furthermore, by providing a bigger dataset, the performance of our kernel keeps increasing due to the pipeline primitive on our functions. On the other hand, our kernel could not keep at this moment in terms of raw performance with an optimized library like OpenBLAS. We figure out that the bottleneck comes from our approach in the kernel's design. In the final section, we list our plans to improve its performance.

2.3.4 OOPS_trsm

parameter	direction	type	description
Side	input	char	specifies whether the symmetric matrix A appears on the left or right in the operation
Uplo	input	char	specifies whether the symmetric matrix A is an upper or lower triangular
TransA	input	char	specifies the form of operation A to be used in the matrix multiplication
Diag	input	char	specifies whether or not A is unit triangular
M	input	int	specifies the number of columns of the matrix B
N	input	int	specifies the number of rows of the matrix B
alpha	input	float	specifies the scalar alpha
lda	input	int	specifies the first dimension of A
&A	input	float	float array of dimension lda, M or N

ldb	input	int	specifies the first dimension of B
&B	input/output	float	float array of dimension ldb, N

This kernel calculates one of the following matrix equations.

$op(A) * X = alpha * B$ or $X * op(A) = alpha * B$, where op(X) = X or op(X) = X^T

The code snippet below provides its implementation based on HLS directives.

```
loop_over_CrowsNN:
  for (int j=0;j<M;j++){
    offset = j;

    loop_over_Ccolumns_wide:
    for (int i=offset+((N-j)%VDATA_SIZE);i<(N-VDATA_SIZE+1);i+=VDATA_SIZE){
    #pragma HLS dataflow
    wide_read_memA(A,wide_AinNN,j*M+j,(j+1)*M,1);
    wide_read_memB((v_dt*)B,wide_BinNN,(j*N+i)/VDATA_SIZE,(M*N+i)/VDATA_SIZE,M/
VDATA_SIZE);
    wide_trsm(j,alpha,wide_AinNN,wide_BinNN,wide_BoutNN,M);
    wide_write_memC((v_dt*)C,wide_BoutNN,(j*M+i)/VDATA_SIZE);
}

 loop_over_Ccolumns_normal:
  for (int k=offset;k<offset+((N-j)%VDATA_SIZE);k++){
    #pragma HLS dataflow
    read_memA(A,AinNN,j*M+j,(j+1)*M,1);
    read_memB(B,BinNN,j*N+k,M*N+k,M);
    trsm(j,alpha,AinNN,BinNN,BoutNN,M);
    write_memC(C,BoutNN,j*N+k);
  }
}
```

The implementation of the TRSM kernel is based on HLS primitives and techniques, already discussed during the previous kernels.

In order to fully exploit the data width of the HBM channels, we implement a vectorization technique where data is fetched in batches. Memory controllers are capable of fetching 256 bits of sequential data every clock cycle, while their frequency has been set on 450 MHz. The frequency of the TRSM kernel is 250 MHz, so in every clock cycle almost two memory transactions can be accomplished through memory controllers. However, because each row of matrix A contains a different number of elements, a separate for loop (loop_over_Ccolumns_normal) must be implemented in order to capture the cases where vectorization cannot be applied, due to the imbalance between the rows.

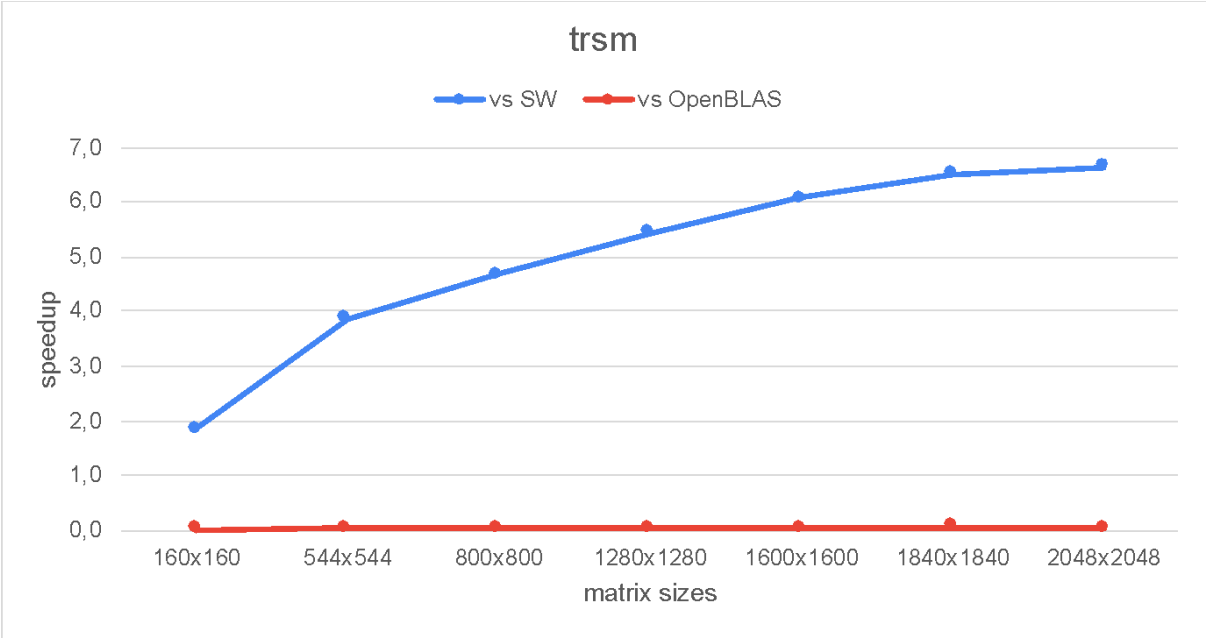
The dataflow primitive is used on the loop of each previously described case, (loop_over_Ccolumns_wide & loop_over_Ccolumns_normal) in order to let the different functions execute in parallel. To fully exploit the dataflow primitive, a streaming interface should be implemented between the functions, as well functions that fetch data to streams (wide_read_mem & read_mem) and write data on memory from streams (wide_write_mem, write_mem).

The wide_trsm and trsm functions calculate the scalar-matrix-matrix product, when vectorization can be applied and when not respectively. Both functions implement the pipeline primitive, in order to initiate an iteration of the inner loop in every clock cycle. However, the loop does not achieve initiation interval equal to 1, which significantly slows down the performance.

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
0.95	1.36	1.09	0.4

The chart below compares the IP performance against the designated software implementations:



Our initial results show a significant speedup over a naive implementation on software. Furthermore, by providing a bigger dataset, the performance of our kernel keeps increasing due to the pipeline primitive on our functions. On the other hand, our kernel could not keep at this moment in terms of raw performance with an optimized library like OpenBLAS. We figure out that the bottleneck comes from our approach in the kernel's design. In the final section, we list our plans to improve its performance.

3. Sparse Matrix - Vector (SpMV) kernel

As described in D2.2, the OOPS library will provide a hardware accelerated SpMV [5] implementation, since it is a widely used operation in many application domains, such as numerical analysis, graphics, graphs, and conjugate gradients.

The current implementation supports CSR representations; the table below lists the input/output parameters:

parameter	direction	type	description
row_ptr&	input	int	vector containing the index of the first non-zero entry of each row in the value vector
col_ind&	input	Int	vector containing column index of each non zero entry
values&	input	float	vector containing all non-zero entries
num_rows	input	int	number of rows of the matrix
nnz	input	int	number of non-zero values
x&	input	float	input vector
y&	input	float	output vector

In both cases, the kernel calculates the matrix-vector product between matrix A and vector x, and the output is stored in vector y:

$$y \leftarrow A * x$$

```
void spmv_csr(hls::stream<double>&values, hls::stream<int>&col_ind, hls::stream<int> &row_ptr,
double **x, hls::stream<double> &b, int srow, int num_rows ) {
execute:
  int m1,m2;
  double sum;
  int col;
  double data;
  double temp;
  double temp2;
  int limit;
  double sum_p[FADD_LAT];
  int step;
  #pragma HLS ARRAY_PARTITION variable=sum_p complete dim=1
  m2=row_ptr.read();

  for (int i = srow; i < (srow+num_rows); i++) {
    m1 = m2;
    m2 = row_ptr.read();
    sum=0;
    loop_init: for(int k=0;k<FADD_LAT;k++) {
```

```
        #pragma HLS UNROLL
          sum_p[k] = 0;
    }
    step = ((FADD_LAT) >= m2) ? m2 : FADD_LAT;
    LOOP: for( int j = m1; j < m2; j+=FADD_LAT) {
        #pragma HLS PIPELINE II=5 rewind
          step = ((j+FADD_LAT) >= m2) ? (m2-j) : FADD_LAT;
          int overflow=0;
          for (int k=0; k<FADD_LAT; k++){
              #pragma HLS UNROLL
                col = col_ind.read();
                data = values.read();
                temp = data;
                temp2 = x[k][col];
                sum_p[k]+=temp*temp2;
                overflow++;
                if(overflow == step) break;
      }
    }

  loop_sum_f: for (int k=0; k<FADD_LAT; k++)
  {
            #pragma HLS UNROLL
          sum += sum_p[k];
  }
     y << sum;
  }
}
```

The code snippet above provides its implementation based on HLS directives. SpMV is a memory-bound algorithm, and its main bottleneck is the random-access pattern on the x input vector. Therefore, the available memory bandwidth is the main performance factor for this HPC kernel. Also, it is commonly applied on double precision floating point data, thus our implementation focuses on this setup. However, both factors, effective bandwidth and double precision arithmetic, make FPGA design challenging. FPGA platforms are commonly BW-constrained compared to CPUs or GPUs and they lack dedicated DSPs for double precision operations. For both performance challenges we apply the following optimizations:

- To maximize the utilization of the HBM bandwidth available on our platform, we stream all input and output data apart from the x vector, we apply dataflow operations, and we use multiple compute units. Each compute unit operates on a subset of rows of the entire problem (partition), and each partition is placed on a separate HBM bank. Each compute unit is connected to a replica of the x vector placed on a dedicated bank for higher BW.
- To cope with the high fp operation latency (especially the accumulation on the inner loop that has a carried dependency), we use loop unrolling and a shift register.

One of the challenges of scaling to multiple compute units is the memory port limitation on Alveo cards. The design can have up to 32 ports to HBM. Thus, to scale up to 16 compute units, we use 2 ports per unit; values and col_ind matrices are streamed sequentially with iteration interval equal to 2 while the inner main loop of the spmv operation has iteration interval equal to 1.

The following table provides the IP utilization resources on the selected FPGA chip:

Flip-Flop (%)	LUTs (%)	BRAMs (%)	DSPs (%)
3	7	9.7	1

The chart below compares the IP performance against a simple software implementation running on a single core and with various other FPGA implementations that we experimented with, that also expose to some extent the design restrictions and trade-offs in-hand. Table 1 summarizes the descriptions of our in-house FPGA implementations. Table 2 summarizes the sizes of the sparse matrices that we used for testing; provided by M3E and holding real data.

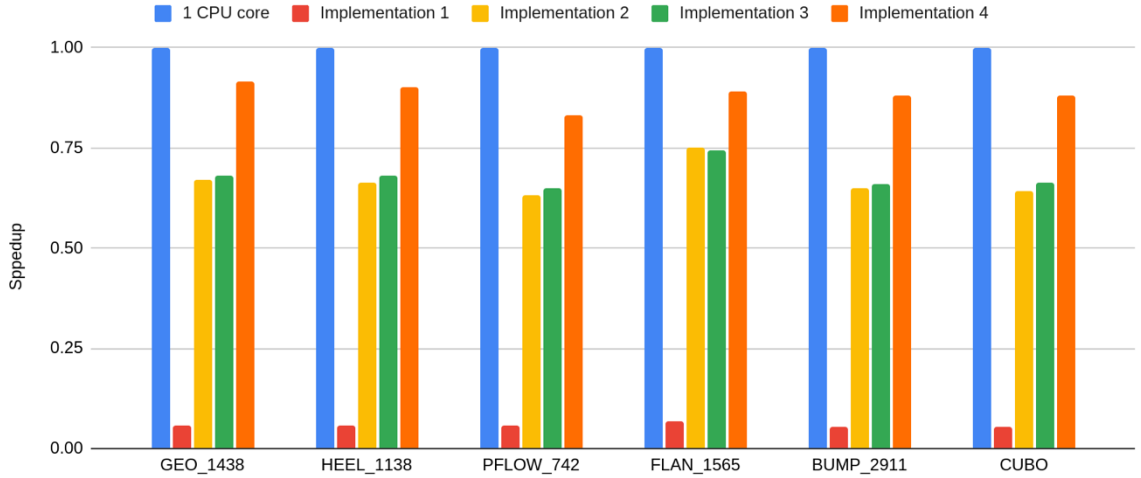
Table1: Various in-house FPGA implementations

	Description	#CUs	Max Problem Size
Implementation 1	Simple implementation using streams for all spmv kernel input/output (col_ind, values, row_ptr, x, y) and using DDR to store the values and col_ind large matrices (to be able to support arbitrarily large problem sizes). The x vector that is accessed irregularly (main bottleneck) is placed on an HBM bank, for higher bandwidth.	1	nnz: 2147483648 num_rows: 33554432
Implementation 2	Same as before, but with multiple CUs. Each CU is in charge of a partition of the problem (a set of rows) and each CU has its own x vector replica, on a local HBM bank, for higher bandwidth. As DDT can serve up to 15 consumers, the design can scale up to 15 units.	15	nnz: 2147483648 num_rows: 33554432
Implementation 3	Same as before but also values and col_ind in HBM banks. To scale up to 16 CUs, as the FPGA card can only support up to 32 memory ports to HBM, we place values and col_ind partitions (per CU) at the same bank and stream them consecutively and not in parallel. (Per CU we use 1 shared port for values and col_ind and 1 shared port for x, row_ptr and y.)	16	nnz: 536870912 num_rows: 33554432
Implementation 4 (OOPS_spmv)	Same as implementation 3 but we also use a shift register and an unrolling of depth 5 to perform the double precision float accumulation on the inner loop, to achieve iteration interval equal to 1. (The latency of a double precision float accumulation is 5 cycles)	16	nnz: 536870912 num_rows: 33554432

Table2: Sparse Matrices

Matrix Name	NNZ	NUM ROWS	Values matrix size (MB)
GEO_1438	63,156,690	1,437,960	481.8473053

HEEL_1138	63,156,690	1,138,443	394.2713699
PFLOW_742	37,138,461	742,793	283.3439713
FLAN_1565	117,406,044	1,564,794	895.73703
BUMP_2911	130,378,257	2,911,419	974.5
CUBE_5M	222,615,369	5,317,443	1698.420479



For FPGA implementations, we observe that scaling to multiple CUs significantly improves performance and shift registers give an extra 40% boost. However, still, the IP performs 10-17% worse than a single CPU core. The main reason is memory BW and irregular x vector access. We will optimize further our IP to exploit better the platform's available memory BW for x.

4. General Computer-Aided Engineering (CAE) solvers

This kernel computes LU decomposition [2] [3] [4]. LU decomposition is the process of creating two matrices, L and U such that the product of those two matrices is the original matrix A. This kernel particularly is important because it can be used to find the determinant of a matrix and speed up solving linear systems and other important calculations involving matrices in computer science.

The figure below shows a simple example of how matrix A is decomposed into lower (L) and upper (U) matrices:

A		L		U
1 1 0 2 1 3 3 1 1	=	1 0 0 2 -1 0 3 -2 -5	*	1 1 0 0 1 -3 0 0 1

The table below lists the input/output parameters:

parameter	direction	type	description
*A	input	float	Input matrix A
*L	output	float	Lower triangular matrix
*U	output	float	Upper triangular matrix
n	input	int	Size of the matrix

The code snippet below provides the first version of the kernel implementation.

```
extern "C"
{
   void krnl_lu(float *A, float *L, float *U, int n)
   {

#pragma HLS INTERFACE m_axi port = A offset = slave bundle = ddr0
#pragma HLS INTERFACE m_axi port = L offset = slave bundle = ddr0
#pragma HLS INTERFACE m_axi port = U offset = slave bundle = ddr0
#pragma HLS INTERFACE s_axilite port = n

      int i = 0, j = 0, k = 0;

      float pixel; // hold data locally so we don't access memory too much
   FOR_I:
      for (i = 0; i < n; i++)
      {
      FOR_JL:
         for (j = 0; j < n; j++)
```

```
        {
          if (j < i)
            L[j * n + i] = 0;
          else
          {
            pixel = A[j * n + i];
          FOR_KL:
            for (k = 0; k < i; k++)
            {
              pixel = pixel - L[j * n + k] * U[k * n + i];
            }
            L[j * n + i] = pixel;
          }
        }
      FOR_JU:
        for (j = 0; j < n; j++)
        {
          if (j < i)
            U[i * n + j] = 0;
          else if (j == i)
            U[i * n + j] = 1;
          else
          {
            pixel = A[i * n + j] / L[i * n + i];
          FOR_KU:
            for (k = 0; k < i; k++)
            {
              pixel = pixel - ((L[i * n + k] * U[k * n + j]) / L[i * n + i]);
            }
            U[i * n + j] = pixel;
          }
        }
      }
    }
}
```

The following table provides the IP utilization resources on the selected FPGA chip:

Registers (%)	Registers	LUTs (%)	LUT	BRAMs (%)	BRAM	DSPs (%)	DSP
~0	4863	~0	7191	~0	2	~0	5

The current implementation provides a simple proof-of-concept that successfully maps the kernel onto the hardware. However, since it does not utilize any HLS directives (e.g. pipelining, loop-unrolling, dataflow), initial tests showed that the current implementation is slower compared to non-optimized software versions. In the final section, we list our plans to improve its performance.

5. Summary and Next steps

The OOPS library 1.0 provides a large set of hardware accelerated kernels for sparse and dense linear algebra calculations, as well as CAE solvers that outperform single threaded software optimized versions. For convenience, the original task description is provided below:

> *In this first increment of the library development we will focus on establishing a common infrastructure for the low-level accelerated functions. To achieve this we will first focus on providing a concise and performant set of lower-level libraries and functions (such as BLAS, solvers, SpMV, etc). In parallel we will profile the more complex open source libraries and identify their key compute kernels to accelerate and begin their development. In this approach we will strive to re-use as many lower-level functions as possible.*

As discussed in the previous chapters, the current OOPS library status meets and even surpasses the D3.5 requirements:

- BLAS L1: All kernels already outperform optimized software implementations, even with very low resource utilization, paving the way for even performant versions based on instantiating additional CUs.
- BLAS L2: As demonstrated by the gemv kernel, a fine-tuned implementation that utilizes very few resources (less than 3%) can outperform optimized single threaded software versions. The plan is to apply the same approach (as in gemv)to all L2 kernels.
- BLAS L3: All kernels outperform non-optimized software implementations even when occupying less than 2% of the available resources. Towards enhancing performance, our plan is to instantiate additional CUs.
- SpMV: The current implementation is 10%-17% compared to software versions. It should be noted that performance is strongly related to the matrix sparsity, since denser matrices enable higher speedups.
- LU decomposition: Current version is not optimized at all and is significantly slower than CPU version. Next steps involve adding HLS directives and optimizations.

Towards more performant implementations, we plan to explore the following paths:

- Device specific implementations: Towards enhancing performance, our plan is to provide device specific kernel implementations that will leverage available technologies. For example, kernels should be able to utilize High-Bandwidth Memory (HBM) when available, to accelerate memory access, or instantiate as many compute units as possible that can fit in the reconfigurable area.
- More CAE solvers: The OOPS library will support hardware accelerated implementations of the Jacobi preconditioner and Krylov Conjugate Gradient (CG) algorithm.
- Utilize more CUs / kernel when possible: As described, many kernels utilize only a fraction of the available reconfigurable area, leaving room for massive parallel data processing with the instantiation of additional CUs.
- Use of UltraRAM: UltraRAM is a memory block in Ultrascale+ FPGA chips that enables faster on-chip transfer rates.
- Public repository: The OOPS library will be available to developers from a public repository.

6. References

[1] Lawson, C. & Hanson, Richard & Kincaid, David & Krogh, Fred. (1979). Basic linear algebra subprograms for FORTRAN usage. ACM Trans. Math. Soft. 5. 308-323. 10.1145/355841.355847.

[2] Ignacio Bravo, César Vázquez, Alfredo Gardel, José L. Lázaro, Esther Palomar, "High Level Synthesis FPGA Implementation of the Jacobi Algorithm to Solve the Eigen Problem", Mathematical Problems in Engineering, vol. 2015, Article ID 870569, 11 pages, 2015.

[3] 21. J. Luo, Q. Huang, S. Chang, X. Song and Y. Shang, "High throughput Cholesky decomposition based on FPGA," 2013 6th International Congress on Image and Signal Processing (CISP), 2013, pp. 1649-1653, doi: 10.1109/CISP.2013.6743941.

[4] 22. Altera Corporation, “Cholesky Solver Reference Design Datasheet”, 2014

[5] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis and N. Koziris, "Understanding the Performance of Sparse Matrix-Vector Multiplication," *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, 2008, pp. 283-292, doi: 10.1109/PDP.2008.41.

[6] G. Perna et al., “Delverable D2.2 – Applications and Systems pairing”, 2021

[7] https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html last visit June 2022