Lab 3

FPGAs as Accelerators

# Introduction

\*\*LSAL Explanation

\*\*File structure

# Software (x86 & Arm)

\*\*workflow

## Unoptimized

## Code Profiling

## Optimized

\*\*valgrind

## Comparison

# 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***:

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

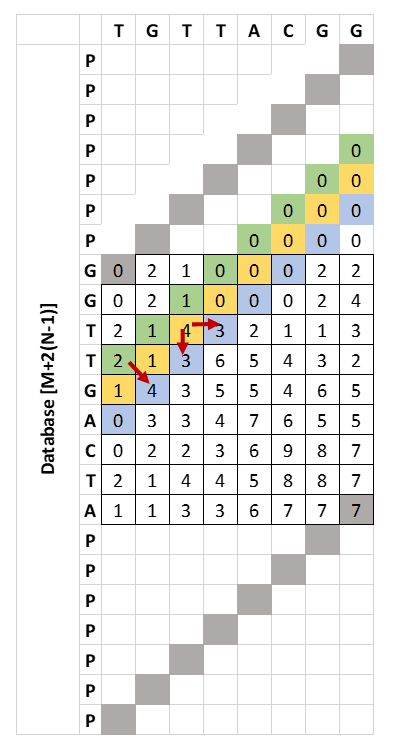
It is important to note that both *C Simulation*, *which tests functionality of the code before it gets synthesised*, and *software emulation using QEMU* are really important as a simple mistake can be detected and fixed long before the time it takes to have the kernel implemented into a bitstream.

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***.

*Image: 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 , *where N is the size of query and M the size of database.*

We realized during the calculation of a diagonal that there are data dependencies on the 2 diagonals above the one in calculation. 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***.

*\*\*\*ADD MORE ABOUT THE LOOPS AND IFs\*\*\**

## 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 |  |  |
| up\_diag | Buffer of the above diagonal |  |  |
| upper\_diag | Buffer of 2 diagonals above the current one |  |  |
| direction\_diag | Buffer of current diagonal |  |  |

### 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 ***3-bits*** to ***enumerate 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