Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологии
Высшая школа интеллектуальных систем и суперкомпьютерных технологий

ОТЧЕТ ПО КУРСОВОЙ РАБОТЕ

Дисциплина: Проектирование реконфигурируемых гибридных вычислительных систем

Тема: Фильтрация изображения

Выполнил студент гр. 01502 С.С. Гаспарян
Руководитель, доцент А.П. Антонов

«30» ноября 2021

Санкт-Петербург
2021

Содержание

Задание

Реализовать алгоритм фильтрации изображения на языке программирования C. Провести синтезирование функции фильтрации с разными временными параметрами – *clock period*. Выбрать наилучший вариант. Провести ряд оптимизации над полученным лучшим решением. Сравнить полученное аппаратное решение с программным, выполнив синтезируемую функцию на ПК для разных разрешений изображений. Провести анализ получившихся результатов и сделать заключение.

Введение

Изображение можно определить как двумерную функцию $f(x, y)$, где x и y – координаты в пространстве (конкретно на плоскости) и значение f которой в любой точке, задаваемой парой координат (x, y), называется интенсивностью или уровнем серого изображения в этой точке [1]. В компьютере изображение представляется, как правило, в виде двумерного массива, где значение любого элемента этого массива и есть интенсивность изображения.

Фильтрация изображения представляет из себя яркостное преобразование, т. е. применение определенного оператора на изображение. В качестве преобразования, в данной работе используется операция математической свертки:

$$f(x,y) \star \mathrm{w}(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x-s, y-t) \qquad (1)$$

где f – исходное изображение и w – ядро свертки. На рисунке 1 показано визуальное представление фильтраций изображения посредством операции свертки [2]. В данной работе изображение будет рассматриваться только в пространственной области. Отдельное внимание стоит уделить обработки границ при фильтрации изображения. Также, в данной работе элементы матрицы изображения, находящиеся на границе изображения, будут приравниваться 0.

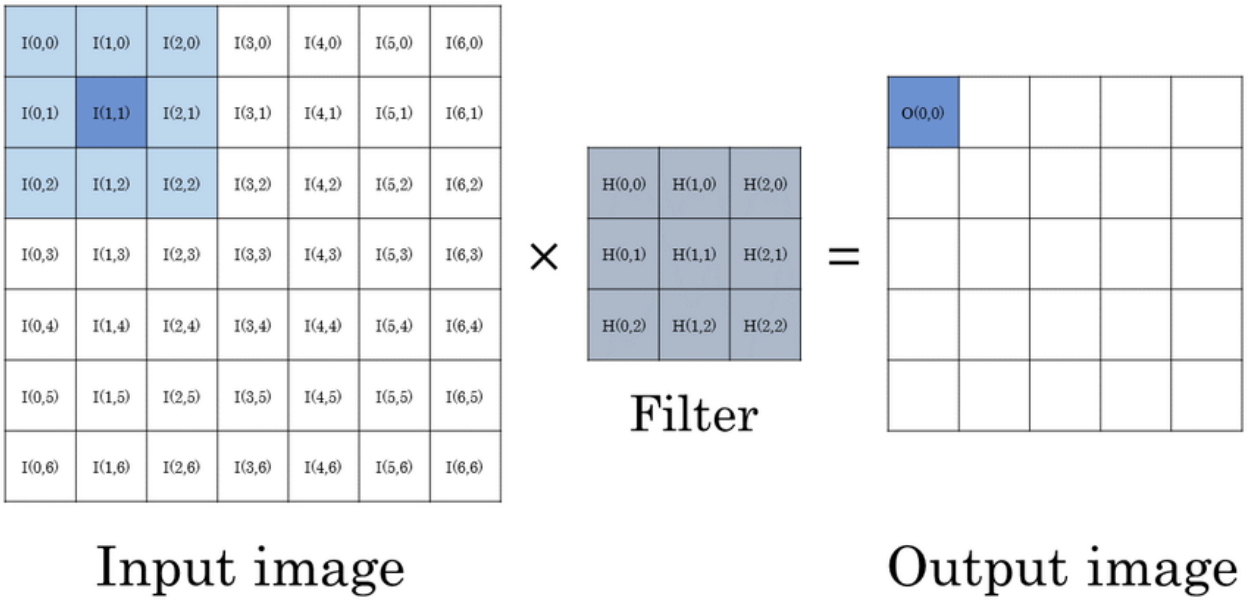


Рис. 1 Фильтрация изображения операцией математической свертки

В качестве ядра свертки будет использовано ядро с распределением функцией Гаусса:

$$\begin{bmatrix} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{bmatrix} \quad (2)$$

Тем самым алгоритм представляет из себя операцию сглаживания изображения.

1. Реализация фильтрации изображения в Vivado HLS

1.1 Алгоритм фильтрации изображения

В листинге 1 приведена простая реализация синтезируемой функции фильтрации изображения без оптимизации. В функции *gauss_blur* на вход подается исходное изображение размером *NxM*, ядро свертки и выходное преобразованное изображение. В функции создаются локальные буферы один для ядра и два для входного изображения. Один локальный буфер выполняет функцию скользящего окна, перемещаясь по изображению, сохраняет текущую область изображения в памяти. Второй буфер сохраняет предыдущее значение строк изображения. Такие промежуточные буферы сокращают

обращения к внешнему порту, то есть самому изображению. В цикле данные изображения записываются один раз в переменную и сохраняются в локальные буферы. Также учитываются границы изображения в условиях ветвления. При попадании на границу, в выходное изображение записывается ноль, иначе выполняется свёртка. В коде приведена вспомогательная функция *filter*, которая выполняется операцию свертки над областью изображения и ядром свертки.

Листинг 1

```
/**
 * Функция filter - выполняет операцию свертки над двумя массивами
 * Аргументы: kernel - ядро свертки размером KxK, содержит коэффициенты
 *            window - окно изображения размером KxK, содержит элементы
части изображения
 * Возвращает результат операции свертки - скаляр типа image_t
 */
image_t filter(image_t kernel[K][K], image_t window[K][K]) {

    int result = 0; // переменная - результат операции свертки

    row_loop: for (int i = 0; i < K; i++) {
        col_loop: for (int j = 0; j < K; j++) {
            result = result + window[i][j] * kernel[i][j];
        }
    }

    // делим на 16, так как в ядре Гаусса хранится только числитель
    return (result / 16);
}

/**
 * Функция gauss_blur - выполняет фильтрацию изображения заданным ядром и
записывает
 *                      результат в выходной массив
 * Аргументы: kernel - ядро свертки размером KxK, содержит коэффициенты
 *            inImage - изображение размером NxM, содержит значения
интенсивности входного изображения
 *            outImage - изображение размером NxM, содержит значения
интенсивности выходного изображения
 */
void gauss_blur(image_t inImage[N][M], image_t gauss_kernel[K][K], image_t
outImage[N][M])
{

    image_t window[K][K];
    image_t kernel[K][K];
```

```
    // сохраняем ядро в локальное хранилище
    for (int i = 0; i < K; i++) {
        for (int j = 0; j < K; j++) {
            kernel[i][j] = gauss_kernel[i][j];
        }
    }
    image_t part_buffer[2][M]; // промежуточный буфер для сохранения
данных локально
    L2: for(int row = 0; row < N + 1; row++) {
        L1: for(int col = 0; col < M + 1; col++) {
            image_t pixel = 0;
           // считываем данные из входного изображения
            if(row < N && col < M) {
                pixel = inImage[row][col];
            }
            // скользящее окно - считываем предыдущие данные, без
считывания из входного массива
            for(int i = 0; i < 3; i++) {
                window[i][0] = window[i][1];
                window[i][1] = window[i][2];
            }

            // считываем предыдущие данные из локального буфера, без
считывания из входного массива
            if(col < M) {
                window[0][2] = part_buffer[0][col];
                window[1][2] = part_buffer[0][col] = part_buffer[1][col];
                window[2][2] = part_buffer[1][col] = pixel;
            }

            // проверка границ изображения
            if(row >= 1 && col >= 1) {
                int outrow = row - 1;
                int outcol = col - 1;
                // если на границе изображения, то результат равен нулю,
иначе выполняем свертку
                // и записываем в выходной массив
                if(outrow == 0 || outcol == 0 || outrow == (N-1) || outcol
== (M-1)) {
                    outImage[outrow][outcol] = 0;
                } else {
                    outImage[outrow][outcol] = filter(kernel, window);
                }
            }
        }
    }
}
```

В листинге 2 представлено содержимое командного файла для автоматического создания проекта. Для проекта была выбрана следующая микросхема – xa7a12tcsg325-1Q.

Листинг 2

```
############################################
#                Lab                #
############################################
open_project -reset course_prj
set_top gauss_blur
add_files ./source/course_prj.c
add_files -tb ./source/course_prj_test.c

open_solution -reset sol1
create_clock -period 6 -name clk
set_clock_uncertainty 0.1
set_part {xa7a12tcsg325-1Q}

csim_design
csynth_design
cosim_design -trace_level all
############################################
#              Solutions              #
############################################
set all_solutions {sol2 sol3 sol4 sol5}
set all_periods   {{8}  {10} {12} {16}}

foreach the_solution $all_solutions the_period $all_periods {
open_solution -reset $the_solution
create_clock -period $the_period -name clk
set_clock_uncertainty 0.1
set_part {xa7a12tcsg325-1Q}

csim_design
csynth_design
cosim_design -trace_level all
}

exit
```

Проект содержит в себе 5 решении, со следующими параметрами:

a. Для *sol1* задается *clock period* 6: *clock uncertainty* 0.1

b. Для *sol2* задается *clock period* 8: *clock uncertainty* 0.1

c. Для *sol3* задается *clock period* 10: *clock uncertainty* 0.1

d. Для *sol4* задается *clock period* 12: *clock uncertainty* 0.1

e. Для *sol5* задается *clock period* 16: *clock uncertainty* 0.1

После синтезирования функции необходимо выбрать лучшее решение, над которым в дальнейшем будут производиться оптимизации.

1.2 Тестирование алгоритма фильтрации изображения

Далее в листинге 3 представлен код теста синтезируемой функции. В функции main двумерный массив, который представляет собой изображение, заполняется тестируемыми числами и ядро для фильтрации с заполненными целыми числами. В цикле происходит вызов синтезируемой функции, далее результат сравнивается с тестовым решением в функции *cmpr_filter*. В функции *cmpr_filter* происходит сравнение эталонного решения, с помощью чтения из файла, и пользовательского решения. Эталонное решение было получено с помощью библиотеки компьютерного зрения *OpenCV* и написано на *Python* – листинг 4. В файлах с эталонным решением хранится изображение в виде массива, для тестируемых чисел – 16 и 128. На границах массива должен храниться 0, а остальные элементы, которые расположены внутри массивы, должны быть заполнены тестируемыми числами. Функция вызывается три раза с разными значениями входного массива.

Листинг 3

```
void set_value(image_t inImage[N][M], image_t value)
{
    for (int i = 0; i < N; i++){
        for(int j = 0; j < M; j++){
            inImage[i][j] = value;
        }
    }
}
```

```
int cmpr_filter(const image_t cmpImage[N][M], const char* filename) {
    FILE* fp = fopen(filename, "r");
    if (fp == NULL) {
        printf("Error: could not open file %s\n", filename);
        return 0;
    }
    int temp = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            // считываем данные из файла
            fscanf(fp, "%d", &temp);
            // сравнение текущего значения изображения со значением из
файла
            if (temp != cmpImage[i][j]) {
                printf("Value not equal! Index: (%d, %d); Values: (%d,
%d); \n", i, j, temp, cmpImage[i][j]);
                return 0;
            }
        }
    }
    fclose(fp);
    return 1;
}
int main()
{
    int pass=0;
    // файлы с эталонным решением
    const char* filenames640[] = {"testdata/test640_16.txt",
"testdata/test640_128.txt"};
    const char* filenames1280[] = {"testdata/test1280_16.txt",
"testdata/test1280_128.txt"};
    const char* filenames1980[] = {"testdata/test1920_16.txt",
"testdata/test1920_128.txt"};
    // Тестируемые значения
    const int value[] = {16, 128};
    // Call the function for 3 transactions
    image_t inImage[N][M];
    image_t outImage[N][M];
    image_t gauss_kernel[K][K] = { {1, 2, 1}, {2, 4, 2}, {1, 2, 1} };

    int t_idx = 0; // индекс, изменяющий проверяемое значение и эталонный
файл на каждой итерации
    for (int i = 0; i < 3; ++i){
        // обновление значений исходных массивов с изображением
        set_value(inImage, value[t_idx]);
        set_value(outImage, 0);

        gauss_blur(inImage, gauss_kernel, outImage);
        pass = cmpr_filter(outImage, filenames640[t_idx]);
```

```
        if (pass == 0)  {
            fprintf(stderr, "----------Fail!-----------\n");
            return 1;
        }
        t_idx = (t_idx + 1) % 2;
    }

    fprintf(stdout, "----------Pass!-----------\n");
    return 0;
}
```

Листинг 4

```
import numpy as np
import cv2

N = 640
M = 480
ddepth = -1
kernel = np.array([[1/16., 1/8., 1/16.],
                   [1/8., 1/4., 1/8.],
                   [1/16., 1/8., 1/16.]])

def Filter(value):
    img = np.zeros((N, M), dtype=np.uint8)

    for i in range(N):
        for j in range(M):
            img[i][j] = value
    # Функция OpenCV фильтрации изображения
    res = cv2.filter2D(src=img, ddepth=ddepth, kernel=kernel)

    # По границам изображения выставляется 0, в соответствии с алгоритмом
    for i in range(N):
        for j in range(M):
            if (i == 0 or j == 0 or i == (N-1) or j == (M-1)):
                res[i][j] = 0

    return np.uint8(res)

if __name__ == '__main__':
    setup_value = 16
    res = Filter(setup_value)
    with open('test{}_{}.txt'.format(N, setup_value), 'wb') as f:
        np.savetxt(f, res, fmt='%d')
    pass
```

2. Результаты синтеза функции с настройками по умолчанию

Для первоначального тестирования функции было выбрано изображение размером 640*x*480. Результаты синтезирования функции для всех решении приведен на рисунке 2 в виде сравнения отчета решении. Как видно из рисунка, все решения укладываются в заданное время. *Latency* имеет разное min и max значение, таким образом, в дальнейшем будут рассматриваться только максимальное значение этих величин. Также стоит отметить, что у решения, которое имеет *clock period* равное или большее 10 нс не изменяется *estimated time* и равняется 9.322 нс.

Performance Estimates

Timing

Clock		sol1	sol2	sol3	sol4	sol5
clk	Target	6.00 ns	8.00 ns	10.00 ns	12.00 ns	16.00 ns
	Estimated	5.790 ns	7.661 ns	9.332 ns	9.332 ns	9.332 ns

Latency

		sol1	sol2	sol3	sol4	sol5
Latency (cycles)	min	3392839	3392839	3392839	3392839	3392839
	max	16958963	13875753	10792543	10792543	10792543
Latency (absolute)	min	20.357 ms	27.143 ms	33.928 ms	40.714 ms	54.285 ms
	max	0.102 sec	0.111 sec	0.108 sec	0.130 sec	0.173 sec
Interval (cycles)	min	3392839	3392839	3392839	3392839	3392839
	max	16958963	13875753	10792543	10792543	10792543

Utilization Estimates

	sol1	sol2	sol3	sol4	sol5
BRAM_18K	3	3	3	3	3
DSP48E	1	1	1	1	1
FF	347	352	306	306	306
LUT	933	905	899	899	899
URAM	0	0	0	0	0

Рис. 2 Результат синтезирования функций для всех решении

На рисунке 3 представлены результаты в виде таблицы с посчитанным *Latency* в нс и на рисунке 4 результаты представлены в виде графика. Как видно из рисунка нет единого лучшего решения относительно время/аппаратные ресурсы. Поэтому в качестве решения для оптимизации будет выбрано решение №1 – *sol1*, так как оно имеет лучший временной

показатель. Для первого решения *Latency* = 98192396 нс = 98192,3 мкс = 98,1 мс.

		sol1	sol2	sol3	sol4	sol5
Clock	Target (ns)	6	8	10	12	16
	Estimated (ns)	5,79	7,66	9,33	9,33	9,33
Latency	(cycles)	16958963	13875753	10792543	10792543	10792543
	(ns)	98192396	106302144	100716011	100716011	100716011
Resources	BRAM_18K	3	3	3	3	3
	DSP48E	1	1	1	1	1
	FF	347	352	306	306	306
	LUT	933	905	899	899	899
	URAM	0	0	0	0	0

Рис. 3 Таблица с результатами синтезирования функций

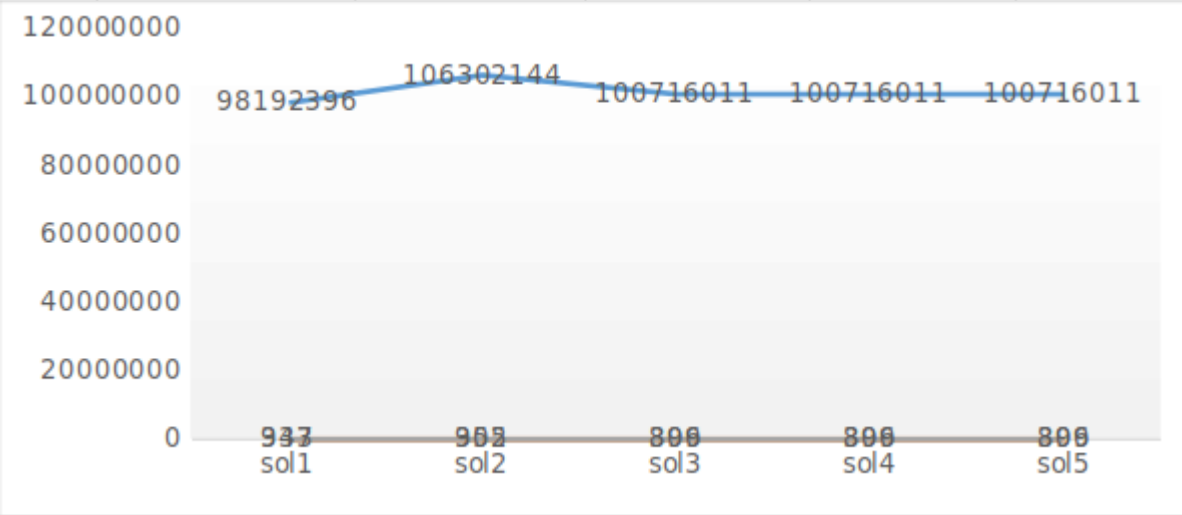


Рис. 4 График с результатами синтезирования функций

На рисунке 5 представлен performance profile для решения *sol1*. Как видно из рисунка *Iteration Latency* для цикла *L2* равно от 5293 до 26457 и *Trip count* равно 641, т. е. почти количеству пикселей в строке. Iteration Latency для цикла *L1* равно от 11 до 55, а *Trip count* равно 481, т.е. почти количеству пикселей в столбце. *Initiation Interval* для функции равен от 3392813 до 16958937.

	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
gauss_blur	-	3392839~16958963	-	3392840 ~ 16958964	-
Loop 1	no	24	8	-	3
Loop 1.1	no	6	2	-	3
L2	no	3392813 ~ 16958937	5293 ~ 26457	-	641
L1	no	5291 ~ 26455	11 ~ 55	-	481
L1.1	no	6	2	-	3
row_loop	no	42	14	-	3
col_loop	no	12	4	-	3

Рис. 5 Performance profile для *sol1*

На рисунке 6 представлены интерфейсы портов, которые были определены по умолчанию для функции.

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	gauss_blur	return value
ap_rst	in	1	ap_ctrl_hs	gauss_blur	return value
ap_start	in	1	ap_ctrl_hs	gauss_blur	return value
ap_done	out	1	ap_ctrl_hs	gauss_blur	return value
ap_idle	out	1	ap_ctrl_hs	gauss_blur	return value
ap_ready	out	1	ap_ctrl_hs	gauss_blur	return value
inImage_address0	out	19	ap_memory	inImage	array
inImage_ce0	out	1	ap_memory	inImage	array
inImage_q0	in	8	ap_memory	inImage	array
kernel_address0	out	4	ap_memory	kernel	array
kernel_ce0	out	1	ap_memory	kernel	array
kernel_q0	in	8	ap_memory	kernel	array
outImage_address0	out	19	ap_memory	outImage	array
outImage_ce0	out	1	ap_memory	outImage	array
outImage_we0	out	1	ap_memory	outImage	array
outImage_d0	out	8	ap_memory	outImage	array

Рис. 6 Port Interface для *sol1*

3. Результаты оптимизации синтезирования функции

В качестве решения для оптимизации было выбрано решение *sol1* с *clock period* 6. Далее приведен план применения оптимизации для данного решения:

1. Изменение интерфейсов портов – *ap_fifo* для входного и выходного массива изображения. Так как, данные будут считываться и записываться последовательно, то можно использовать данные типы интерфейсов. Также для повышения производительности нужно рассмотреть оптимальное конфигурирование массивов скользящего окна «*window*» и ядра свертки «*kernel*». С помощью директивы *array_partition* можно увеличить количество считываемых портов, а так как мы непрерывно считываем и записываем в эти буферы данные и они небольшого размера, это должно дать повышение производительности.

2. Разворачивание цикла внутри функции *filter* в метке *row_loop* – так как цикл состоит из небольшого количества итерации, то данный цикл можно развернуть. Такой подход должен увеличить производительность за счет устранения постоянных проверок выхода из цикла и возвращения по метке – *loop back*, изменяя программный счетчик. При этом не должно сильно увеличиться количество, затрачиваемых, аппаратных ресурсов.

3. Добавление конвейеризации в цикле *L1*. Также под циклом *L1* будут развернуты все внутренние циклы, что должно дать повышение производительности.

3.1 Изменение интерфейсов портов и конфигурации массивов

На рисунке 7 представлены директивы, добавленные для повышения производительности. Данные директивы вынесены в отдельное решение – *sol6*. Интерфейс *ap_fifo* был добавлен для входного массива(изображения) и выходного массива. Настройки данной директивы были установлены по умолчанию. Изменение конфигураций массивов представлено в виде добавления директивы *array_partition* с параметром *cycle* и *factor*=3 для *dim*=1.

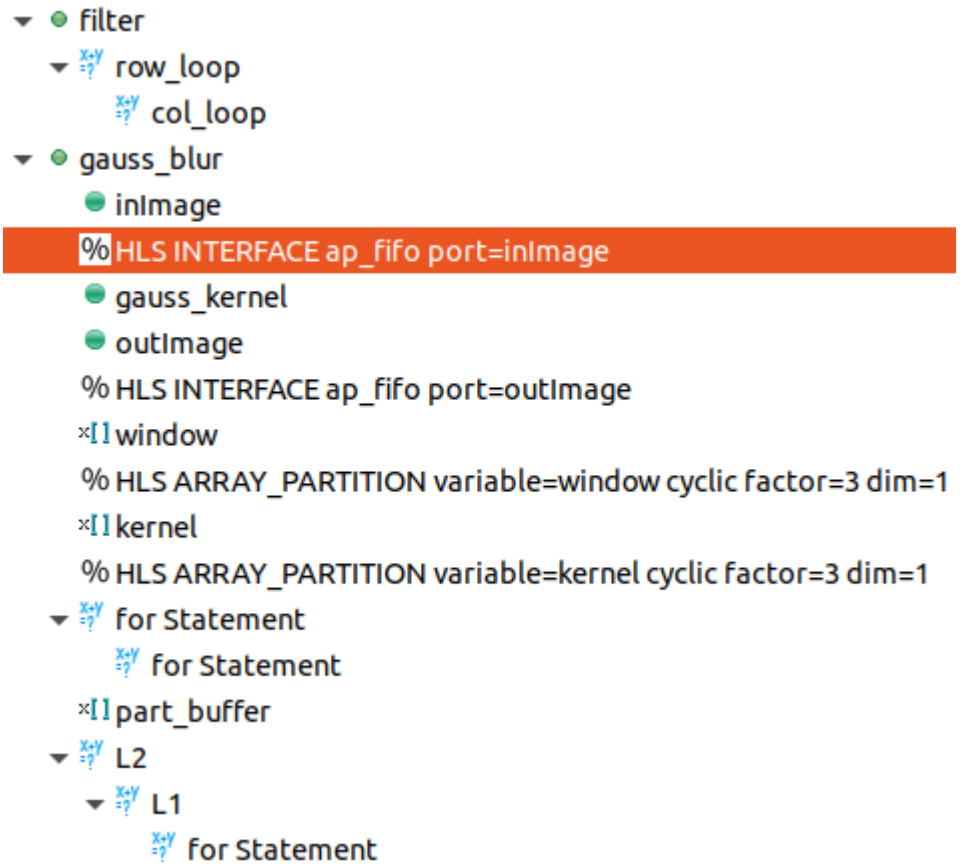


Рис. 7 Директивы, добавленные для *sol6*

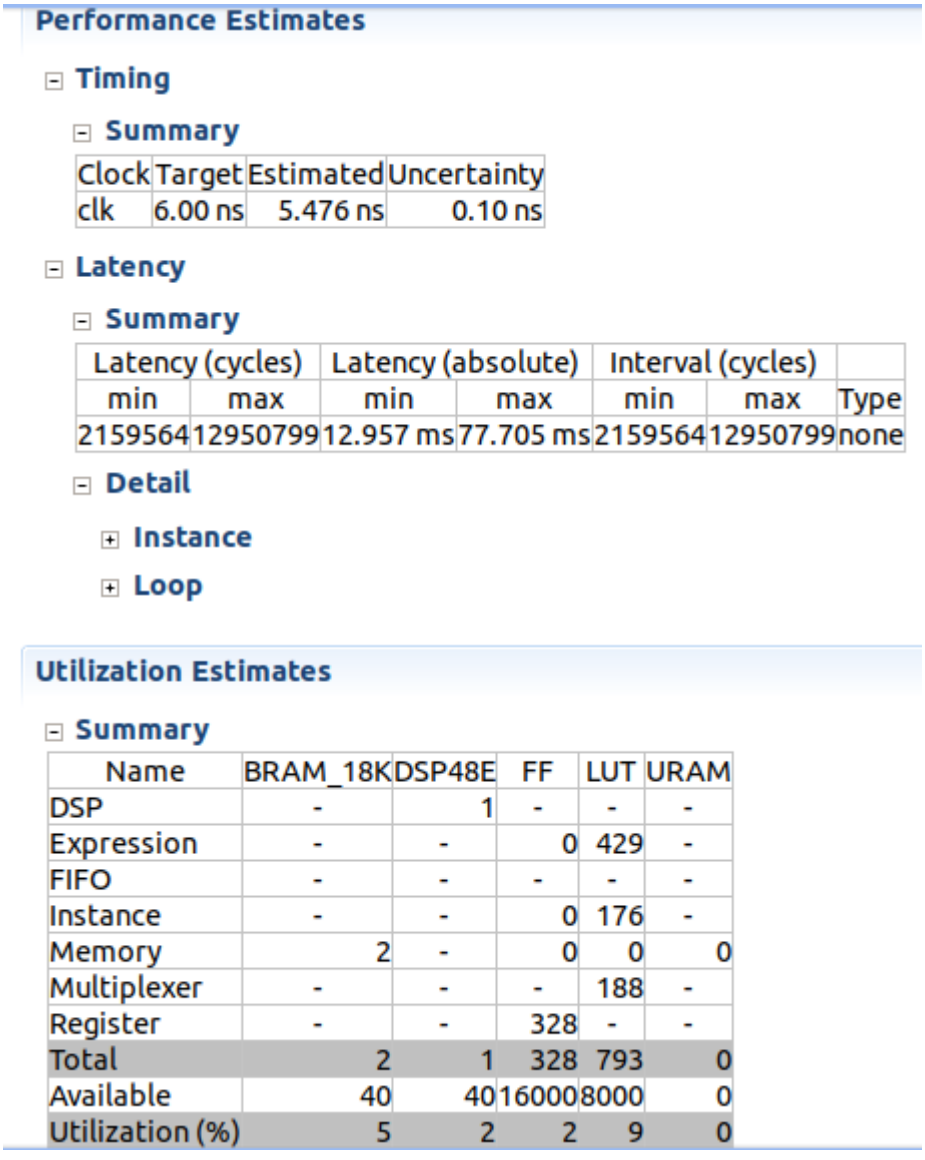
Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	gauss_blur	return value
ap_rst	in	1	ap_ctrl_hs	gauss_blur	return value
ap_start	in	1	ap_ctrl_hs	gauss_blur	return value
ap_done	out	1	ap_ctrl_hs	gauss_blur	return value
ap_idle	out	1	ap_ctrl_hs	gauss_blur	return value
ap_ready	out	1	ap_ctrl_hs	gauss_blur	return value
inImage_dout	in	8	ap_fifo	inImage	pointer
inImage_empty_n	in	1	ap_fifo	inImage	pointer
inImage_read	out	1	ap_fifo	inImage	pointer
gauss_kernel_address0	out	4	ap_memory	gauss_kernel	array
gauss_kernel_ce0	out	1	ap_memory	gauss_kernel	array
gauss_kernel_q0	in	8	ap_memory	gauss_kernel	array
outImage_din	out	8	ap_fifo	outImage	pointer
outImage_full_n	in	1	ap_fifo	outImage	pointer
outImage_write	out	1	ap_fifo	outImage	pointer

Рис. 8 Интерфейсы портов *sol6*

На рисунке 8 представлены интерфейсы портов, после синтезирования с данными директивами. Далее на рисунке 9 приведен результат синтезирования функции после добавления новых директив – *performance* и *utilization estimates*. Как видно из рисунка, после изменения интерфейсов и конфигурации массивов, производительность изменилась в лучшую сторону. *Latency* стало 70918575.32 нс = 70918.5 мкс = 70.9 мс. Время уменьшилось на ~28 миллисекунд. Также стоит отметить, что уменьшилось количество затрачиваемых аппаратных ресурсов – уменьшилось количество *FF* триггеров на 19 и *LUT* 140. Для последующих оптимизации будут использоваться данные директивы, так как они дают повышение производительности.



Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
clk	6.00 ns	5.476 ns	0.10 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
2159564	12950799	12.957 ms	77.705 ms	2159564	12950799	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	1	-	-	-
Expression	-	-	0	429	-
FIFO	-	-	-	-	-
Instance	-	-	0	176	-
Memory	2	-	0	0	0
Multiplexer	-	-	-	188	-
Register	-	-	328	-	-
Total	2	1	328	793	0
Available	40	40	16000	8000	0
Utilization (%)	5	2	2	9	0

Рис. 9 *Performance* и *utilization estimates* для *sol7*

3.2 Добавление unroll в функции filter

Добавление данной директивы было вынесено в отдельное решение – *sol7*. На рисунке 10 представлены все добавленные директивы. Как видно из рисунка, были оставлены директивы из прошлой оптимизации и добавлена директива *unroll* с флагом *-skip_check_exit* для метки *row_loop* в функции *filter*.

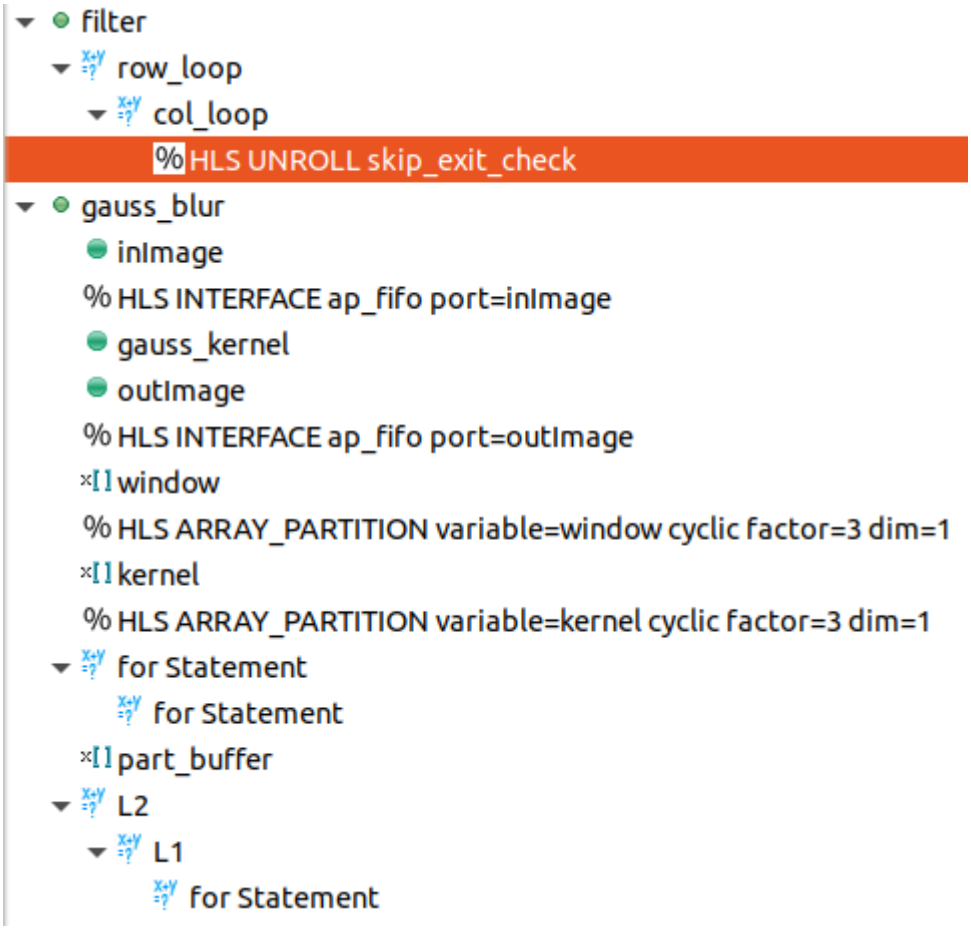
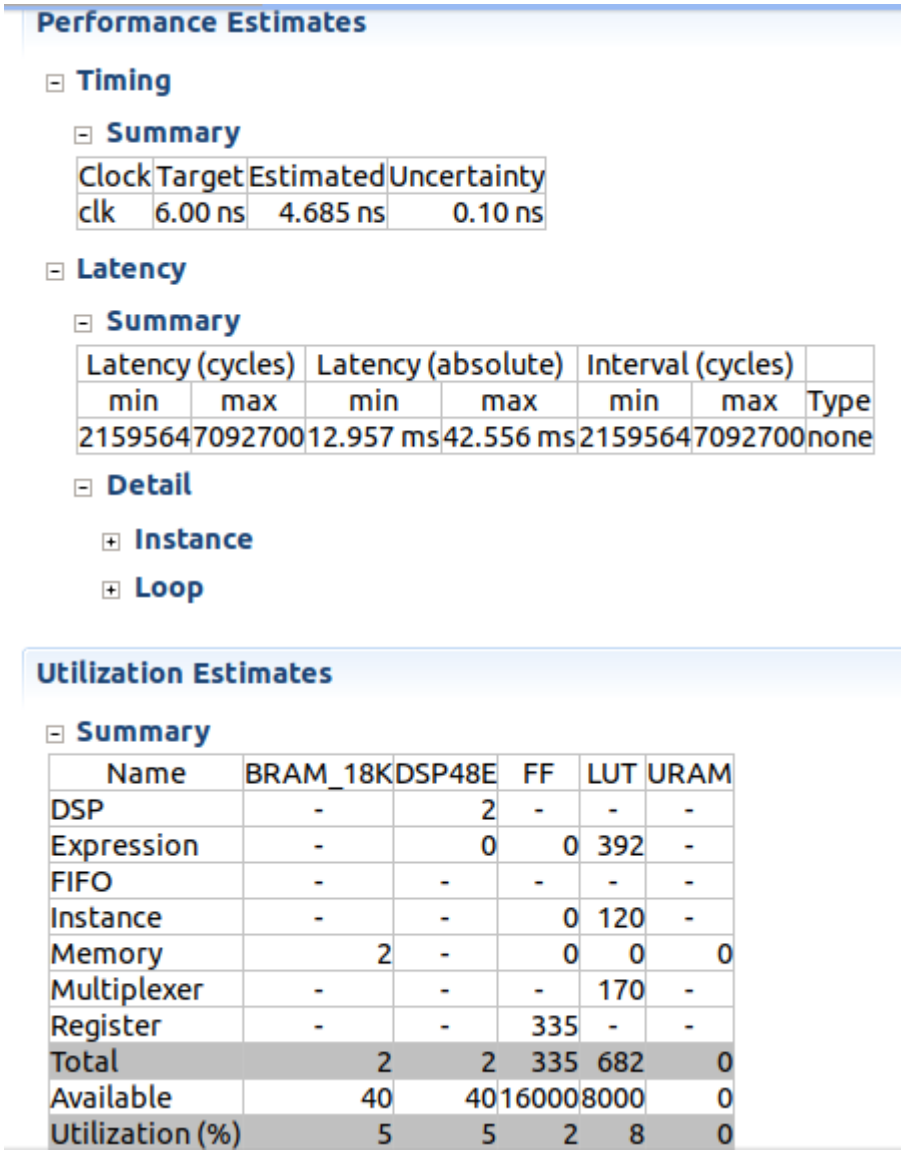


Рис. 10 Директивы, добавленные для *sol7*

Далее на рисунке 11 приведен результат синтезирования функции после добавления новых директивы *unroll* – *performance* и *utilization estimates*. Как видно из рисунка, после применения развертки цикла производительность улучшилась. *Latency* стало 33229299.5 нс = 33229.2 мкс = 33.2 мс. Время стало почти в 2 раза меньше, чем результат с добавлением интерфейсов и почти в 3 раза меньше, чем без добавления оптимизации. Также изменилось количество требуемых аппаратных ресурсов – *FF* увеличилось, а *LUT* уменьшилось.



Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
clk	6.00 ns	4.685 ns	0.10 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
2159564	7092700	12.957 ms	42.556 ms	2159564	7092700	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	2	-	-	-
Expression	-	0	0	392	-
FIFO	-	-	-	-	-
Instance	-	-	0	120	-
Memory	2	-	0	0	0
Multiplexer	-	-	-	170	-
Register	-	-	335	-	-
Total	2	2	335	682	0
Available	40	40	16000	8000	0
Utilization (%)	5	5	2	8	0

Рис. 11 *Performance* и *utilization estimates* для *sol7* с *unroll*

3.3 Добавление pipeline

Добавление данной директивы было вынесено в отдельное решение – *sol8*. На рисунке 12 представлены все добавленные директивы. Как видно из рисунка, были оставлены директивы из прошлой оптимизации и добавлена директива *pipeline* для цикла *L1*.

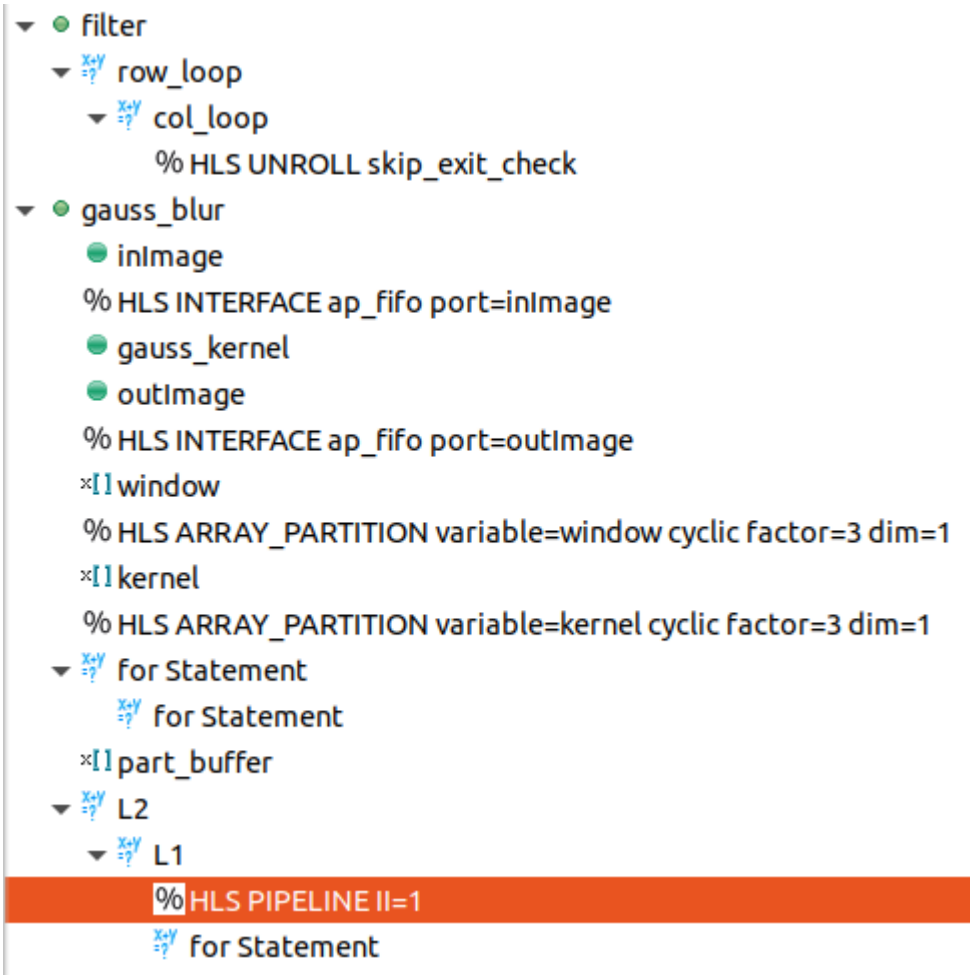


Рис. 12 Директивы, добавленные для *sol8*

Далее на рисунке 13 приведен результат синтезирования функции после добавления новых директивы *pipeline* – *performance* и *utilization estimates*. Как видно из рисунка, после добавления конвейеризации производительность увеличилась. *Latency* стало 1786963.5 нс = 1786.9 мкс = 1.7 мс. Время стало в ~19 раз меньше, чем результат с предыдущими оптимизациями и в ~57 раз меньше, чем без добавления оптимизации. Также увеличилось количество требуемых аппаратных ресурсов – увеличилось количество триггеров *FF* и *LUT*.

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
clk	6.00 ns	5.795 ns	0.10 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
308363	308363	1.850 ms	1.850 ms	308363	308363	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	5	-	-	-
Expression	-	0	0	663	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	2	-	0	0	0
Multiplexer	-	-	-	185	-
Register	0	-	818	128	-
Total	2	5	818	976	0
Available	40	40	16000	8000	0
Utilization (%)	5	12	5	12	0

Рис. 13 *Performance* и *utilization estimates* для *sol8* с *pipeline*

3.4 Сравнение результатов оптимизации

На рисунке 14 представлено сравнение результатов решении до и после оптимизации в виде таблицы. На рисунке 15 данное сравнение представлено в виде графика. Как видно из рисунков значение аппаратных ресурсов почти не изменяется, если изменяется, то незначительно. Параметр производительности же, изменяется значительно. Некоторые оптимизации увеличивают производительность в десятки раз. В проекте были применены известные автору оптимизации, которые логично было рассматривать в данном алгоритме. Некоторые оптимизации, такие как добавление *pipeline* для

цикла *L2* или добавления *dataflow*, ухудшали результат по аппаратным ресурсам, либо по временным показателям, либо не изменяли результат вообще.

		sol1	sol6	sol7	sol8
Clock	Target (ns)	6	6	6	6
	Estimated (ns)	5,79	5,48	4,69	5,80
Latency	(cycles)	16958963	12950799	7092700	308363
	(ns)	98192396	70918575	33229300	1786964
Resources	BRAM_18K	3	2	2	2
	DSP48E	1	1	2	5
	FF	347	328	335	818
	LUT	933	793	682	976
	URAM	0	0	0	0

Рис. 14 Таблица с результатами синтезирования функций после оптимизации

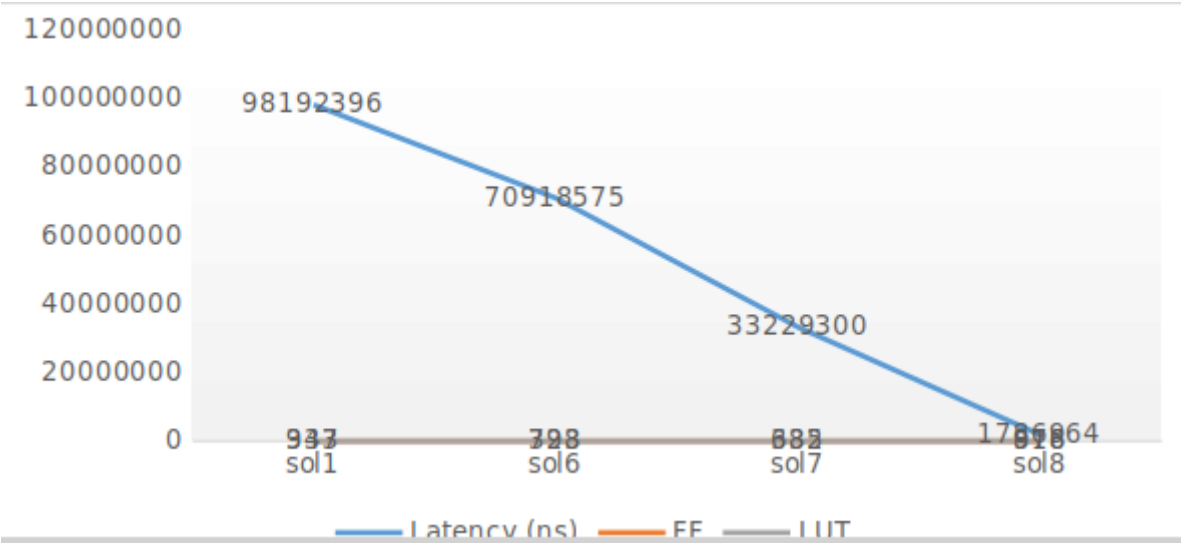


Рис. 15 График с результатами синтезирования функций после оптимизации

4. Исследование оптимального решения на разных разрешениях изображения

4.1 Разрешение изображения 1280*x*720

Результат синтезирования функции фильтрации изображения со всеми оптимизациями для разрешения 1280*x*720 представлен на рисунке 16. Как видно из рисунка *Latency* стало 6266917.75 нс = 6266.9 мкс = 6.2 мс. Также увеличилось количество требуемых аппаратных ресурсов. Таким образом, данное решение позволяет обрабатывать изображения с таким разрешением 160 кадров в секунду.

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
clk	6.00 ns	6.758 ns	0.10 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
923643	923643	6.242 ms	6.242 ms	923643	923643	none

Detail

Instance

Loop

Utilization Estimates

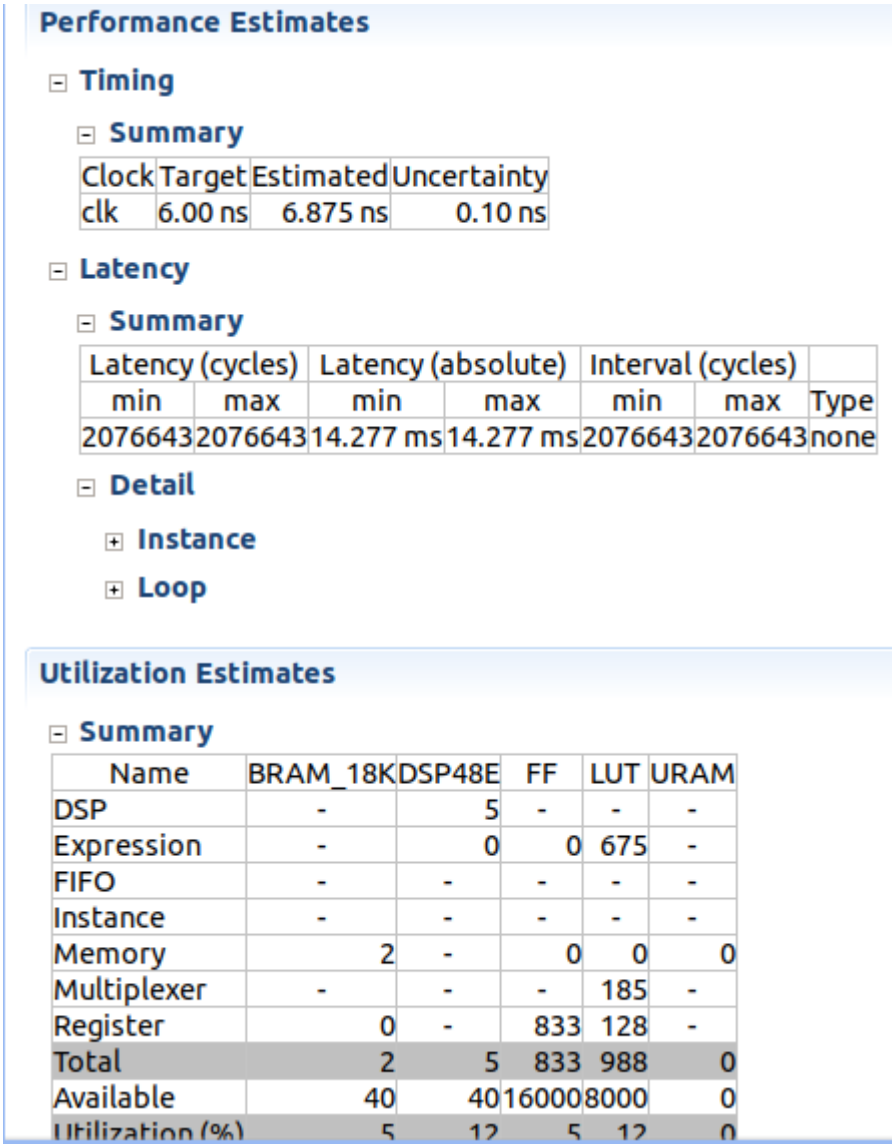
Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	5	-	-	-
Expression	-	0	0	671	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	2	-	0	0	0
Multiplexer	-	-	-	185	-
Register	0	-	827	128	-
Total	2	5	827	984	0
Available	40	40	16000	8000	0
Utilization (%)	5	12	5	12	0

Рис. 16 *Performance* и *utilization estimates* разрешение 1280*x*720

4.2 Разрешение изображения 1920x1080

Результат синтезирования функции фильтрации изображения со всеми оптимизациями для разрешения 1920*x*1080 представлен на рисунке 17. Как видно из рисунка Latency стало 14276920.62 нс = 14276.9 мкс = 14.2 мс. Также увеличилось количество требуемых аппаратных ресурсов. Таким образом, данное решение позволяет обрабатывать изображения с таким разрешением 70 кадров в секунду.



Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
clk	6.00 ns	6.875 ns	0.10 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
2076643	2076643	14.277 ms	14.277 ms	2076643	2076643	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	5	-	-	-
Expression	-	0	0	675	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	2	-	0	0	0
Multiplexer	-	-	-	185	-
Register	0	-	833	128	-
Total	2	5	833	988	0
Available	40	40	16000	8000	0
Utilization (%)	5	12	5	12	0

Рис. 17 *Performance* и *utilization estimates* разрешение 1920*x*1080

5. Исследование программной реализации функции на ПК

5.1 Реализация модифицированного теста

Исходный код модифицированного теста для запуска синтезируемой функции gauss_blur на ПК приведен в листинге 5. Тест проверки идентичен тесту проверки корректности функции для *Vivado HLS*. Добавлен замер времени выполнения функции на ПК. Количество запусков функции равно 32, где на каждой итерации замеряется среднее время выполнения функции и проверяется результат с эталонным решением из файла. Проект был собран с помощью инструмента сборки *CMake* с флагом «*-DCMAKE_BUILD_TYPE=Release*», который создает «релизную» версию проекта, то есть включены все оптимизации для компилятора. В качестве компилятора использовался *gcc-9.3.0*. В таблице 1 представлены характеристики ПК:

Таблица 1

CPU	*Intel Core i5-6200U 2.3 GHz*
Core	*2*
Threads	*4*
RAM	*8 Gb*
OS	*Linux Ubuntu 20.04 LTS*

Листинг 5

```
void set_value(image_t inImage[N][M], image_t value)
{
    for (int i = 0; i < N; i++){
        for(int j = 0; j < M; j++){
            inImage[i][j] = value;
        }
    }
}

void set_zero(image_t inImage[N][M])
{
    for (int i = 0; i < N; i++) {
        for(int j = 0; j < M; j++) {
            inImage[i][j] = 0;
        }
    }
}
```

```
int cmpr_filter(const image_t cmpImage[N][M], const char* filename) {
    FILE* fp = fopen(filename, "r");
    if (fp == NULL) {
        printf("Error: could not open file %s\n", filename);
        return 0;
    }
    int temp = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            // считываем данные из файла
            fscanf(fp, "%d", &temp);
            // сравнение текущего значения изображения со значением из
файла
            if (temp != cmpImage[i][j]) {
                printf("Value not equal! Index: (%d, %d); Values: (%d,
%d); \n", i, j, temp, cmpImage[i][j]);
                return 0;
            }
        }
    }

    fclose(fp);
    return 1;
}


int main()
{
    int pass=0;

    // имена файлов с эталонным решением
    const char* filenames640[] = {"test640_16.txt", "test640_128.txt"};
    const char* filenames1280[] = {"test1280_16.txt", "test1280_128.txt"};
    const char* filenames1980[] = {"test1920_16.txt", "test1920_128.txt"};
    // значения для проверки решения
    const int value[] = {16, 128};

    image_t inImage[N][M];
    image_t outImage[N][M];
    image_t gauss_kernel[K][K] = { {1, 2, 1}, {2, 4, 2}, {1, 2, 1} };

    struct timespec t0, t1;
    double acc_time = 0.0;
    int t_idx = 0; // индекс, изменяющий проверяемое значение и эталонный
файл на каждой итерации
    // Call the function for 32 transactions
    for (int i = 0; i < 32; ++i){
        // обновление значений исходных массивов с изображением
        set_value(inImage, value[t_idx]);
```

```
        set_zero(outImage);
       // замер времени выполнения функции фильтрации
        if(clock_gettime(CLOCK_REALTIME, &t0) != 0) {
            perror("Error in calling clock_gettime\n");
            exit(EXIT_FAILURE);
        }
        gauss_blur(inImage, gauss_kernel, outImage);
        if(clock_gettime(CLOCK_REALTIME, &t1) != 0) {
            perror("Error in calling clock_gettime\n");
            exit(EXIT_FAILURE);
        }
       // подсчет среднего для текущей итерации
        double diff_time = (((double)(t1.tv_sec -
t0.tv_sec))*1000000000.0) + (double)(t1.tv_nsec - t0.tv_nsec);
        acc_time += diff_time;
        double temp_avg_time = acc_time / (i + 1); // take average time
        printf("Elapsed time: %.4lf nanoseconds\n", temp_avg_time);
        // функция сравнения результата с эталонным решением
        pass = cmpr_filter(outImage, filenames640[t_idx]);
        if (pass == 0)  {
            fprintf(stderr, "----------Fail!----------\n");
            return 1;
        }

        t_idx = (t_idx+1) % 2;

    }

    fprintf(stdout, "----------Pass!----------\n");
    return 0;
}
```

5.2 Результаты запуска модифицированного теста на ПК

В таблице 2 представлены результаты запуска функции на ПК для разрешении 640*x*480, 1280*x*720 и 1920*x*1080 соответственно. Среднее время выполнения функции после 32 итерации для разрешения 640*x*480 *Latency* = 15757008.03 нс = 15757.0 мкс = 15.7 мс. Для разрешения в 1280*x*720 *Latency* = 46069766.21 нс = 46069.7 мкс = 46.0 мс. И для разрешения 1920*x*1080 Latency = 102750738.06 нс = 102750.7 мкс = 102.7 мс. Результаты хуже, чем в Vivado HLS.

Таблица 2

Разрешение изображения	Среднее время выполнения ПК, мс	Среднее время выполнения Vivado, мс
640*x*480	15.7	1.7
1280*x*720	46.0	6.2
1920*x*1080	102.7	14.2

6. Сравнительный анализ аппаратного и программного решения

Результаты выполнения функции на ПК для всех разрешении являются хуже, чем решения со всеми оптимизациями в Vivado HLS, но лучше, если не рассматривать последнюю оптимизацию с добавлением *pipeline*. В таблице 2 приведены результаты аппаратного и программного решения. Как видно из таблицы, во всех решениях аппаратная реализация оказалась значительно быстрее программной. На рисунке 18 приведет график зависимости разрешения от времени выполнения функции на ПК и аппаратной реализации. Стоит также учитывать, что программная реализация не использовала многопоточность, что может значительно повысить производительность.

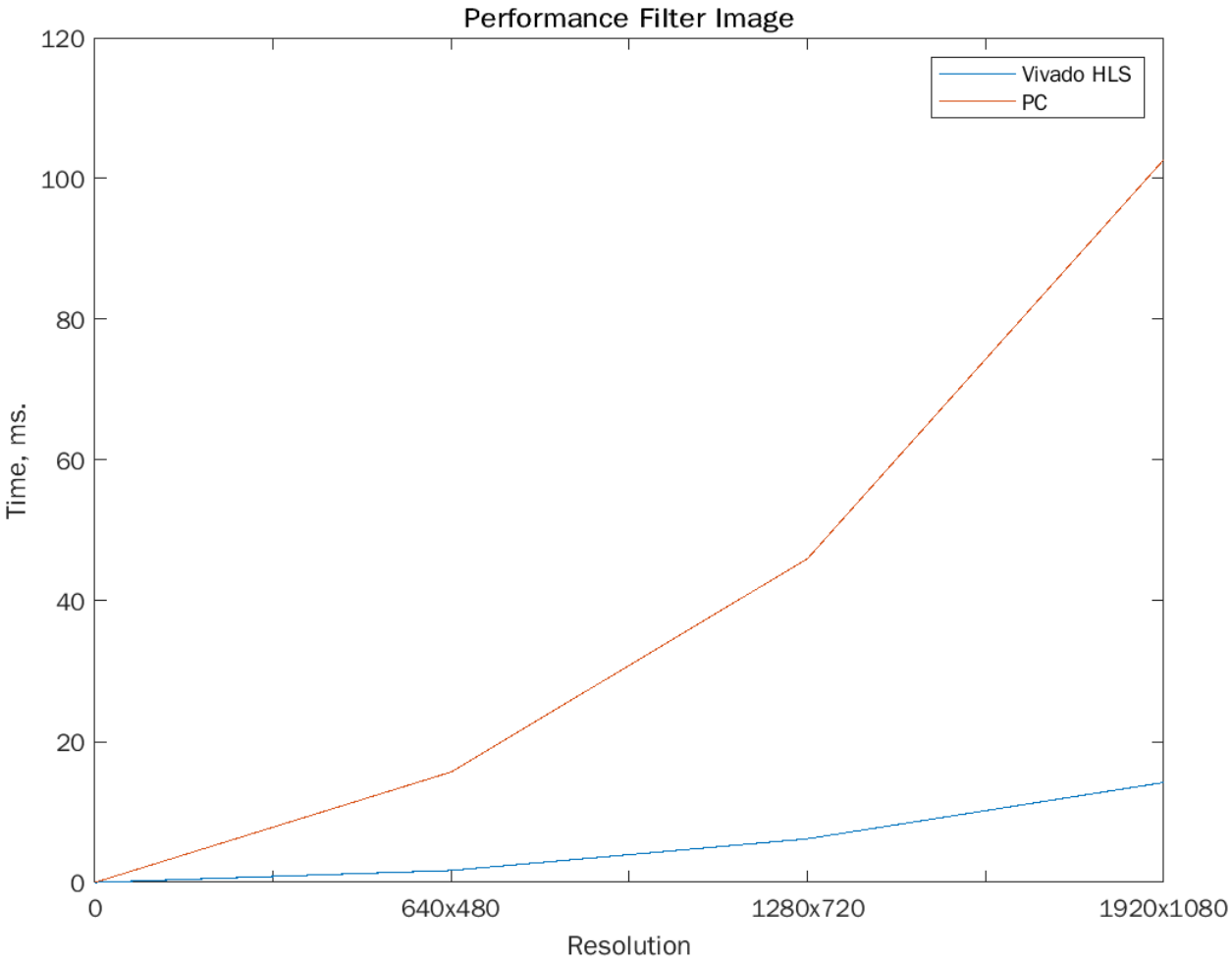


Рис. 18 График результатов программной и аппаратной реализации

Заключение

В ходе курсового проекта была разработана функция фильтрации изображении. Были рассмотрены и проанализированы различные методики оптимизации программного кода в среде *Vivado HLS*. Как видно из результатов при правильном пониманий алгоритма можно значительно ускорить время выполнения функции. Всё, что потребовалось для улучшения результата – это добавление директив для оптимизации.

Также стоит отметить, что полученный результат аппаратного решения получился значительно лучше, чем программная реализация. Единственное, что не учитывалось при сравнениях результатов это добавление многопоточной реализации для версий ПК, хотя инициализации потоков в системе может вызывать накладные расходы. Стоит учитывать количество затрачиваемых аппаратных ресурсов, особенно это касается при обработке изображении более высокого разрешения. Например, в работе [3] представлен пример, как можно сократить количество *LUT* добавлением дополнительного цикла при проверке условия.

Другие типы оптимизации были также рассмотрены, но не включены в работу, так как не принесли никаких положительных результатов. Например, добавление *dataflow* директивы в реализацию функции уменьшило *Latency* или добавление *pipeline* для цикла *L2* значительно увеличило количество аппаратных ресурсов, что не является приемлемым.

Список использованных источников

1. Rafael C. Gonzalez and Richard E. Woods. 2008. Digital Image Processing. Prentice Hall, Upper Saddle River, N.J.

2. Baskin C. et al. Streaming architecture for large-scale quantized neural networks on an FPGA-based dataflow platform //2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). – IEEE, 2018. – С. 162-169.

3. Kastner R., Matai J., Neuendorffer S. Parallel programming for FPGAs //arXiv preprint arXiv:1805.03648. – 2018.

Приложение

Проект тестировался на *Linux Ubuntu 20.04 LTS*.

Структура проекта:

/*KP_01502_10*/ – корневой каталог

/*KP_01502_10*/*doc*/ - каталог с отчетом в формате *docx* и *pdf*

/*KP_01502_10*/*source*/*course_prj.h* – заголовочный файл в котором объявляется синтезируемая функция и устанавливается разрешение изображения *N* и *M*.

/*KP_01502_10*/*source*/*course_prj.c* – исходный файл с синтезируемой функцией.

/*KP_01502_10*/*source*/*course_prj_test.c* – исходный файл с тестированием функции.

/*KP_01502_10*/*source*/*course_prj_modify.c* – исходный файл с тестированием функции на ПК.

/*KP_01502_10*/*source*/*testdata/* – каталог, в котором хранятся файлы с эталонными данными

/*KP_01502_10*/*source*/*cvFilter.py* – *Python* скрипт для создания эталонного решения

/*KP_01502_10*/*course_cmd.tcl* – командный файл для создания проекта.

/*KP_01502_10*/source/*CMakeLists.txt* – файл описания сборки проекта для *CMake*. Для компиляции перейти в каталог: «../*KP_01502_10*/*source*/», далее создать каталог для сборки и перейти в него: «*mkdir build*» => «*cd build*» и собрать проект «*cmake* ..» => «*make*».