

# Lab 3

## FPGAs as Accelerators

### Introduction

The goal of this Lab is to implement the **LSAL algorithm** on the x86, Arm and FPGA of the zedboard and to optimize it in every case using **Valgrind**, **Intel advisor** and **Vitis HLS**.

LSAL stands for *local sequence alignment algorithm* and is used in *bioinformatics* and works by giving an input of two strings, a database  $D$  of length  $M$ , and a query  $Q$  of length  $N$ .

The objective is to find regions of similarity between subsequences of all possible lengths. An  $M \times N$  similarity matrix  $S$  is constructed to hold similarity scores for each subsequence comparison.

A second matrix (called direction matrix) is formed to hold the direction path through matrix  $S$  to produce an *alignment*. The cell value of the direction matrix indicates the direction followed to reach that cell. The optimal alignment is produced by starting from the position of the maximum score in matrix  $S$  and following the directions found at the same index in the direction matrix until a zero in  $S$  is found.

In typical cases, the size  $M$  of the database string can be in the order of millions of symbols.

### File Structure

The submitted files follow the following format.

```
├── FPGA
│   ├── lsal.cpp
│   └── lsal.host
├── Software
│   ├── lsal_optimized.cpp
│   └── Makefile
└── Report.pdf
```

The *FPGA* folder contains the *lsal\_host.cpp* code that runs on the ARM of the zedboard and the *lsal.cpp* code that runs on the FPGA.

The *Software* folder contains the *lsal\_optimized.cpp* code for the software and the *MAKEFILE* that allow it to compile for the desired purpose.

### Software (x86 & Arm)

#### Workflow

The general workflow for the development of the code which is meant to run on x86 and ARM environments is as follows:

1. First we **implement** the LSAL algorithm without considering the CPU running time and only taking into account the *functionality* of the code.
2. Then we **profile** the code using **Valgrind** and **Intel Advisor** while identifying the parts of the code that take the most amount of time to run and which of them can change.
3. Depending on the results of the previous step we change the necessary parts of the code but now aiming for **performance**.
4. Lastly, we **compare** the running times of the optimized and unoptimized version both by CPU running times and **Roofline analysis** results.

In both the unoptimized and optimized code we have added a **debug** DEFINE that allows us to debug any possible problem by just using the command *make debug* of our *MAKEFILE*.

#### Unoptimized

The **unoptimized code** represents an *implementation* of the LSAL algorithm where the output matrix is being calculated row-wise.

In the code we have added a **second for loop** (compared to the lab input) in order for the algorithm to calculate the values of the **first row** because it meets a lot of edge cases which would be not efficient to place it in the *main loop* as all those edge cases would

need to be checked for every value of the output arrays while this only needs to happen while the first row is being filled out.

In the main loop (that was already there from the lab input) we have added all the cases and the necessary calculations. At the end of the for loop we save the results both in the similarity matrix and the direction matrix while checking if the max index needs to be "dethroned".

The results for a varying number of array sizes (Query and Database) are shown in the table below.

x86 vs ARM Unoptimized (sec)				
N	M	ARM	x86	Difference
32	32	0.000098	0.00002	79.5918%
32	65536	0.175177	0.037108	78.8169%
256	65536	1.402216	0.303335	78.3675%
256	300000	6.417753	1.388256	78.3685%

As we can see the x86 is **78.82% faster** on average than the ARM processor of the zedboard.

## Optimized

In order to profile the unoptimized code, we used **Valgrind** a tool that helps to identify parts of the code that take up most of the execution time.

## Branching

The **branches** are some of the instances where the code was **slow** and our solution to the problem was to reduce the if commands by **unrolling** the loops.

For example, the **northwest** and the **west** during the calculation of cells of the first column **cannot be read** because they are out of the bounds of the similarity array. Thus for the first cell of the each row the calculation of northwest and west **is skipped** by unrolling the inner loop.

## Division & Modulo

Also calculations needed to find the index of the similarity and direction array are hard to calculate thus take **more time** to be executed (Mostly **modulo** % and **division** /). So to eliminate the need to calculate the x

and y indexes of the matrices we **divide** the main loop to **two smaller** ones with one of them representing the rows and the other the columns.

## Defines

It was noticed that **constant variables** were used to define penalty, match, mismatch scores and enumerate the directions of the direction matrix (N, W, NW, C). That may have a **bad effect** on performance because to access those values it is needed to access memory. So even though the boost performance was minor we transformed every constant variable to defined values that became static during compilation (**#define**).

x86 vs ARM Optimized (sec)				
N	M	ARM	X86	Optimization
32	32	0.000045	0.000015	66.6667%
32	65536	0.076079	0.024057	68.3789%
256	65536	0.586488	0.196926	66.4228%
256	300000	2.685104	0.897239	66.5846%

## Comparison

Lastly we have run the **Roofline Analysis** that compares the optimized and unoptimized code (when run on x86) using the **Intel Advisor** as shown below:

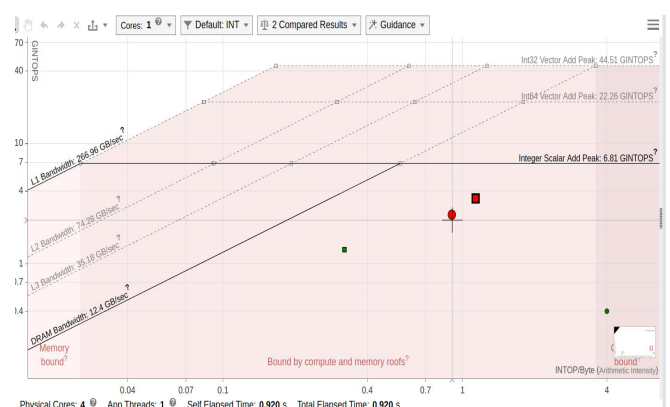


Image: Diagram of Roofline Model Analysis using Intel Advisor

The **red circle** represents the unoptimized version whereas the **red square** represents the optimized version of the code.

As we can see the optimized version has achieved **better parallelism** (as it is placed higher) and is even less dependent on the memory bandwidth (as it is placed further to the right).

With these changes the running times have dropped significantly with an average decrease of **32.65%** on the x86 and **56%** in the ARM processor of the zedboard.

As a conclusion, the optimized performs better on all aspects and has reduced running time both on x86 and on Arm by **44.32%** on average.

x86 (sec)				
N	M	Unoptimized	Optimised	Optimization
32	32	32	0.00002	25.0000%
32	65536	65536	0.037108	35.1703%
256	65536	65536	0.303335	35.0797%
256	300000	300000	1.388256	35.3693%

ARM (sec)				
N	M	Unoptimized	Optimised	Optimization
32	32	0.000098	0.000045	54.0816%
32	65536	0.175177	0.076079	56.5702%
256	65536	1.402216	0.586488	58.1742%
256	300000	6.417753	2.685104	58.1613%

## Hardware (FPGA)

During the last part of the lab we use **Vitis HLS** to implement the **LSAL algorithm** on **FPGA** hardware.

In order to have the optimum outcome we defined a **streamlined workflow**:

1. First our team tries different **optimization methods** on **Visual Studio Code IDE**.
2. Then functionality of the C code is tested using **Vitis HLS's C Simulation**.
3. At this point synthesis and **Synthesis Design Report** of **Vitis HLS** is used to analyze the resources in-use and the time estimations.
4. Next we use the provided **makefile** to **emulate** the kernel design using **QEMU**
5. Lastly we test the design **natively** on the **FPGA** and extract the **timing** of our implementation

It is important to note that both **C Simulation**, which tests functionality of the code before it gets synthesised, and software emulation using **QEMU** are

really important as a simple mistake can be detected and fixed long before the time it takes to have the kernel implemented into a bitstream.

Also **Synthesis Design Report** is a great tool to have in order to make sure the **HLS optimizations** you use have been implemented in the correct way. For example an easy mistake to make is placing a **#pragma** above a loop instead of inside it, which could result in vastly different implementations.

## Algorithmic Optimizations

The main optimization introduced in the category of algorithmic optimizations is using **diagonals** to calculate the **similarity matrix**. As mentioned during the lecture in this way we **limit data dependencies** and "unlock" **parallelism**.

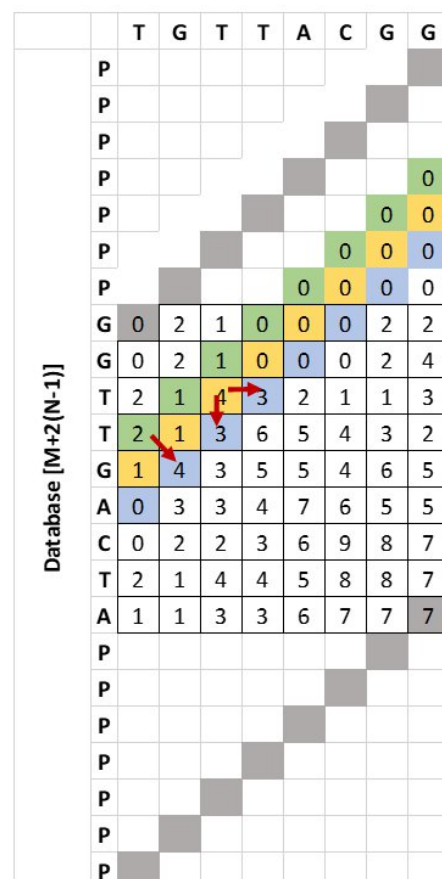


Figure 1: Showing diagonal parallelism, image provided during lecture.

This method requires that every diagonal is of the same size. Thus we need to enlarge the database by  $2(N - 1)$ , where  $N$  is the size of query and  $M$  the size of

database.

In order to traverse the direction matrix **two nested loops**. The outer one counts the diagonals, which are  $M + (N - 1)$  in crowd. The first diagonal to be calculated is the one that starts at the top right element of the extended direction matrix.

The other one indicates which element of the diagonal is calculated, from top right to bottom left. We decided to traverse the diagonal in such a way because it much easier to calculate the index variable that is needed to store the location of the maximum value.

We realized that there are **data dependencies** on the 2 diagonals above the one in calculation during the calculation of a diagonal. That means once the diagonal is calculated it can be moved to the buffer of the above diagonal.

Also one thing noted to us is that at the beginning of each iteration of the loop we have a **write-after-read** on the current diagonal, because at the end of a iteration we write the diagonal we just calculated to main memory while at the start of the next we try to **write** to it. That may be an **obstacle** to having a **pipelined** design with **low LL**.

To limit the **data dependencies** we **double** the size of the diagonal direction and the current diagonal (stores temporary similarity matrix values of the current diagonal) matrices.

## HLS Optimizations

This section includes every *optimization* that is **RTL specific** using **HLS**.

### Array buffers

A method of making memory access faster is by using memory blocks that are near the processing unit of the acceleration. The designer in this case has 2 main options:

- by using **Block RAM** (BRAM), that is a distinct memory block surrounded by Look-up Tables, and
- by using the **Flip-Flops** inside the Look-up Tables of the FGPA, which is faster but at the

expense of making the rest of the LuT hard to utilize.

Thus for the best results a combination of the two is needed.

For our implementation the arrays has the following characteristics:

Array	Description	#pragma	
string1	Buffer of query	PARTITION	factor=2 cyclic
string2	Buffer of database	PARTITION	factor=2 cyclic
current_diag	Buffer of current diagonal	PARTITION	factor=2 block
up_diag	Buffer of the above diagonal	PARTITION	factor=2 cyclic
upper_diag	Buffer of 2 diagonals above the current one	PARTITION	factor=2 cyclic
direction_diag	Buffer of current diagonal	PARTITION	factor=2 block

\*\*\*Γιατί cyclic ή block\*\*\*

### Arbitrary Elements

Another easy **optimization** to make is to *reduce the size of the bus* used on the design.

For example in the case of the **query** and **database** we know as a fact that they need to *enumerate* just the 4 nucleotides in existence plus one more undefined state we use in combination with the diagonals optimization. Thus we only need to use **3-bits** to those **enumerate 5 states**,  
( $2^3 = 8 > 5$  states).

In the exact same fashion the bits used to represent the **direction matrix** are reduced to **3-bits**.

To be noted other data structures such as the similarity matrix, it's diagonal buffers and max\_index **must not** be represented by a **smaller number of bits** because the RTL designers cannot easily calculate the maximum value each structure is going to store.

## Comparison

## Results

FPGA vs CPU

χαχα