Lab 3

FPGAs as Accelerators

# Introduction

\*\*LSAL Explanation

\*\*File structure

# 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 identify 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 and 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 removing the define from the comments.

## Unoptimized

The [unoptimized code](#Unoptimized_code) represents an implementation of the LSAL algorithm where the output matrix is being completed one by one and its values are calculated in series with an integral of one.

In the code we have added a second for loop 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 be checked for every value while this only needs to happen while the first row is being completed. 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.

Τhe results for a varying number of array sizes (Query and Database) being as shown in the picture below.

As we can see the x86 is 82% faster on average than the ARM processor of the zedboard.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x86 vs ARM Unoptimized** | | | | |
| Ν | Μ | ARM | x86 | Difference |
| 32 | 32 | 0.000098 | 0.000013 | 86.7347% |
| 32 | 65536 | 0.175177 | 0.034103 | 80.5323% |
| 256 | 65536 | 1.402216 | 0.263528 | 81.2063% |
| 256 | 300000 | 6.417753 | 1.196854 | 81.3509% |

## Code Profiling

For the code profiling we used Valgrind where we identified wich parts of the code took up the most amount of time and which of them could change.

The below lines took a staggering percentage of the total running time:

      i = index % n; // column index

      j = index / n; // row index

## Optimized

The two mentioned lines of code could change by completely removing them and replacing them with a second for loop that was responsible for the columns of the data array. With the addition of this second inner loop we were also able to remove the checks for the implementation of the first column which further reduced the running time.

Another change that we made was to change the const short variables into defines. We did this because DEFINE statements are handled by the pre-processor which allows for faster running times. We also DEFINED the index value for the same purpose as seen below:

#define GAP\_i -1

#define GAP\_d -1

#define MATCH 2

#define MISS\_MATCH -1

#define CENTER 0

#define NORTH 1

#define NORTH\_WEST 2

#define WEST 3

#define index i+j\*N

With these changes the running times have dropped significantly with an average decrease of 37% on the x86 and 56% in the ARM processor of the zedboard.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x86** | | | | |
| Ν | Μ | unoptimized (s) | optimised (s) | Optimisation |
| 32 | 32 | 0.000013 | 0.000012 | 7.6923% |
| 32 | 65536 | 0.034103 | 0.018605 | 45.4447% |
| 256 | 65536 | 0.263528 | 0.137813 | 47.7046% |
| 256 | 300000 | 1.196854 | 0.628575 | 47.4811% |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ARM** | | | | |
| Ν | Μ | unoptimised (s) | optimised (s) | Optimisation |
| 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% |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x86 vs ARM Optimized** | | | | |
| Ν | Μ | ARM | x86 | Difference |
| 32 | 32 | 0.000045 | 0.000012 | 73.3333% |
| 32 | 65536 | 0.076079 | 0.018605 | 75.5452% |
| 256 | 65536 | 0.586488 | 0.137813 | 76.5020% |
| 256 | 300000 | 2.685104 | 0.628575 | 76.5903% |

## 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.A screenshot of a graph

Description automatically generated

The circle represents the unoptimized version whereas the 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 more to the right).

As a conclusion, the optimized performs better on all aspects and has reduced running time both on x86 and on Arm by 46.5% on average.

# Hardware (FPGA)

\*\*workflow

## Algorithmic Optimizations

## HLS Optimizations

## Comparison

# Results

## FPGA vs CPU

[](#Unoptimized_code)

Unoptimized Code 1