Pragmas and Data Motion Networks

Introduction

This lab guides you through the process of handling data transfers between the software and hardware accelerators using various pragmas and the SDSoC API.

Objectives

After completing this lab, you will be able to:

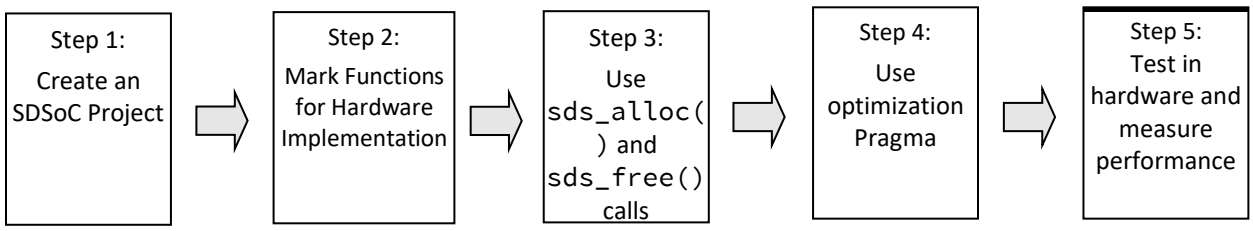
- Use pragmas to select ACP or AFI ports for data transfer.
- Use pragmas to optimize your design.
- Use pragmas to select different data movers for your hardware function arguments.
- Understand the use of `sds_alloc()` and `sds_free()` calls and how it differs from `malloc()` and `free()` calls.
- Analyze built hardware.

Procedure

This lab is separated into steps that consist of general overview statements that provide information on the detailed instructions that follow. Follow these detailed instructions to progress through the lab.

This lab comprises five primary steps: You will create an SDSoC project, mark two functions for hardware implementation, use `sys_port` and `data_mover` pragmas and analyze the built hardware, and, use `malloc()` and `free()` calls and see their impact on the hardware.

General Flow for this Lab



Create an SDSoC Project Step 1

1-1. Launch SDSoC and create a project, called *lab2*, using the empty application template again, targeting the Zed board.

1-1-1. Open SDSoC by selecting **Start > All Programs > Xilinx Design Tools > SDx2018.2**

The Workspace Launcher window will appear. Use the same workspace as for lab1.

1-1-2. Click **OK**.

Click **X** on the *Welcome* tab, if displayed, to close it.

1-1-3. Select **File > New > SDx Project** to open the New Project GUI.

1-1-4. Enter **lab2** as the project name, select *zed* via drop-down button, select *standalone* as the target OS, and click **Next**.

The Templates page appears, containing source code examples for the selected platform.

1-1-5. Select e**mpty application** in case of *zed* as the source.

1-1-6. Click **Finish**.

The *Project Explorer* tab will display the **lab2** project directory. The **lab2** folder also shows the **project.sdsoc** file. Double-clicking on it will display what you see in the right-side pane.

Now copy/paste the sources from lab1 > src directory to lab2 > src directory. Simply select them with CTRL+C and then copy them with CTRL+V inside the SDSoC GUI.

Mark Functions for Hardware Implementation Step 2

1-2. Mark *madd* and *mmult* functions for the hardware accelerations with default clock speed.

1-2-1. Expand **mmult.cpp** and **madd.cpp** under *lab2 > src* in the Project Explorer tab, right click on *mmult* and *madd* functions.

Alternatively, click on the "+" sign in the Hardware Functions area to open the list of functions which are in the source file. Using Ctrl key and mouse clicks, select *mmult* and *madd* entries and click **OK**.

The two function names will be added into the Hardware Functions window. Notice that they will be using the default clocks.

1-2-2. Select **Build Configurations > Set Active > SDRelease**

1-2-3. In the SDSoC Project Overview pane on right, select the Bitstream generation option.

Replace `malloc()` with `sds_alloc()` Step 3

3.1 Observe main.cpp

You will see that the allocation of memory in the heap of the computer for all the matrices `A, B, C` and `D` is done with `malloc()`. `Malloc()` reserves the memory but it does not ensure that it will be physically contiguous. For this reason, when the hardware is created in the FPGA a DMA capable of scatter gather is needed. The scatter gather DMA will translate the virtual memory addresses into physical memory addresses that are in different sections of the memory. This results in a negative performance impact.

The `sds_alloc()` allocates memory that is physically contiguous and improves performance. The problem is that if the required buffer is big maybe there is not enough memory to achieve this.

Replace the malloc calls for `A, B, C` and `D` for `sds_alloc` like:

```
A = (float *)sds_alloc(N * N * sizeof(float));
```

and the free calls for `sds_free` to release the memory at the end. Leave `D_sw` as it is because this matrix is only for the software implementation and not used by the hardware accelerator.

Add pragmas to improve performance Step 4

3.1 Microarchitecture optimizations

1. Pipeline the dot-product loop with II=1 to unroll the inner loop
2. Add pipelined copy loops to local dual-port BRAMs partitioned for parallel access.
3. Partition arrays into different BRAMS (with 2 ports each) so that 32 reads can be performed in parallel.

In `madd.cpp`:

```
for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
        #pragma HLS PIPELINE II=1
        C[i*N+j] = A[i*N+j] + B[i*N+j];
```

With this pragma the tool will try to optimize the hardware so a new j iteration can start in every clock cycle. The adder will be pipelined to improve performance.

In `mmult.cpp`:

```
float Abuf[N][N], Bbuf[N][N];
#pragma HLS array_partition variable=Abuf block factor=<write a number> dim=2
#pragma HLS array_partition variable=Bbuf block factor=<write a number> dim=1

for(int i=0; i<N; i++) {
    for(int j=0; j<N; j++) {
        #pragma HLS PIPELINE
        Abuf[i][j] = A[i * N + j];
        Bbuf[i][j] = B[i * N + j];
    }
}
```

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        #pragma HLS PIPELINE
        float result = 0;
        for (int k = 0; k < N; k++) {//this inner loop gets unrolled
            float term = Abuf[i][k] * Bbuf[k][j];
            result += term;
        }
        C[i * N + j] = result;
    }
}
```

The array partition is very important here because of the way that matrix multiplication works. Matrix multiplication involves multiplying all the elements in the row of matrix A with all the elements of a column of Matrix B and then add all of them to get one element of the result. This is done by the loop with index K.

Then repeat this with all the rows and columns of A and B done by index I and J. The objective is to create hardware that can do all the K multiplications in parallel. We need to access all the elements of row of A in parallel and all the elements of a column of B in parallel.

The first loop copies A and B into `Abuf` and `Bbuf`. Then

```
#pragma HLS array_partition variable=Abuf block factor=<write a number> dim=2
```

This pragma will create several independent memories for `Abuf` along the second dimension which is the column dimension. Hence, each column is stored in a memory and then all those memories can be read in parallel extracting all the row elements in parallel. The second pragma achieves the same thing along the first dimension (e.g. rows). Determine the best partition factor considering the BRAM memory features and the matrix size.

Answer this question:

1. **What is the optimal partition factor? Why?**

Notice that the position of the pipeline pragma means that the K loop which is inside will be automatically unrolled according to the behavior of SDSoC.

A good way to understand the difference between unroll and pipeline is:

The pipeline pragma will try to create hardware that can start a new index I and J in each clock cycle.

On the other hand, unroll tries to create parallel hardware so that the loop interactions are reduced depending on the unroll factor. For example, if the loop has 8 iterations and you set the unroll factor to 8, then the loop will only execute once, and enough hardware will be present to perform the 8 computations in that iteration.

For example:

```
for(int i=0; i<8; i++)
{
    A[i] = B[i]+C[i];
}
```

If we unroll by 8:

```
for(int i=0; i<8; i++)
{
    #pragma HLS unroll factor=8
    A[i] = B[i]+C[i];
}
```

The effect is like if we write in C:

```
for(int i=0; i<1; i++)
{
       A[i] = B[i]+C[i];
       A[i+1] = B[i+1]+C[i+1];
              ….
       A[i+7] = B[i+7]+C[i+7];
}
```

Pipeline will try to build a deep pipeline using registers so that each iteration of the loop can start with a low initiation interval of typically 1 clock cycle. So instead of building hardware horizontally we do it vertically. If 1 cannot be achieved the compiler will try larger numbers until it finds a solution. In general, this vertical parallelism is more hardware efficient than horizontal and results in high performance and lower complexity. It should be tried first but they can be combined as well in the search for the optimal configuration.

3-2 System Optimizations

The sequential access pragma tells the SDSoC compiler that all the accesses will be done to virtual addresses that are sequential in memory. This means that if the algorithm uses element A then the next element will be A+1. For this to work the buffers must be allocated in physically contiguous memory and for this reason the memory allocations done with `malloc` must be replaced with `sds_alloc`. These two modifications allow the usage of an efficient DMA engine called axidma_simple instead of scatter gather or just AXI masters.

In **`mmultadd.h`**

```
#pragma SDS data access_pattern(A:SEQUENTIAL, B:SEQUENTIAL, C:SEQUENTIAL)
void mmult (float A[N*N], float B[N*N], float C[N*N]);

#pragma SDS data access_pattern(A:SEQUENTIAL, B:SEQUENTIAL, C:SEQUENTIAL)

void madd(float A[N*N], float B[N*N], float C[N*N]);
```

Note:

By default SDSoC will try to choose the ports and configuration that will result in best performance with the information that has available. It is possible to use pragmas to change this default behavior. For example

```
#pragma SDS data sys_port(in_A:ACP, in_B:AFI)
```

This will tell SDSoC to use ACP (the default connection type) explicitly for in_A. in_B will have an AFI type which will connect it to one of the PS7 HP ports.

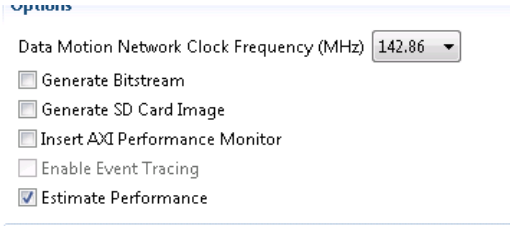
The pragma below will for the port in_A to explicitly use a scatter gather DMA. Notice that the DMA needs sequential accesses to the data and the IP will hang if this is not the case. The scatter gather DMA can work with memory that has been allocated using malloc but the DMA_SIMPLE needs memory allocated with `sds_alloc`.

```
#pragma SDS data data_mover(in_A:AXIDMA_SG)
```

Estimate performance Step 5

Performance optimization is a useful tool that enables the designer to obtain estimations on performance and complexity without having to do a full implementation. It is enabled with the button available in the IDE as shown in this figure and then a performance estimation report can be obtained.

This estimation does not consider possible bottlenecks moving data to the accelerator so sometimes is not very accurate. Experiment with this performance estimation and answer this question:



2. **How many clock cycles the performance estimation reports for the hardware?**

Obtain a new implementation and run it on the board — Step 6

Once you have completed all the modifications build the project. Before that make sure that the option to generate bitstream is enabled and make sure that you are selecting both functions for hardware implementation.

Once the project completes execute it on the board and take note of the observed speed up factor. Also check the logs to find out how many device resources are being used.

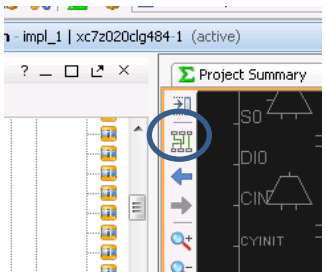
3. **How many clock cycles does the application takes on hardware now?**
4. **What is the acceleration factor after the optimizations? For example, if it is twice as fast then the factor will be 2x.**
5. **What type of DMA IP is being used to move data in and out?**
6. **Check the report sds.rpt to verify how many LUTs and registers resources you are using now.**

Analyzing the implemented design in Vivado — Step 7

Open the Vivado tool and the vivado xpr project that should be located under *\SDRelease_sds\p0\vivado\prj* Once the project opens click on implemented design to view how the logic has been mapped to the device. This will show the FPGA die view under Device. Zoom until you can see one of the slices. The logic used inside the slice will show in a green colour while unused logic remains black. In Zynq two slices form a CLB as in the Virtex-4 but there are differences in the rest of the slice architecture. Exploring the slice architecture answer these questions:

7. **How many flip-flops and how many LUTs are present in a Zynq slice?**
8. **How many inputs the LUTs has and observing the internal architecture what adder size could you implement in a single zynq slice? For example, if you think the answer is a 2-bit adder then write 2.**

A powerful feature in the Zynq slices is that LUTs can be partitioned or fractured so a single LUT can implement a large combinatorial function or two smaller ones and for this reason two outputs are available in each LUT. Click on button shown in the figure below to be able to see the routing wires on the FPGA view.



9. **Observing the slice routing details which output you will use of the LUT if you were trying to implement an adder and in which output of the slice the sum result will be available**?

Conclusion

In this lab, you used various pragmas to control the generated data motion network and number of data movers. You used sys_port and data_mover pragmas and observed the type of ports used. You also used `sds_alloc()` and `sds_free()` calls to handle the contiguous memory usage instead of `malloc()` and `free()`. Then you used optimization pragmas such as pipeline and array_partition to achieve a good acceleration factor of the hardware solution compared with the software solution. The hardware utilization increased as a result of the additional resources that were deployed to achieve hardware acceleration. Finally, you used the Vivado tool to open the implemented design and analyze the physical implementation of the design.

Final exercise

You have now concluded the guided parts of the labs that account for 25% of the final mark. It is now time to perform the quiz on-line in Blackboard. Once you have done the quiz you should move to the open-ended part of the assessment that counts for 25% in which you will aim at creating algorithms for two important problems:

1. Large Matrix multiplication

The first problem is to do matrix multiplication that can handle large matrix sizes. The challenge is that if you simply try to increase the buffer size you will run of internal BRAM memory and the 1024x1024 matrix will not implement. Use the available notes from the final lecture to get ideas on how you can do this efficiently. **You should aim at creating something that works correctly even if it is slower than software**. Once you have something working you can aim at using pragmas and making changes to make it faster.

2. Advanced Encryption Standard (AES)

The second problem is about creating a data flow implementation of the advanced encryption standard algorithm. This is considered an optional exercise to improve your marks and should only be attempted after having done a successful implementation of Matrix Multiplication that works correctly. A successful implementation is one that works correctly but also that achieves acceleration compared with the software version (even if this acceleration is small). It is not possible to get marks for AES if the matrix multiplication system is not working.

***AES tips**: a good idea to get a high performance system is to create multiple instances of the processing functions and make sure the output of a function is the input to the next then use the dataflow directive to make use of function pipelining, use buffers between functions to help the compiler to implement the dataflow correctly. For example in this code we are creating a buffer called inter0 of data stream_t type. data_stream_t is defined in aes_enc.h in your sample code in Blackboard.*

```
data_stream_t inter0;

      #pragma HLS STREAM variable=inter0 depth=1
```

Then you can have a function that writes to inter0 and another that reads from inter0 pipelined with a dataflow directive as follows:

```
#pragma HLS DATAFLOW
    array2stream(state,state_stream,byte_count);
    addroundkey(state_stream,0,inter0,ekey,byte_count);
    subbytes(inter0,inter1,byte_count);
//… Other functions needed to complete AES
```

Notice that the input array state is transformed into state_stream with a function called `array2stream()` that is made available in the the sample code for AES in blackboard. So, a possible approach is to create a dataflow architecture with all the functions involved in AES and manually unroll the for loops.

In this final lab you will work with your team partner, but you will not be getting any lab books etc. to properly reflect the way you will tackle this problem as part of work done in a real company. Marking will be based on the

correctness and performance your solution. When you get your system to work you should get it verified by a demonstrator or academic to check functionality, performance and complexity. Marking is based on 3 parts:

PLAGIARISM WILL RESULT IN ZERO MARKS FOR THE EXERCISE!

1: **Implementation:** which has two components: **performance** (how much faster is your approach compared with the software) and **complexity** (the more resources you use in the FPGA the better, try to max it out so there is nothing left to use). For example, if you only use 10% of the available resources then your solution is not really exploiting what the hardware can do. Please, note that the mark for the implementation can only be obtained if your solution is functionally correct. There is no point in making a very fast matrix multiplier if the result of the multiplication is incorrect.

Mark =((what is the percentage of the FPGA used) * 0.2 + (what is the acceleration factor of your solution compared with the ARM) * 10) * (does it work correctly ? : 0 if no, 1 if yes)

Here is the ARM performance using the NEON unit for different matrix sizes as reference:

256x256	80.64 ms
512x512	787.36 ms
1024x1024	**6603.72 ms**
4096x4096	437192.35 ms

For example, a design that uses 80% of the FPGA and improves the performance compared to the optimized ARM by a factor of 8 and works correctly will get:

Mark1 = (80 * 0.2 + 8 * 10) * 1 = 16 + 80 = 96 %

When you are happy with your solution you should approach a demonstrator or the lecturer to verify your work and write down your values for these questions so that your mark can be calculated. This should be done at some point during the lab sessions.

1. What is the percentage of the FPGA used?
2. What is the acceleration factor of your solution compared with the ARM?
3. Does it work? (yes = 1, no = 0)

If your mark is higher than 100% then it will be capped at 100%.

If you get MM to work correctly and attempt AES, then there will be a mark2 calculated using the same formula. If you do not attempt AES or if you attempt AES but MM does not accelerate then **mark2** will be capped to 0. Remember that AES is optional and only required for bonus marks.

2: **report and code quality** (blackboard submission including quality of description of solution and quality of figures without copy/pasting, code extracts used to explain key concepts of the work done, code properly formatted and with comments): Mark3

The final overall mark is calculated as follows:

Final mark = Mark1 * 0.4 + Mark2 * 0.2 + Mark3 * 0.4

As a guide here are possible mark ranges:

1. A solution that does not performance matrix multiplication correctly (<50% mark)
2. A correct solution that is not very fast or implements parts of the computation in the ARM processor (~50-60%)
3. A correct solution that is significantly faster than the ARM processor code and uses most of the FPGA hardware with minimal intervention of the ARM processor in the computation (~70%). Note that there are results in Blackboard with an optimized matrix multiplication algorithm using only the ARM processor suitable for comparison purposes with your solution.

4. Point 4 extended with a good report, good code and good figures (~80%)
5. Point 5 extended with a successful AES implementation (>80%)

As indicated in the lecture notes you need to submit a single zip file in Blackboard containing:

Short report explaining what have you done and how for the MM and optionally also for the AES problem. Include the results in terms of performance and complexity. You are welcome to add figures to explain concepts but do not copy/paste from notes or Xilinx materials. A good report will start with an introduction to the problem and the SDSoC flow followed with a description of the proposed solution and any problems encountered that could include bug in the tools. Then figures reporting the performance and complexity of the solution. A good report should include a conclusion section indicating the learning outcomes that have been achieved.

All the source code you have used to obtain these results so that it can be implemented and run on the ZED board.

Learning outcomes

1. Learn how to obtain optimal hardware performance with coding changes in high-level algorithm descriptions to make them more suitable for hardware.
2. Learn how to analyze the performance results to understand the effects of optimization pragmas in the final solution.
3. Learn how to verify your results using real FPGA boards.

Marking scheme

Marking for this lab is based on the Quiz that will ask questions as shown in the lab and assign marks depending on the level of difficulty. Additional quiz questions will assess that you have understood the theory regarding FPGA architecture. You can use your notes when doing the quiz, but it needs to be done individually. The final exercise is marked according the obtained results as indicated in section above **Final exercise.**