## Оглавление

## 1. ЦЕЛЬ РАБОТЫ

Рассмотреть влияние различных методов оптимизации на время выполнения реальной программы, а также рассмотреть способы уменьшения исполняемых файлов программ на примере программы «Hello world».

## 2. ЗАДАЧИ

1. Проанализировать исходную задачу и её реализацию. Определить недостатки существующего решения и возможные пути оптимизации. Измерить время выполнения элементов исходного варианта программы.
2. Оптимизировать программу на уровне алгоритма. Измерить скорость работы программы после внесения оптимизаций.
3. Оптимизировать программу на машинно-независимом уровне. Измерить скорость работы программы после внесения оптимизаций.
4. Оптимизировать программу на машинно-зависимом уровне. Измерить скорость работы программы после внесения оптимизаций.
5. Провести анализ полученных результатов.
6. Оптимизировать по размеру исполняемый файл программы “Hello world”.
7. Сделать выводы на основании проделанной работы

# 3. ХОД РАБОТЫ

## 4. Анализ существующего решения

Постановка задачи

В качестве программы для оптимизации была выбрана курсовая работа «Раскрашивание карты» по дисциплине «Структуры данных». Условие решаемой задачи описывается так: необходимо разработать консольную утилиту, на вход которой подаётся изображение формата BMP, представляющее карту из белых областей, разделённых чёрными линиями-границами (допустимая толщина линий не предопределена). Результатом выполнения программы должно быть изображение формата BMP с раскрашенными областями, при этом никакие смежные области не должны иметь один и тот же цвет.

Анализ решения

Основной задачей является раскраска областей, поэтому оптимальным вариантом её решения является раскрашивание графа связей областей исходного изображения.

Решение данной задачи можно разделить на два основных модуля:

1. Обработка входного изображения.
    a. Определение областей.
    b. Построение графа связей смежных областей.
2. Раскрашивание областей.
    a. Раскрашивание графа.

Так как содержимое и объём входных данных неоднородны, необходимо универсальное и однозначное представление информации обо всех областях. Таким представлением является маска исходного изображения, по которой для каждого пикселя исходного изображения определяется его причастность к конкретной области. Так же, это представление удобно для использования последующих этапов алгоритма. Важнейшей чертой такого представления является то, что объёмы ресурсов, затрачиваемые на его формирование, сильно зависят от объёма входных данных, так как необходимо описать каждый пиксель или их группы. То есть нераспределённый алгоритм формирования такой маски

характеризуется плохой масштабируемостью. То же самое можно сказать и об алгоритме построения графа связей по такой маске. В виду того, что области могут пересекаться в любом количестве пикселей границы, формируется необходимость проверять каждый из них.

Так же на время выполнения программы сильно влияет вид представления графа связей. Одним из вариантов такого представления является дерево со списком смежных областей. Такая структура данных имеет место быть, однако битовая матрица смежностей менее требовательна к ресурсам памяти и открывает возможность использования битовых операций при раскрашивании графа, которые на некоторых машинах исполняются процессором за один такт.

Сама задача раскраски графа является нетривиальной, поэтому было выбрано существующее её решение – жадная раскраска.

Таким образом алгоритм решения данной задачи разбивается на этапы:

1. Формирование маски изображения.
2. Определение областей в маске.
3. Построение графа связи областей.
4. Раскрашивание графа.
5. Применение маски к выходному изображению, в соответствии с определённым цветов.

Для определения результатов оптимизации важно определить по каким параметрам входных данных следует оценивать время выполнения модулей:

1. Обработка входного изображения: размер входного изображения.
2. Раскрашивание графа: количество областей.

## Анализ существующей реализации

С помощью средств профилирования Microsoft Visual Studio 2019 был проведён анализ времени, затрачиваемого на выполнение этапов алгоритма. Результаты профилирования представлены на Рисунке 1.

| Function Name | Number of Calls | Elapsed Inclusive Time % | Module Name |
|---|---|---|---|
| mainCRTStartup | 1 | 99,75% | mtg_old.exe |
| __scrt_common_mai... | 1 | 98,44% | mtg_old.exe |
| main | 1 | 98,44% | mtg_old.exe |
| init_MField | 1 | 63,50% | mtg_old.exe |
| buildRow | 1 000 | 60,75% | mtg_old.exe |
| fread | 2 000 007 | 31,59% | ucrtbase.DLL |
| defineAreasIds | 1 | 30,39% | mtg_old.exe |
| fseek | 2 002 006 | 29,36% | ucrtbase.DLL |
| __stdio_common_vfp... | 75 088 | 20,70% | ucrtbase.DLL |
| printf | 75 087 | 20,54% | mtg_old.exe |
| ProgBar_update | 11 579 | 20,28% | mtg_old.exe |
| setAreaId_ByPoint | 37 | 13,38% | mtg_old.exe |
| setColumnId_withDir | 56 730 | 12,96% | mtg_old.exe |
| put_toRes | 1 | 2,31% | mtg_old.exe |
| buildLinks | 37 | 1,40% | mtg_old.exe |
| ProgBar_display | 54 | 0,89% | mtg_old.exe |
| fwrite | 1 000 001 | 0,88% | ucrtbase.DLL |
| findNextStep | 56 662 | 0,40% | mtg_old.exe |

Рисунок 1 - Результаты профилирования.

По результатам профилирования можно сделать следующие выводы об этапах выполнения существующей реализации:

- Алгоритм формирования маски изображения является слишком ресурсозатратным (жёлтая группа).

Существующий алгоритм преобразует строки изображения в последовательность отрезков, содержащих информацию о своей длине и типе включенных пикселей (граница или область). Он применяется для сокращения необходимой памяти и экономит время при присваивании отрезка к области: исключается необходимость последовательного присваивания ячейки строки к области. Однако при больших входных данных возможный выигрыш по памяти несравним с потерями времени на формирование такого представления.

- На фоне затрат на формирование маски изображения алгоритм определения областей маски (красная группа) выглядит выгодным, однако необходимость прохода по всей границе каждой области приводит к огромным затратам при увеличении объёма входных данных.

- Использование стандартных функций работы с файлами данного алгоритма приводит к большим потерям по времени (голубая группа).
- Алгоритм применения маски и цветов к выходному изображению так же требует большего времени при больших объёмах входных данных.

Полученные выводы говорят о необходимости изменения структур данных и алгоритмов всех модулей решения.

## 5. Оптимизация на уровне алгоритма

Модуль обработки входного изображения

Представленный выше анализ позволяет заключить, что одним из вариантов исключения проблемы масштабируемости данного модуля является замена текущего алгоритма на распределённый. Такая оптимизация применима практически на всех современных персональных компьютерах, так как применение технологий параллельных вычислений для обработки изображений, в частности используемого в данной работе фреймворка OpenCL, очень распространено.

Однако для большее эффективного применения данной технологии требуется замена некоторых структур данных, которые теперь требуют больших объёмов памяти. Так, графическая информация о изображении читается разом во временный массив. Маска представляет из себя массив индексов областей, каждая ячейка которого соответствует одному пикселю.

Так же было принято решение замены структуры, представляющей граф связей областей, на битовую матрицу смежностей: бит соответствует наличию ребра между вершинами. Это решение продиктовано тем, что такое представление более выгодно по памяти и открывает возможности оптимизации раскраски графа за счёт использования битовых операций.

Новый алгоритм этого модуля выглядит так:

1. Формирование маски изображения

Ячейкам маски, которые соответствуют границе присваивается специальный индекс.

2. Определение областей изображения:

2.1. Распространение индекса от эпицентра.

Для гарантии того, что каждая область будет обработана, было выбрано правило, по которому ячейка маски становится «эпицентром». Каждому эпицентру присваивается уникальный идентификатор (индекс). Остальные ячейки ждут появления индекса у своих соседей. Таким образом, если в области появляется один эпицентр, от него «волной» соседние ячейки внутри области будут перенимать уникальный индекс эпицентра формируя «пятна».

Однако ввиду сложности взаимодействия выполняющихся потоков и неопределённости области, сложно определить систему правил, по которой каждой области будет соответствовать один эпицентр. Из этих ограничений было принято решение: эпицентром является угловая (в определённом направлении) ячейка. Это условие гарантирует присутствие хотя бы одного эпицентра в области и ограничивает их количество. Из такого метода определения эпицентра вытекает необходимость сведения индексов полученных пятен к единому общему.

2.2. Сведение индексов соседних эпицентров к одному значению.

Из каждой ячейки, которая находится на границе пятен, происходит сравнение её индекса и индекса соседних пятен. Больший индекс пятна заменяется на меньший.

На практике для формирования индексов пятен используется массив уникальных идентификаторов. На входе выполнения этого этапа значение ячейки массива соответствует её индексу. Но в ходе выполнения это значение является индексом ячейки, у которой пятно заимствует индекс (заимствование было установлено пятном младшего идентификатора).

Таким образом из этого массива формируется список заимствования идентификатора. Те ячейки, для которых индекс равняется значению, являются «родительским» пятном, от которого соседние пятна наследуют идентификатор.

То есть уже на этом этапе можно однозначно определить соответствие ячейки маски определённой области.

2.3. Понижение полученных индексов областей до последовательности натуральных чисел.

Однако для удобства соответствия индекса вершины графа и области полученные уникальные идентификаторы заменяются на последовательность натуральных чисел (начиная с единицы).

2.4. Применение индекса ко всей области.

Завершающий этап определения областей применяет соответствующий уникальный идентификатор ко всем соседним пятнам внутри одной области. Теперь каждая ячейка маски хранит уникальный идентификатор своей области.

3. Построение графа.

Ячейки границы ищут индексы областей по вертикали и горизонтали. При наличии связи у смежных областей атомарным ИЛИ устанавливается соответствующий бит в матрице смежностей.

4. Формирование выходного изображения.

Дополнением к этому модулю является применение значения цвета пикселя в соответствии с маской и полученной раскраской графа. То есть подготовка результата выполнения программы к выводу.

Результаты оптимизации на уровне алгоритма

Так как изменения на данном уровне оптимизации затронули только модуль обработки входного изображения имеет смысл оценить изменение времени выполнения данного модуля при разных размерах входного изображения и количествах областей.

Таблица 2.1. Результаты измерения времени работы программ при едином размере входных изображений (2312 x 2312) и разном числе областей.

| Имя входного файла | Количество областей | До оптимизации, с | После оптимизации, с | Ускорение |
|---|---|---|---|---|
| f1_01.bmp | 64 | 10,683 | 2,971 | 360% |
| f1_02.bmp | 256 | 12,371 | 2,458 | 503% |
| f1_03.bmp | 337 | 14,997 | 2,365 | 634% |
| f1_04.bmp | 657 | 18,233 | 3,687 | 495% |
| f1_05.bmp | 1457 | 22,776 | 3,316 | 687% |

Таблица 2.2. Результаты измерения времени работы программ при едином числе областей (184) и разных размерах входных изображений.

| Имя входного файла | Размер изображения | До оптимизации, с | После оптимизации, с | Ускорение |
|---|---|---|---|---|
| p3_01.bmp | 1500 x 1500 | 9,752 | 3,687 | 264% |
| p3_02.bmp | 2250 x 2250 | 17,371 | 4,359 | 399% |
| p3_03.bmp | 3750 x 3750 | 48,568 | 6,541 | 743% |
| p3_04.bmp | 5625 x 5625 | 150,415 | 9,967 | 1509% |
| p3_05.bmp | 7500 x 7500 | 183,751 | 10,568 | 1739% |

По полученным результатам можно сделать заключение, о том, что благодаря применённым оптимизациям обработка входных данных увеличении количества областей в среднем в 5 раз (536%), обработка больших файлов в среднем ускорена в 9 раз (931%).

## 6. Машино-независимая оптимизация

Модуль раскрашивания графа

Алгоритм раскраски графа не был изменён, однако его существующая реализация основана на представлении графа в виде дерева со списками, из чего вытекает высокая стоимость большинства задействованных операций. Такое представление может быть удобным для некоторых задач, но для решения данной задачи оптимальным является представление графа в виде битовой матрицы смежности.

Такая замена структуры данных позволяет свести операции поиска и выбора цвета к битовым операциям.

Так, вместо затратного поиска в дереве операцией BSF (bit scan forward) из столбца вершины графа рассчитывается индекс первого соседа.

Применённые машинно-независимые оптимизации

| До оптимизации | После оптимизации |
|---|---|
| Замена затратного цикла стандартной функцией поиска первого бита | |
| `int32_t index = -1;`<br>`for (char a = 0; a < 8 *`<br>`sizeof(uint32_t); a++) {`<br>`  if (mask & (1 << a)) {`<br>`  index = a; break;`<br>`  }`<br>`}` | `int32_t index = -1;`<br>`_BitScanForward(&index, mask);` |
| Избавление от лишних операций разыменовывания | |
| `for (size_t vid = 0; vid < g-`<br>`>vertex_count; vid++) {`<br>`g->order[vid]->color_id =`<br>`vertex_get_available_color(g, g-`<br>`>order[vid]->id, &used_colors);`<br>`}` | `struct vertex_t* cv = NULL;`<br>`for (size_t vid = 0; vid < g-`<br>`>vertex_count; vid++) {`<br>`cv = g->order[vid];`<br>`cv->color_id =`<br>`vertex_get_available_color(g, cv->id,`<br>`&used_colors);`<br>`}` |
| Вынесение инварианта из тела цикла | |
| `while (_BitScanForward(&n_index, mask)) {`<br>`res |= g->vertex_row[cell_index *`<br>`bitfield_cell_flags_count +`<br>`n_index].color_id;`<br>`mask &= ~(1 << n_index);`<br>`}` | `size_t cell_offset = cell_index *`<br>`bitfield_cell_flags_count;`<br>`while (_BitScanForward(&n_index, mask)) {`<br>`res |= g->vertex_row[cell_offset +`<br>`n_index].color_id;`<br>`mask &= ~(1 << n_index);` |

| | |
|---|---|
| | }<br> |
| Исключение лишних вызовов дорогих стандартных функций | |
| for (i = 0; i < ((MFieldsRows*)cMF->Row + RowPart)->length; i++, cX++) {<br>fwrite(RGBBYTES, sizeof(byte), 3, FILES.bmpresult);<br>}<br>} | fwrite(bmp->linear_sequence, sizeof(char), bmp->linear_sequence_size, bmp->output); |
| Исключение лишних условных переходов | |
| if (n.s0 == -1 && n.s1 == -1) return true;<br>return false; | return (n.s0 == -1 && n.s1 == -1); |

Результаты машино-независимой оптимизации

Таблица 3.1. Результаты измерения времени работы программ при едином размере входных изображений (2312 x 2312) и разном числе областей.

| Имя входного файла | Количество областей | До оптимизации, с | После оптимизации, с | Ускорение |
|---|---|---|---|---|
| f1_01.bmp | 64 | 10,683 | 2,039 | 524% |
| f1_02.bmp | 256 | 12,371 | 2,135 | 579% |
| f1_03.bmp | 337 | 14,997 | 2,498 | 600% |
| f1_04.bmp | 657 | 18,233 | 2,837 | 643% |
| f1_05.bmp | 1457 | 22,776 | 3,137 | 726% |

Таблица 3.2. Результаты измерения времени работы программ при едином числе областей (184) и разных размерах входных изображений.

| Имя входного файла | Размер изображения | До оптимизации, с | После оптимизации, с | Ускорение |
|---|---|---|---|---|
| p3_01.bmp | 1500 x 1500 | 9,752 | 2,690 | 363% |
| p3_02.bmp | 2250 x 2250 | 17,371 | 3,717 | 467% |
| p3_03.bmp | 3750 x 3750 | 48,568 | 5,983 | 812% |
| p3_04.bmp | 5625 x 5625 | 150,415 | 9,342 | 1610% |

| | | | | |
|---|---|---|---|---|
| p3_05.bmp | 7500 x 7500 | 183,751 | 9,687 | 1897% |

По полученным результатам можно сделать заключение, о том, что благодаря применённым оптимизациям обработка входных данных увеличении количества областей в среднем в 6 раз (+89%), обработка больших файлов в среднем ускорена в 10 раз (+99%).

## 7. Машино-зависимая оптимизация

Ассемблерные вставки

На данном уровне оптимизации некоторые операции были заменены ассемблерными вставками:

```
//_BitScanForward(&used_colors_count, ~used_colors);
__asm {
    not used_colors         ;
    bsf ebx, used_colors    ;
    mov used_colors_count, ebx;
}

/*
_BitScanForward(&bit_index, allowed);
c <<= bit_index;
*used_colors |= c;
*/
__asm {
    mov ecx, used_colors;
    bsf ebx, allowed;
    bts [ecx], ebx;
}
```

Конфигурация проекта

Так же была изменена конфигурация данного проекта в среде Visual Studio 2019. В разделе «Оптимизация» были выбраны флаги /O2 (максимальная оптимизация по скорости), /Oi (включение подставляемых функций), /Ot (предпочтение скорости кода), /GL (оптимизация всей программы).

В разделе «Создание кода» были выбраны флаги /GF (включение объединения строк), /Gy (включение компоновки на уровне функций), /Qpar (создание параллельного кода).

Результаты машино-зависимой оптимизации

Таблица 3.1. Результаты измерения времени работы программ при едином размере входных изображений (2312 x 2312) и разном числе областей.

| Имя входного файла | Количество областей | До оптимизации, с | После оптимизации, с | Ускорение |
|---|---|---|---|---|
| f1_01.bmp | 64 | 10,683 | 1,962 | 544% |
| f1_02.bmp | 256 | 12,371 | 2,005 | 617% |
| f1_03.bmp | 337 | 14,997 | 2,159 | 695% |
| f1_04.bmp | 657 | 18,233 | 2,656 | 686% |
| f1_05.bmp | 1457 | 22,776 | 2,973 | 766% |

Таблица 3.2. Результаты измерения времени работы программ при едином числе областей (184) и разных размерах входных изображений.

| Имя входного файла | Размер изображения | До оптимизации, с | После оптимизации, с | Ускорение |
|---|---|---|---|---|
| p3_01.bmp | 1500 x 1500 | 9,752 | 2,009 | 485% |
| p3_02.bmp | 2250 x 2250 | 17,371 | 2,330 | 746% |
| p3_03.bmp | 3750 x 3750 | 48,568 | 2,603 | 1866% |
| p3_04.bmp | 5625 x 5625 | 150,415 | 3,956 | 3802% |
| p3_05.bmp | 7500 x 7500 | 183,751 | 4,691 | 3917% |

По полученным результатам можно сделать заключение, о том, что благодаря применённым оптимизациям обработка входных данных увеличении количества областей в среднем в 6 раз (+47%), обработка больших файлов в среднем ускорена в 23 раза (1270%).

## 8. Результаты оптимизации

По результатам замеров времени на каждом этапе оптимизации были построены два графика, которые отображают значения коэффициента ускорения работы программы при разных входных данных.

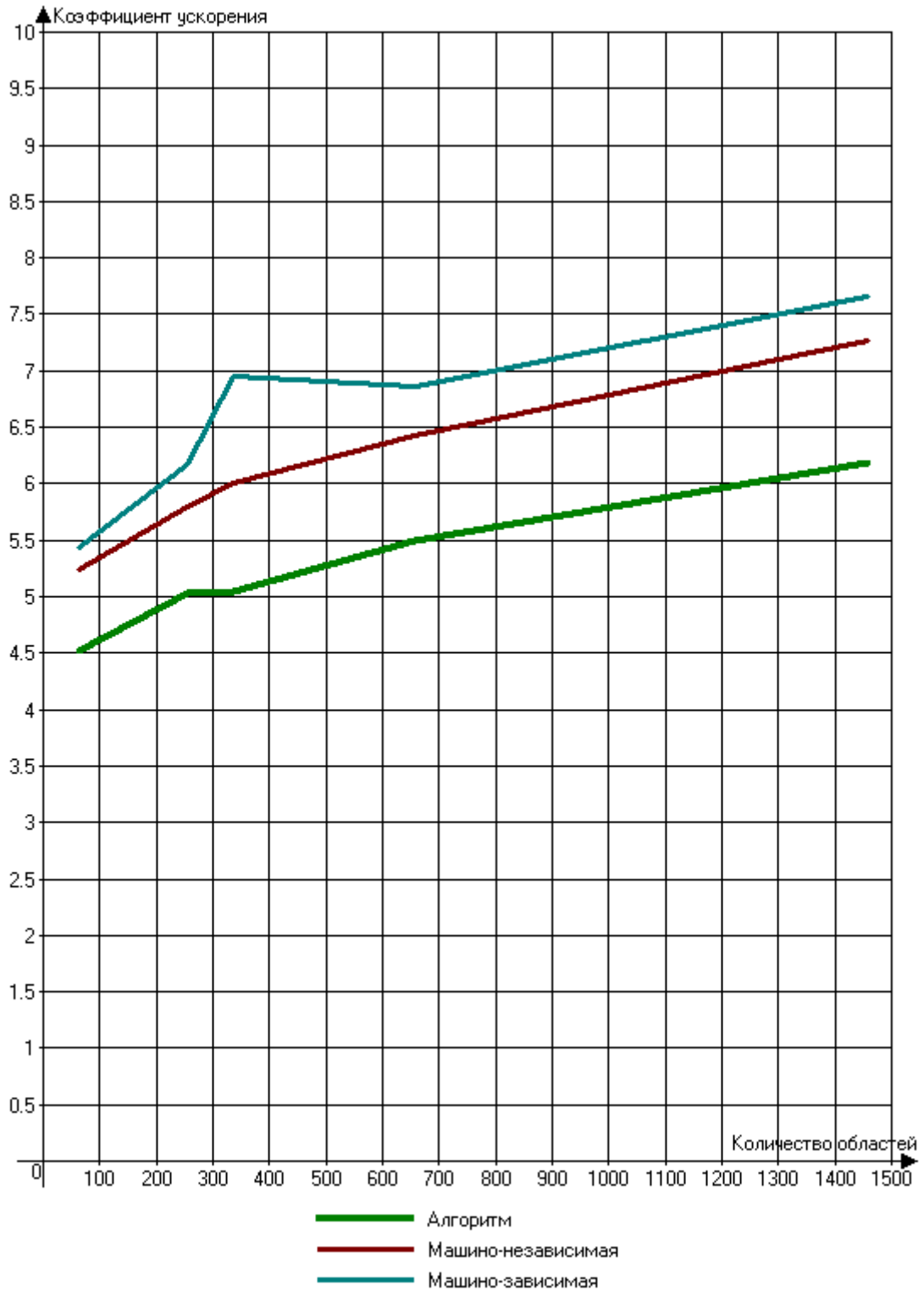


Рисунок 2 - Коэффициент ускорения при разных количествах областей.

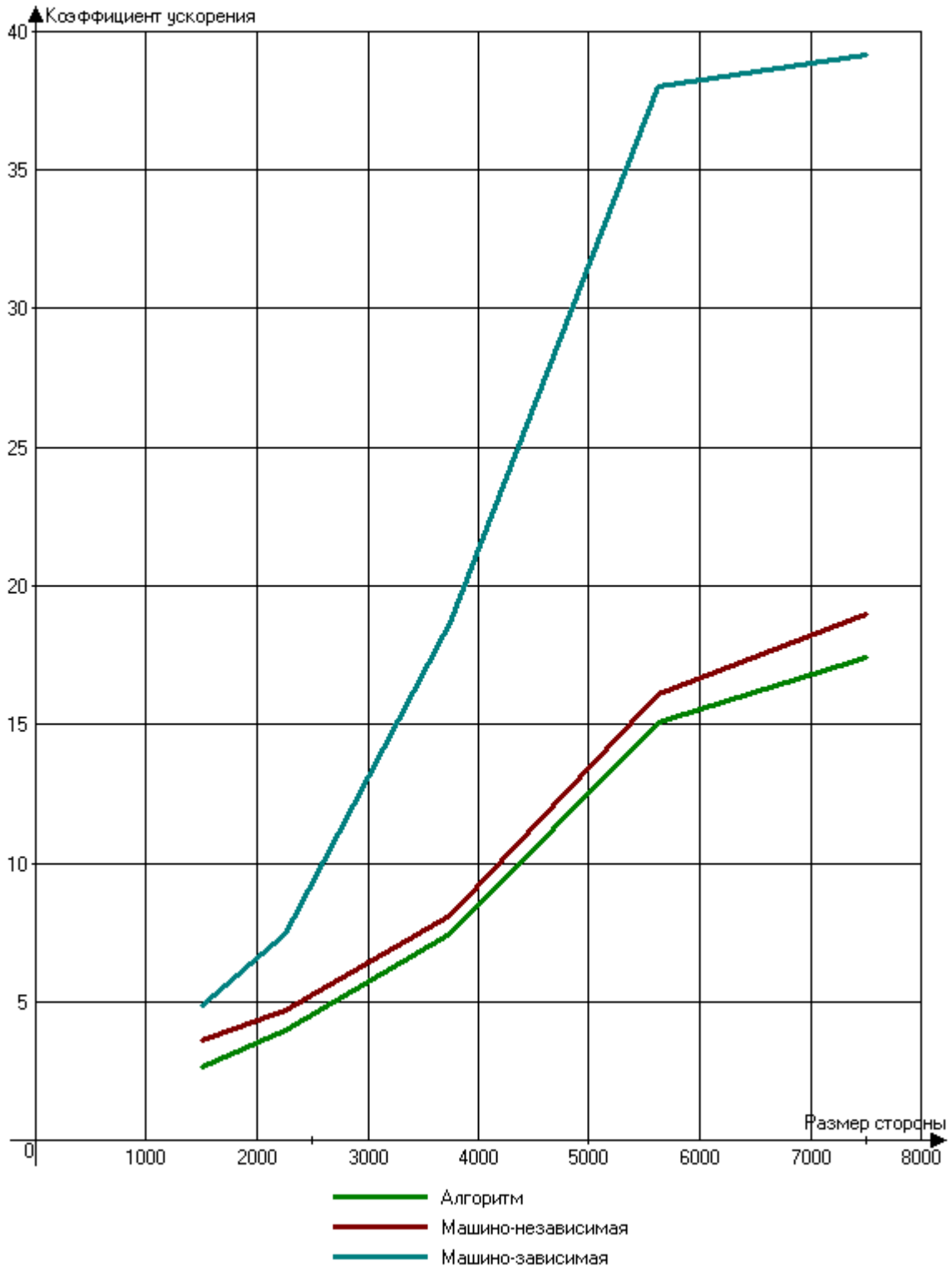


Рисунок 3 - Коэффициент ускорения при разных размерах входного изображения.

По результатам замеров времени можно сделать вывод о том, что коэффициент ускорения сильно зависит от вида входных данных, однако минимальное полученное после всех оптимизаций ускорение: 485%. Наибольшую

эффективность оптимизированная программа проявляет при работе с большими размерами входного изображения, что говорит о значительном улучшении масштабируемости новой реализации модуля обработки входного изображения.

## 9. Оптимизация Hello World

Необходимо настроить генерацию исполняемого файла так, чтобы его размер был минимальным. На Рисунке 4 представлен исходный код программы.

```
#include <stdio.h>

int main() {
    printf("Hello world");
}
```

Рисунок 4 - Исходный код оптимизируемой программы.

При генерации со стандартной конфигурацией MS Visual Studio 2019 Release размер исполняемого файла составляет 9 216 байт.

Флаги оптимизации были изменены для оптимизации по размеру. Они представлены на Рисунке 5. После этих изменений размер файла не изменился.

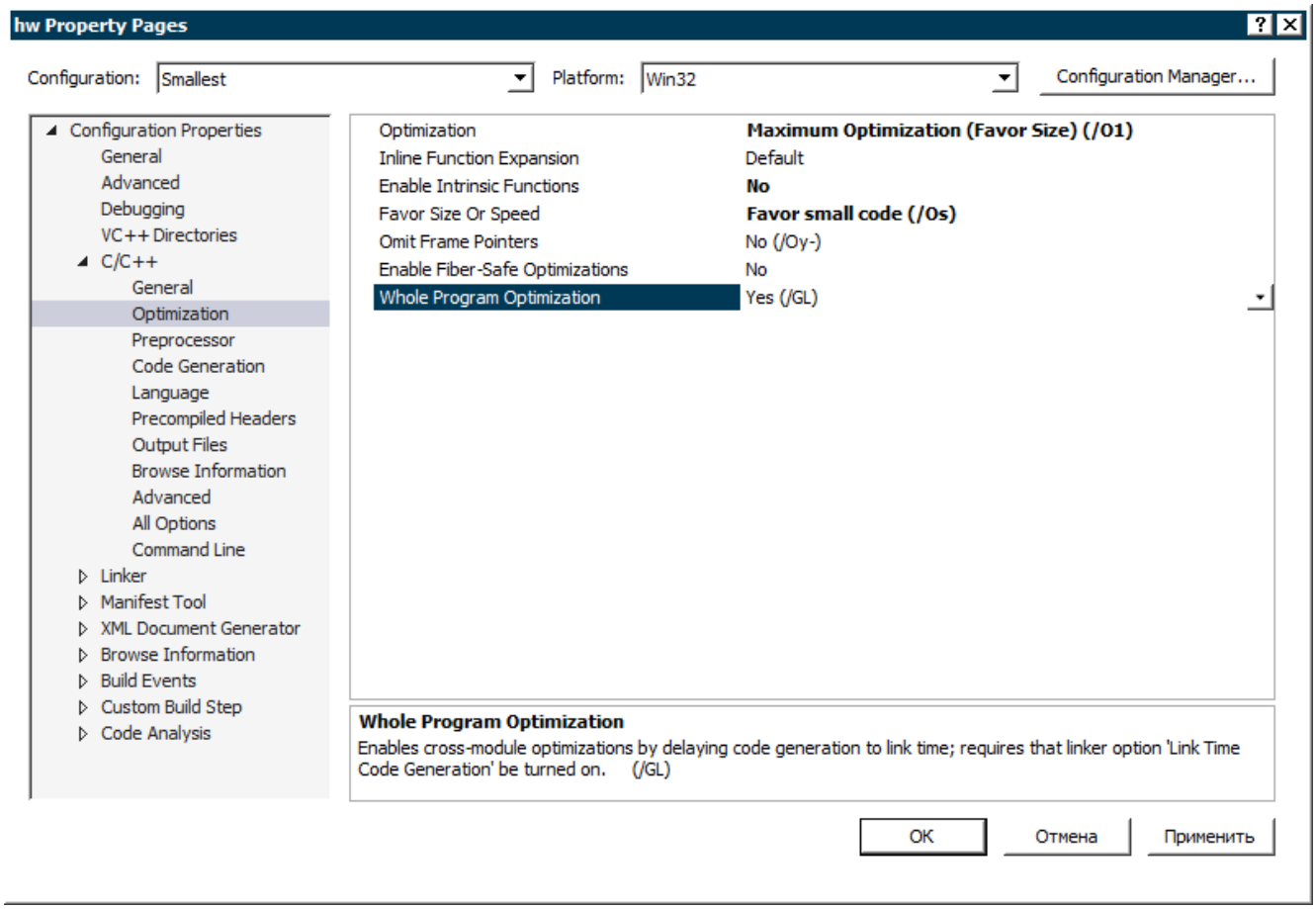


Рисунок 5 - Параметры оптимизации.

Далее изменениям были подвержены параметры генерации кода. Они представлены на Рисунке 6. Однако эти изменения тоже не оказали влияния на размер файла.

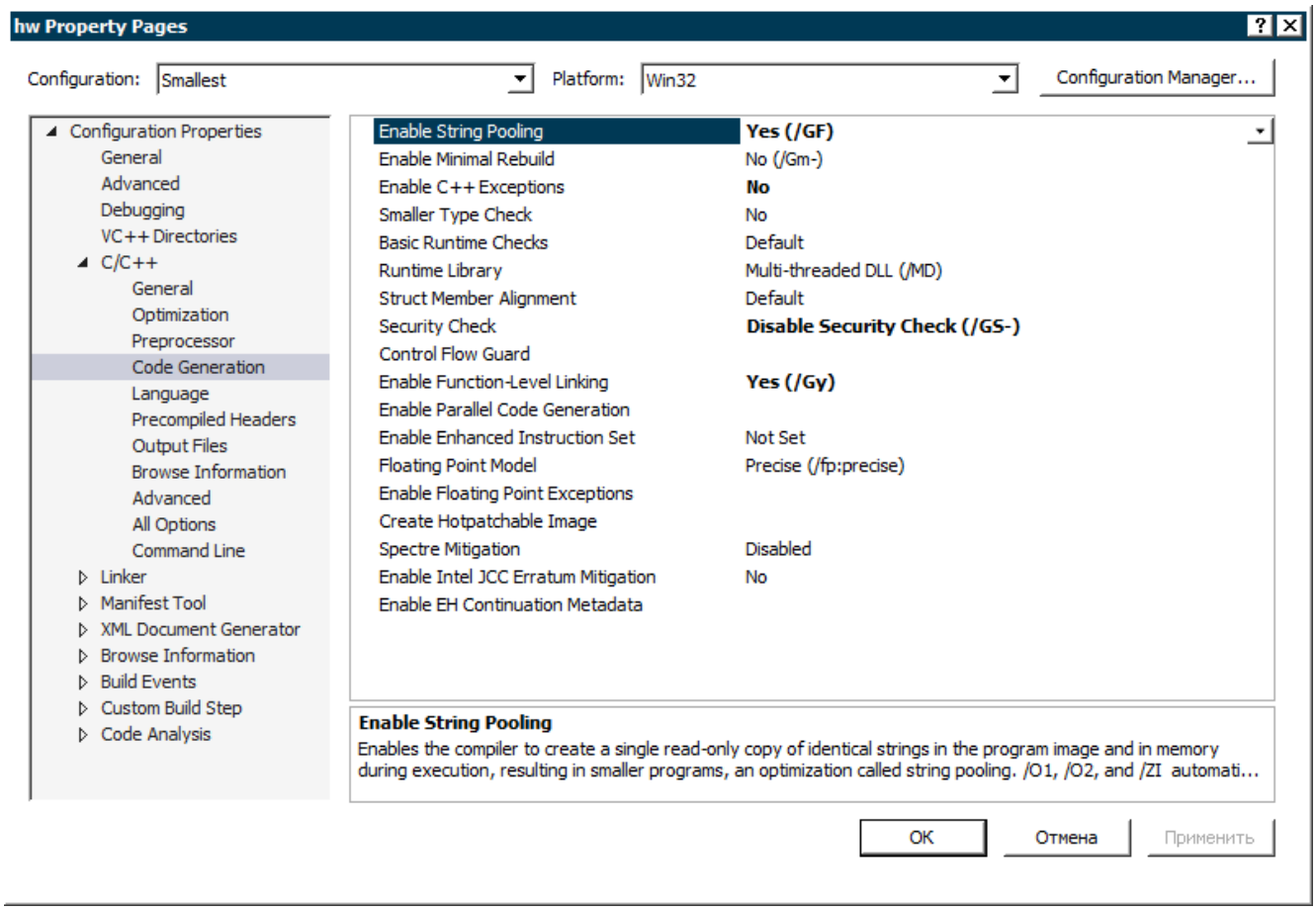


Рисунок 6 - Параметры генерации кода.

Ещё одним способом уменьшения размера исполняемого файла является отказ от библиотеки среды выполнения, которая обеспечивает поддержку стандартных функций языка Си. Этот подход требует изменения кода программы и ориентацию его на конкретное окружение. Например, для Windows следует использовать WinAPI. На Рисунке 7 представлен новый код программы.

```
void main(void)
{
    WriteConsoleA(
        GetStdHandle(STD_OUTPUT_HANDLE),
        "Hello World",
        11,
        NULL,
        NULL
    );
}
```

Рисунок 7 - Hello world на WinAPI.

Такой подход позволяет избавиться от поддержки стандартных библиотек.

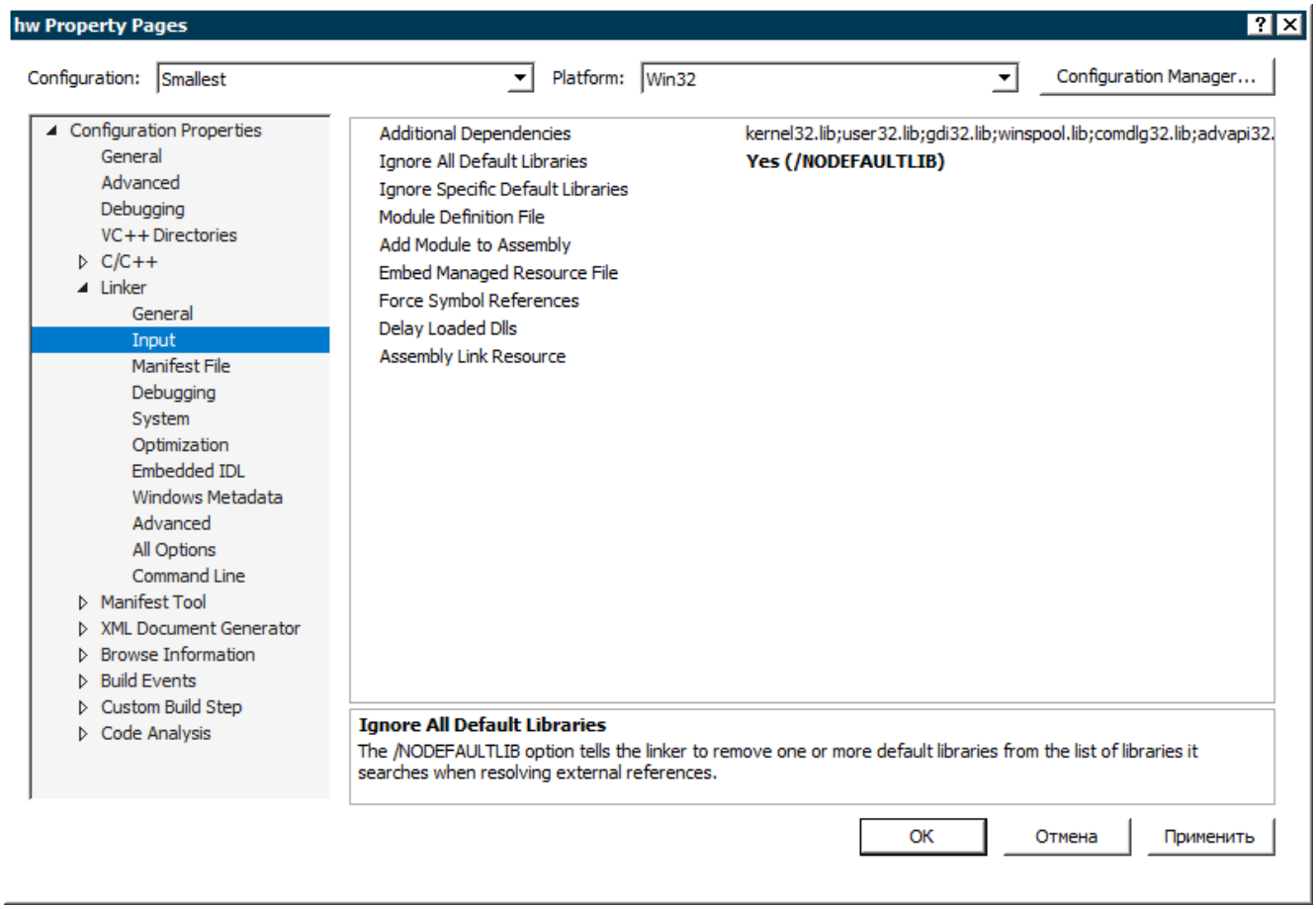


Рисунок 8 - Параметры ввода компоновщика.

Так же была отключена генерация файла манифеста, отладочной информации, указана точка входа.

После внесённых изменений размер исполняемого файла составляет 1 536 байт. То есть его размер уменьшился в шесть раз.

## 10. ВЫВОДЫ

В ходе выполнения курсовой работы было рассмотрено влияние различных методов оптимизации на время выполнения реальной программы.

Наибольший выигрыш по времени был получен при применении оптимизации на уровне алгоритма (около 400%). Результаты тестирования показали, что основным достоинством переработанного модуля обработки входного изображения является его масштабируемость, которая была достигнута распределением алгоритмов этого модуля. Именно его ориентированность на параллельные вычисления даёт наибольший прирост эффективности при больших объёмах входных данных. Благодаря переработке этого модуля время выполнения оптимизированной программы сократилось в несколько раз.

Основная часть оптимизации на машино-независимом уровне заключалась в понижении мощности операций, и, в частности, сведения большинства операций раскрашивания графа к битовым. Такие изменения привели к увеличению эффективности примерно на 90%.

Оптимизации на машино-зависимом уровне содержали в себе включение ассемблерных инструкций и настройку параметров генерации кода. Результатом этих оптимизаций является прирост эффективности оптимизации, который при разных входных данных принимает значения от нескольких десятков процентов до тысячи. Так же оптимизации на этом уровне позволили уменьшить размер исполняемого файла в три раза.

Так же оптимизации размера исполняемого файла подверглась программа «Hello world». После отказа от стандартных библиотек Си с последующим переходом на WinAPI и настройкой параметров генерации исполняемого файла удалось добиться уменьшения размера файла в шесть раз.