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** 3](#_Toc89248153)

[**Введение** 3](#_Toc89248154)

[**1. Реализация фильтрации изображения в Vivado HLS** 4](#_Toc89248155)

[**1.1 Алгоритм фильтрации изображения** 4](#_Toc89248156)

[**1.2 Тестирование алгоритма фильтрации изображения** 8](#_Toc89248157)

[**2. Результаты синтеза функции с настройками по умолчанию** 11](#_Toc89248158)

[**3. Результаты оптимизации синтезирования функции** 14](#_Toc89248159)

[**3.1 Изменение интерфейсов портов и конфигурации массивов** 14](#_Toc89248160)

[**3.2 Добавление unroll в функции filter** 17](#_Toc89248161)

[**3.3 Добавление pipeline** 18](#_Toc89248162)

[**3.4 Сравнение результатов оптимизации** 20](#_Toc89248163)

[**4. Исследование оптимального решения на разных разрешениях изображения** 22](#_Toc89248164)

[**4.1 Разрешение изображения 1280*x*720** 22](#_Toc89248165)

[**4.2 Разрешение изображения 1920x1080** 23](#_Toc89248166)

[**5. Исследование программной реализации функции на ПК** 24](#_Toc89248167)

[**5.1 Реализация модифицированного теста** 24](#_Toc89248168)

[**5.2 Результаты запуска модифицированного теста на ПК** 26](#_Toc89248169)

[**6. Сравнительный анализ аппаратного и программного решения** 28](#_Toc89248170)

[**Заключение** 29](#_Toc89248171)

[**Список использованных источников** 30](#_Toc89248172)

[**Приложение** 31](#_Toc89248173)

# **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:

(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.

Изображение выглядит как стол

Автоматически созданное описание

Рис. 1 Фильтрация изображения операцией математической свертки

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

(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.

Изображение выглядит как стол

Автоматически созданное описание

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.

Изображение выглядит как стол

Автоматически созданное описание

Рис. 3 Таблица с результатами синтезирования функций

Изображение выглядит как стол

Автоматически созданное описание

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.

Изображение выглядит как текст

Автоматически созданное описание

Рис. 5 Performance profile для *sol1*

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

Изображение выглядит как стол

Автоматически созданное описание

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.

Изображение выглядит как текст

Автоматически созданное описание

Рис. 7 Директивы, добавленные для *sol6*

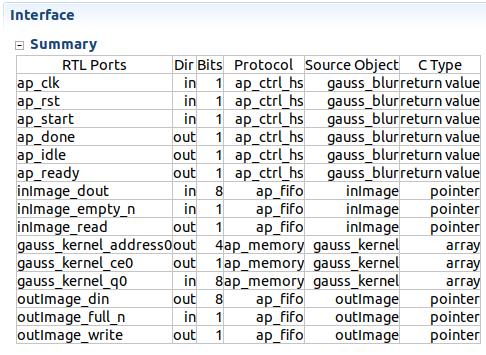


Рис. 8 Интерфейсы портов *sol6*

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.

Изображение выглядит как стол

Автоматически созданное описание

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*.

Изображение выглядит как текст

Автоматически созданное описание

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.

Изображение выглядит как стол

Автоматически созданное описание

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*.

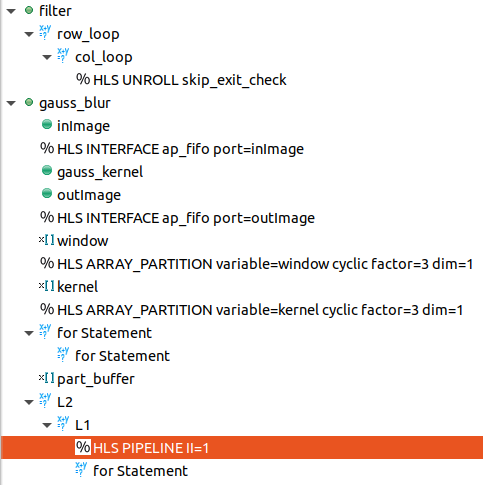


Рис. 12 Директивы, добавленные для *sol8*

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*.

Изображение выглядит как стол

Автоматически созданное описание

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.

Изображение выглядит как стол

Автоматически созданное описание

Рис. 14 Таблица с результатами синтезирования функций после оптимизации

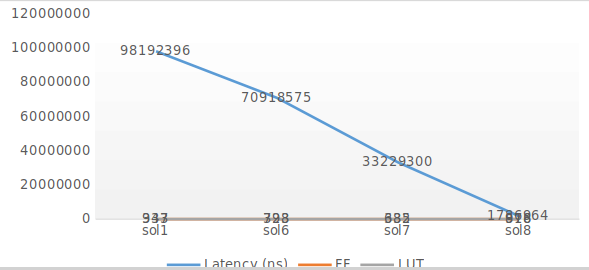


Рис. 15 График с результатами синтезирования функций после оптимизации

# **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.

**Изображение выглядит как стол

Автоматически созданное описание**

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.

Изображение выглядит как стол

Автоматически созданное описание

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.

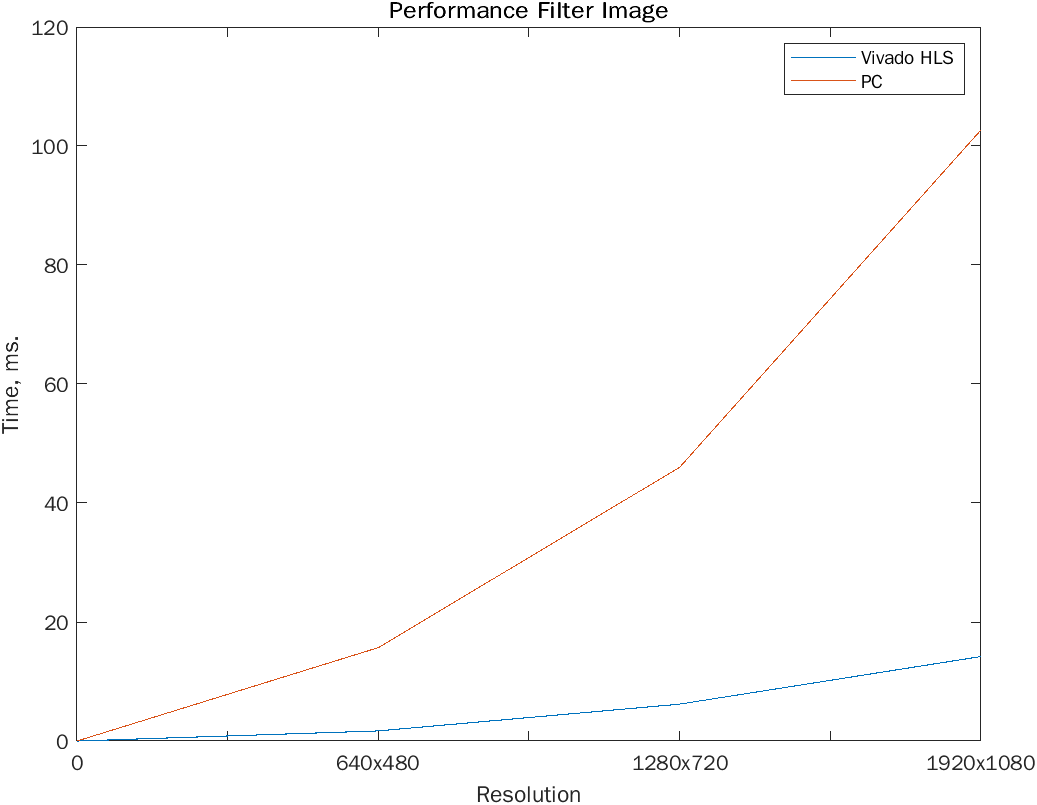


Рис. 18 График результатов программной и аппаратной реализации

# **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*».