High-Level Synthesis: Accelerating Alignment Algorithm using SDSoC

Steven Derrien & Simon Rokicki

The objective of this lab is to present how High-Level Synthesis (HLS) can be used to accelerate a given application using FPGAs. We will study the Smith-Waterman algorithm, which is often used in bioinformatics. The acceleration will be made using Xilinx SDSoC software and running on a Zybo board.

This document will often point toward additional information located on 2016.2 version of Xilinx documents “Vivado Design Suite User Guide – High Level Synthesis” and “SDSoC Environment – User Guide”.

Part 1: Presentation of Smith-Waterman algorithm (from Wikipedia)
Smith-Waterman algorithm performs local sequence alignment. For two sequences of size n and m, it will fill a matrix of scores as follows:

\[
M(i, j) = \max \left\{ \begin{array}{ll}
0 & \\
M(i-1, j-1) + s(a_i, b_j) & \text{Match/Mismatch} \\
M(i-1, j) + W & \text{Deletion} \\
M(i, j-1) + W & \text{Insertion}
\end{array} \right.
\]

Where \( a, b \) are the input strings, \( s \) a similarity function, and \( W \) the penalty due to a deletion or an insertion. Once the matrix is completely filled, the alignment score is the maximal score in the matrix and the alignment can be found by reversing the building process (e.g. If the score comes from a match/mismatch, we go to cell \((i-1, j-1)\); if it comes from a deletion, we go to \((i-1, j)\) etc.).

For example, let’s applying the algorithm on sequences ACACACTA and AGCACACA with a score of 2 for a match and -1 for a mismatch or an insertion/deletion.

The score matrix built represented on the figure beside and the final alignment is highlighted.

We can see that the maximal score of alignment is 12 and the corresponding alignment is:

\[
\begin{array}{cccccccccccc}
\end{array}
\]

Implement a first version of the algorithm where we only compute the maximal score (no need to build the alignment). Execute it on the host computer and note the execution time.

Now that a first version of the code is implemented, we will see how to quickly develop a hardware accelerator for it using SDSoC.
Part 2: A first look at Xilinx SDSoC

SDSoC is a tool from Xilinx which allows software developers to easily exploit FPGAs to accelerate their applications. It is based on Xilinx Vivado and Xilinx Vivado HLS tools and is made to make these technologies more accessible.

First we will try to understand the different software being used:

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
<th>Avg. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vivado HLS</td>
<td>From a c/c++ procedure and user directives, it generates a RTL representation of a hardware accelerator.</td>
<td>10/20 sec</td>
</tr>
<tr>
<td>Vivado</td>
<td>From a RTL representation, it generates the bit-stream used to program the FPGA.</td>
<td>10/20min</td>
</tr>
<tr>
<td>SDSoC</td>
<td>From a complete c/c++ project, it allows to develop a hardware/software system where most critical procedures are mapped to hardware.</td>
<td>-</td>
</tr>
</tbody>
</table>

As we can see, SDSoC is simply an overlay to use Vivado and Vivado HLS transparently. During this lab, we will first use SDSoC to generate a naïve version of our hardware/software system. Then we will use tools from Vivado HLS to further optimize the accelerator we developed. We will try to use performance estimation whenever it is possible to avoid the time-consuming part of the process (e.g. bit-stream generation).

➔ Start SDSoC (see cheat sheet) and create a new SDSoC project. Choose the platform zybo and a Linux project. On the next tab, choose an empty application. Once the project is created, add the sources of the project (copy and paste from a file explorer should be working).

➔ Build the project for the Zybo. As no procedure will be mapped into hardware, the compilation will be fast. Copy the generated elf file (from folder SDDebug/sd_card) on the SD card, insert it on the board and switch it on. Setup a serial connection to the board (see cheat sheet) and launch the application (/mnt/SmithWaterman.elf).

➔ The design will now be mapped into hardware: modify the input size to 128 to fit the hardware limitations.

➔ On the Project Explorer, expand the c file you added and right-click on the smithWaterman procedure. Choose Toogle HW/SW to indicate that you want this procedure to be mapped in hardware.

➔ Go in the file project.sdsoc and select the option “Performance estimation”. Start the compilation once more.

The performance estimation mode will call Vivado HLS but not Vivado: the hardware accelerator will be designed but not actually compiled for the FPGA. When you’ll run the application on the board, the software part is executed normally, memory transfers are done but when the execution will reach the function mapped into hardware, it will execute it in software and also use a model from Vivado HLS to estimate the time needed by the accelerator. Using this method, the tool will provide a precise estimation of the speed-up from using the accelerator.

More details on performance estimation can be found on page 15 of the SDSoC documentation.
Once the compilation is done, a new window called Estimate Performance will open. In here you can click to run the application and collect performance estimation.

At this step, you developed the software part of the system and asked SDSoC to accelerate a given procedure. However, it is difficult to understand exactly how the accelerator has been designed using your c code. In order to understand it more precisely and to further optimize it, we will now use Vivado HLS and its dedicated tools.

Open the file project.sdsoc and select the smithWaterman procedure in the Hardware Function tab. Launch Vivado HLS by using the corresponding icon (“Launch Vivado HLS” above the clock frequency choice). This will open Vivado HLS with a project containing your procedure.

In this new window, you can open the C file and choose to see the Directive panel on the right part of the window. You will now have a synthetic view of you code with all directive used by Vivado HLS to generate the accelerator. Those directives can concern inputs/outputs, memories and loops. Here is a quick description of most interesting directives:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Target</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>Input/output</td>
<td>Used to define which kind of input/output to use.</td>
</tr>
<tr>
<td>Array Partition</td>
<td>Memory</td>
<td>Used to partition a memory into several smaller banks. This will increase the number of read/write ports but Vivado needs to be able to prove that there will be no conflict.</td>
</tr>
<tr>
<td>Unroll</td>
<td>Loop</td>
<td>Unroll a loop.</td>
</tr>
<tr>
<td>Pipeline</td>
<td>Loop</td>
<td>Pipeline a loop.</td>
</tr>
</tbody>
</table>

You can find the complete list of directive and their precise description on the Vivado HLS User Guide, page 131.

Add a dependency on the inner-most loop to unroll it by a factor of 2 and perform the synthesis (play icon on the upper part of the window, called “Run C Synthesis”).

Once the synthesis is performed, a summary is opened which give information about the timing, the performance estimation for each loop and the resource utilization. This summary is the tool you will use to efficiently explore different solutions while developing your hardware accelerator. The report is organized as follows:

- Timing information give you the critical path of the design, which constraints the maximal frequency available.
- Latency information corresponds to the number of clock cycle needed to execute the procedure in hardware. It is derived by analyzing loop bounds and latency of a single iteration. You can have more details by expanding the “Loop” part.
- Hardware utilization provides you an estimation of how much you used the hardware available on the board.

You can also have access to a more detailed summary by choosing the option “Open analysis perspective”. This perspective offers you to analyze the schedule done and to find hints to understand why some transformation are not done. Log messages from the console are also of interest.
Try to unroll the loop with different factors and measure the impact on performance, timing and area utilization.

Another important consideration when doing HLS is the size of variables. Vivado HLS offers the possibility to create values with customized bit size. You can use them as described below:

```c
#include <ap_int.h>
ap_uint<14> a = 0;  //14 bit unsigned variable
ap_int<14> a = 0;  //14 bit signed variable
```

- Change variables type to adapt their size and observe the impact on performance.
- Remove the directive for loop unrolling.
Part 3: Optimizing the algorithm – Skewing and Tiling

As we seen in previous section, a naïve porting of the algorithm on FPGA is not efficient at all. If we really want to bring performance, we have to modify the code in order to exploit more efficiently the FPGA capabilities.

As we can see from the algorithm description, computing the score for iteration \((i,j)\) requires scores from \((i-1, j)\), \((i-1,j-1)\) and \((i,j-1)\). This results in the dependency profile described on the figure beside. We have no parallelism on i-loop nor on j-loop.

The hardware accelerator generated by SDSoC with this implementation will compute one score at a time and will have a very poor utilization of FPGA’s capabilities.

In order to remove dependencies on j-loop, we propose to perform loop-skewing on j. We apply the following transformation:

\[
(m, n) \text{ with } m = i + j \quad \text{and} \quad n = j
\]

With this transformation, loop on n can now be parallelized. If we want efficient hardware acceleration, we still have to define how this parallelism is exploited.

We will now perform loop tiling on n to expose small parallel tiles that can be used as kernels in the hardware accelerator. For this we split the loop on n into a nested loop \((nn, n)\): \(nn\) will go from 0 to \(SIZE/TILE\_SIZE\) and \(n\) from 0 to \(TILE\_SIZE\).

Implement these loop transformations on the code and use Vivado HLS to measure the impact on the efficiency of the accelerator and on the utilization of the FPGA.
We will now modify the code to prevent storing all intermediate results in the matrix. Observe that to perform a given iteration, you only need results from the two previous iterations.

Modify the application to reduce the memory usage of the application. Once it is done, you can go back to high size inputs.

Part 4: Optimizing the algorithm – Loop Pipelining

Once the kernel handling a tile of iteration is correctly defined, loop pipelining is a natural way to increase the efficiency of the hardware accelerator without increasing its size.

The idea of loop pipelining is to start a new iteration of the loop before the end of the previous one. In our case, let say that handling a single iteration needs three cycles: loading previous scores, computing the new one and storing it in memory. A non-pipelined version of the loop will work like this:

```
Load prev. score → Compute score → Store result
```

In this case, the total execution time would be LATENCY*NB_ITERATION. If we pipeline the loop with an Initiation Interval (II) of 1 (e.g. if we start a new iteration after 1 cycle), the loop looks like this:

```
Load prev. score → Compute score → Store result
```

```
Load prev. score
```

```
Load prev. score
```

```
Load prev. score
```
In this case, the total execution time would be \( \text{LATENCY} + \text{NB\_ITERATION} \times \text{II} \). On average, we can say that the accelerator needs \( \text{II} \) cycles to handle one iteration. Finding the \( \text{II} \) depends on resource constraints (is there enough memory ports to pipeline the loop?) or inter-iteration dependencies (does iteration \( x \) need result from iteration \( x-1 \)).

Pages 138 to 151 of the Vivado HLS Documentation provide an in depth study of loop pipelining and of transformations needed to make it applicable.

In our application, we are concerned by these two problems: an iteration may need the result of a previous one and we may have conflict when we access the memory where these score are stored or when we read the inputs.

- To solve the problem of memory ports, we can exploit the vectorized aspect of memory accesses: inside the kernel, we read 4 inputs from A which are next to each other. There are two ways to exploit this property to reduce pressure on memory ports: i) define the memory as an array of bigger words, and at the start/end of each kernel execution, use a loop to split them into smaller words; ii) Use the ARRAY_PARTITION directive in the concerned memory. This last option may not be as efficient as it seems.

- To solve the problem of inter-iteration dependencies, we can add a new dimension on \( n \) in order to ensure that the distance between two dependent iterations is greater than the latency of an iteration. This idea is summarized on following Figure. If the tool is not able to prove that there is no more inter-iteration dependencies, the use of the directive DEPENDENCE can force the decision.

Part 5: Measuring AXI performance

Finally, one of the last bottleneck of your application will be the way you interface it with the Linux running on the Arm core. Even if your hardware accelerator is efficient, if the overhead required to move
data from main memory to the on-chip memory and start the accelerator is too important, the overall execution time will be impacted.

In this part, we will use SDSoC AXI performance monitor to examine the way the accelerator is used by the system.

- In the project.sdsoc file, enable the event tracing and build the project. This step may last for 30 minutes.
- Once the project is build, copy the content on the SD card and boot the device.
- Setup a network connection between the board and the computer (see cheat sheet). Right click on the elf file in the SDDebug folder and choose Run as... Trace Application (SDSoC Debugger).

SDSoC will now start the application on the device and collect execution trace. After the collecting is done, you’ll see the result in the window called AXI State View (if not opened, you’ll find it in Window, Show View, Other, Xilinx).

This window retrace how the accelerator is used: how data are sent and treated by the hardware. The ideal scenario is when date transfer is done during the hardware treatment.

Pages 23 to 31 of the SDSoC documentations provide more details on how to guide the accelerator use.
1. Starting SDSoC

Starting SDSoC needs to configure correctly the environment and the license file:

```
export SWT_GTK3=0
export XILINX_LICENSE_FILE=/lereste/sdsoc/SDSoC/Xilinx.lic
source /lereste/sdsoc/SDSoC/2016.2/settings64.sh
sdsoc &
```

2. Setting up a serial connection to the board

To establish a serial connection with the board, we will use SDSoC SDK Terminal. Open the tab called SDSoC terminal on the lower part of SDSoC window and click on the + to configure a new connection.

Set the baud rate at 115200 and use the port /dev/ttyUSB1.

You’ll be connected to the board as root.

**Note**: Sometimes when you power-off the device without closing the terminal, the file /dev/ttyUSB1 is still locked and the next time you plug the device, system will create a /dev/ttyUSB2 instead. You can check which one to use by listing files on /dev and picking the highest ttyUSBn.

3. Setting up the network connection to the board

In order to debug your program, SDSoC will need the IP address of the board. In order to establish the connection, you’ll need to setup a DHCP server on the board, which will attribute an address to the host computer.

- On the SD card, add the file dnsmasq_base provided with the project files.
- Start the board and establish a serial connection to the board.
- On the board, run the following:

```
/usr/sbin/udhcpd -I 192.168.0.1
ifconfig eth0 192.168.0.1
```

- On the host machine, check if the connection is established correctly:

```
ifconfig
```

The result of the command should display an IP for eth1 interface.

The board is connected to the host computer and its IP is 192.168.0.1. You can connect using SSH: login/password is root/root. You can also configure SDSoC TCF agent to use this IP address on the lower left part of the main windows.
Cheat sheet: How to use the environment

Information for cairn-cao1

1. Starting SDSoCa.
Starting SDSoC needs to connect on cairn-cao1, configure correctly the environment and the license file:

```bash
ssh -Y cairn-cao1.irisa.fr
export SWT_GTK3=0
export XILINXD_LICENSE_FILE=1717@loth3.enssat.fr
source /opt/Xilinx/SDSoC/2016.1/settings64.sh
sdsoc &
```

2. Setting up a serial connection to the board
To establish a serial connection with the board, we will use SDSoC SDK Terminal. Open the tab called SDSoC terminal on the lower part of SDSoC window and click on the + to configure a new connection.

Set the baud rate at 115200 and use the port /dev/ttyUSB1.

You’ll be connected to the board as root.

**Note:** Sometimes when you power-off the device without closing the terminal, the file /dev/ttyUSB1 is still locked and the next time you plug the device, system will create a /dev/ttyUSB2 instead. You can check which one to use by listing files on /dev and picking the highest ttyUSBn.

3. Setting up the network connection to the board

Not possible currently....