Итоговое практическое задание по OpenMP: параллельная реализация программы обработки изображений

Введение

В этом задании вам необходимо провести анализ приложения, обрабатывающего bmp изображения, и повысить скорость работы данного приложения. Перед началом выполнения данного задания мы настоятельно рекомендуем вам просмотреть все видео нашего курса, если вы еще этого не сделали, и пройти все тесты.

Для начала выполнения данного задания вам необходимо скачать архив со стартовым проектом и распаковать его в директорию, в которой вы будете выполнять практическое задание. Для выполнения данного задания вам понадобится Microcoft Visual Studio 2015 и Inter Parallel Studio 2016. Инструкцию по установке данных программных продуктов вы можете найти в вводном модуле нашего курса.

Рекомендации

Анализ программы с использованием Intel Amplifier необходимо проводить только используя конфигурацию Release и на изображениях большого размера (все исходные изображения, содержащие в своем имени слово large). Имя файла с изображением, которое считывается, задается в 9 строчке кода в файле main.cpp. При отладке вашего приложения в конфигурации Debug рекомендуется использовать маленькое изображение (test_image_small.bmp).

После изменений, которые вы сделали в коде, рекомендуется запустить программу и провести сравнение полученного изображения с эталонным изображением (output_image_small_ref.bmp для маленького изображения, аналогичные файлы есть для больших изображений).

Файлы входящие в задание

Папки:
ImageProcessing – основная папка всего решения;
Image – папка, содержащая исходные изображения и итоговые изображения, получающиеся после работы программы.

Файлы с исходным кодом:
EasyBMP.cpp – файл содержит реализацию класса BMP;
main.cpp – файл, содержащий основную функцию main;
gaussian_blur.cpp – файл, содержащий реализацию функции Гауссового размытия.

Заголовочные файлы:
EasyBMP.h – основной заголовочный файл, содержащий настройки и включающий другие заголовочные файлы по работе с изображениями в формате bmp;
EasyBMP_BMP.h – заголовочный файл, содержащий описание класса BMP;
EasyBMP_DataStructures.h – заголовочный файл, содержащий описание основных классов данных;
image_proc.h – заголовочный файл, содержащий описание основных функций обработки изображений.

1. **Первая компиляция приложения и создание исходной точки сравнения**

В этом задании вам необходимо провести компиляцию приложения и зафиксировать время выполнения программы и получаемый результат.

После того как вы откроете проект в Visual Studio, дважды нажав на файл ImageProcessing.sln или открыв его через меню «File-Open-Project/Solution..», вам необходимо выбрать release режим компиляции приложения в меню, показанном на рисунке 1.

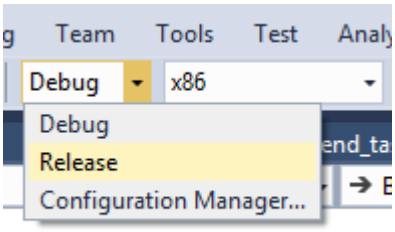

Рисунок 1 – Меню выбора настроек компиляции

После этого необходимо выбрать компилятор Intel. Это можно сделать, вызвав контекстное меню, нажав правой кнопкой на названии проекта «ImageProcessing» и выбрав компилятор Intel в меню «Intel Compiler», как это показано на рисунке 2.

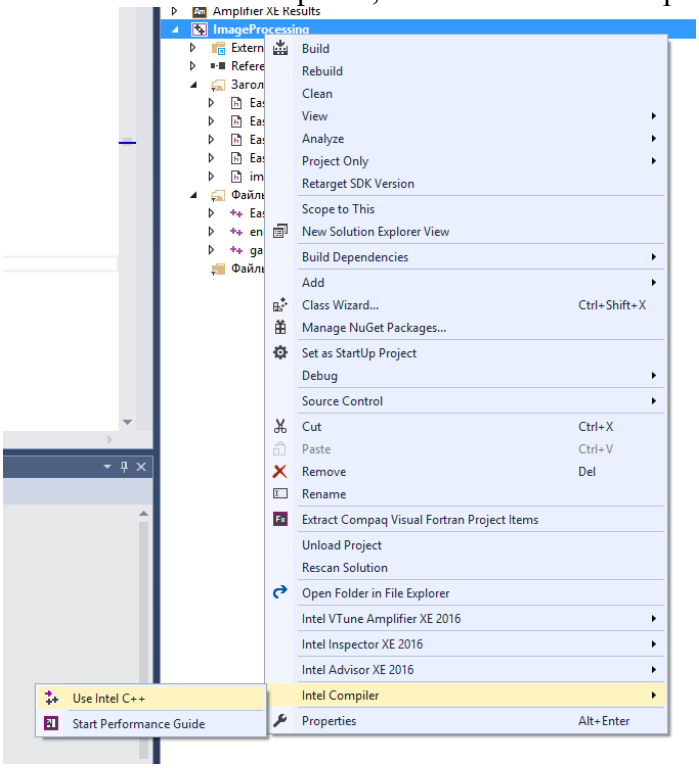

Рисунок 2 – меню выбора компилятора Intel

После этого необходимо пересобрать приложение и запустить его. Запомните или запишите где-то время выполнения программы. После того как вы все успешно выполните, необходимо раскомментировать строчку с вызовом функции Submit_Part1() в файле main.cpp, еще раз скомпилировать приложение и запустить его. В результате запуска у вас должен создаться файл submit1.bin в директории ImageProcessing, который вам и необходимо загрузить для проверки на платформе Coursera для первой части финального задания.

2. Оптимизация и векторизация первой hotspot функции FindClosestColor

В этой части задания вам будет необходимо провести оптимизацию и векторизацию функции, которая в приложении выполняется дольше всего.

Для того чтобы определить «горячие»(hotspot) функции, надо воспользоваться Intel Amplifier и провести basic hotspot analysis. Если вы все правильно сделали, то у вас при анализе должен получиться следующий результат (На вкладке Summary пункт Top Hotspots). Дольше всего выполняются пользовательские функции BMP::FindClosestColor, затем BMP::GetColor, IntSquare и Gaussian_blur.

Функция FindClosestColor выполняет поиск ближайшего цвета из доступной палитры к текущему цвету обрабатываемого пикселя, вычисляя расстояние между текущим цветом пикселя и цветом из палитры, для того чтобы сохранить изображение с заданной глубиной цвета. Из этого следует, что данная функция вызывается для каждого обрабатываемого пикселя изображения, т.е. очень часто. Изучив код данной функции (чтобы быстро перейти к коду, во вкладке Bottom-up дважды нажмите на строчку с функцией FindClosestColor и в открывшемся окне дважды левой кнопкой нажмите на любой строчке кода), вы увидите, что в ней поиск ближайшего цвета осуществляется в цикле while. Сначала замените цикл while на цикл for, так как в любом случае цикл выполняется заданное число раз. Цикл for предпочтительнее использовать при распараллеливании или векторизации.

Теперь внимательно посмотрите на цикл и на то, что в нем выполняется. В конце имеются такие строчки кода

```
if (BestMatch < 1)
{
        i = NumberOfColors;
}
```

В данном случае условие будет истинным только тогда, когда текущий цвет и цвет из палитры совпали, и дальнейший поиск будет прекращен. Но данная конструкция плохо влияет на выполнение команд в процессоре (это отдельная большая тема оптимизации приложений), а цвет совпадать будет очень редко, поэтому удалим данные строчки кода. Можете провести сравнение время выполнения программы до удаления этих строчек и после.

Далее замените вызов функции IntSquare (данная функция возводит число в квадрат) на прямое перемножение аргументов данной функции. Данное преобразование эквивалентно включению опции компилятора, разрешающей инлайнить функции.

После этого вы можете попытаться реализовать параллельное выполнение цикла for, но в данном цикле вычислений не сильно много, а функция вызывается часто, поэтому затраты на создание, синхронизацию потоков в конце параллельной секции и уничтожение потоков будут большие. Давайте попробуем векторизовать этот цикл. В том, что цикл не был векторизован, вы можете убедиться, включив опцию opt_report в свойствах проекта (рисунок 3)

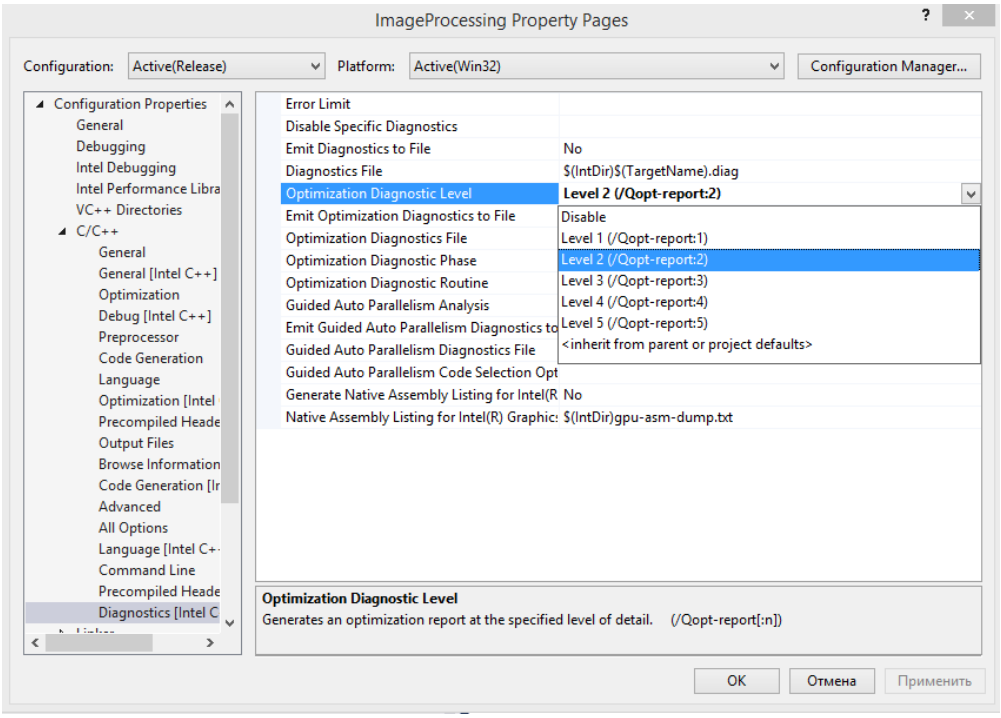

Рисунок 3 – включение опции выводящей диагностическую информацию компилятора

После этого перестройте проект, и вы увидите отчеты компилятора по оптимизации и векторизации прямо в исходных кодах. Обратимся к сообщению, которое нам выдал компилятор на цикл for в функции FindClosestColor. У вас должно появиться следующее сообщение «loop was not vectorized: loop with function call not considered an optimization candidate.». Компилятор не может векторизовать цикл, в котором производится вызов функции. В нашем случае вызывается функция GetColor, которая возвращает i-цвет из палитры.

Для того чтобы устранить данную проблему, необходимо либо сделать функцию GetColor векторной, либо вынести вызов функции за цикл. Пойдем по второму пути, так как в данном случае он немного проще. В классе BMP есть функция GetColors (тип возвращаемого значения RGBApixel*), которая возвращает указатель на массив, в котором хранится палитра для заданной глубины цвета. Впишите следующий код, который вызывает данную функцию, перед циклом for

```
RGBApixel* Colors = GetColors();
```

а затем замените использование переменной Attempt в цикле, в следующих строчках кода

```
int TempMatch = ((int)Attempt.Red - (int)input.Red) * ((int)Attempt.Red -
(int)input.Red)
          + ((int)Attempt.Green - (int)input.Green) * ((int)Attempt.Green -
(int)input.Green)
          + ((int)Attempt.Blue - (int)input.Blue) *((int)Attempt.Blue -
(int)input.Blue);
```

на использование массива `Colors` и удалите строчку внутри цикла, вызывающую функцию GetColor.

Теперь еще раз посмотрите на вывод диагностической информации компилятора по данному циклу (не забудьте пересобрать проект). Теперь компилятор вам должен сообщить, что он не может векторизовать цикл, так как имеются зависимости по данным, и подскажет, в каких строчках. Как видите, это из-за условного оператора в теле цикла. Для того чтобы избавиться от этой зависимости, вам необходимо завести массив с именем

TempMatchArr. Память под данный массив должна выделяться динамически. Не забудьте освободить память перед выходом из функции. Данный массив будет хранить значения расстояний, вычисленные для каждого i-цвета из палитры до цвета текущего пикселя. Затем разбейте основной в функции на два цикла. В первом будут проходить вычисления расстояния между цветами, во втором – поиск минимума расстояний и его номера. После этого соберите программу и запустите ее. Запомните или запишите время выполнения программы и сравните с тем временем, что было раньше, а также выполните еще раз hotspot анализ и сравните результаты с предыдущим анализом

Теперь пересоберите проект и обратитесь к диагностическому сообщению компилятора. Вы увидите, что, в принципе, компилятор может векторизовать первый цикл, но считает это неэффективным.

Заставьте компилятор векторизовать цикл воспользовавшись директивой omp simd. А затем сравните время выполнения до векторизации и после данного цикла. Удобнее сравнивать проведя hotspot анализ и сравнив анализ до использования директивы и после.

Как вы видите, векторизация только увеличила время выполнения. Неэффективность векторизации связана с тем, что данные, скажем, необходимые для выполнения первого векторного вычитания (Colors[0].Red, Colors[1].Red, Colors[2].Red6 Colors[3].Red),

```
            TempMatchArr[i]    =    ((int)Colors[i].Red    -    (int)input.Red)    *
((int)Colors[i].Red - (int)input.Red)
                  +      ((int)Colors[i].Green      -      (int)input.Green)      *
((int)Colors[i].Green - (int)input.Green)
                  + ((int)Colors[i].Blue - (int)input.Blue) *((int)Colors[i].Blue -
(int)input.Blue);
```

в памяти лежат с пропусками, так как в памяти элементы массива Colors записаны так, как это показано на рисунке 4. Поэтому при загрузке данных в векторные регистры они загружаются не все сразу, а по одному элементу, так как между ними имеются пропуски. В таких случаях необходимо изменить структуру хранения данных, в нашем случае надо перейти от массива структур к структуре массивов.

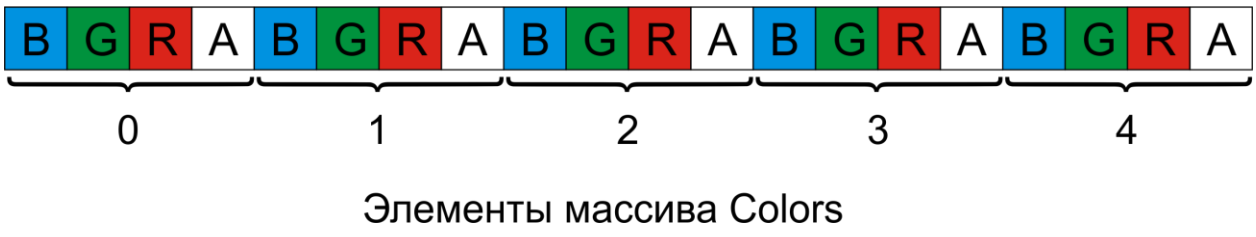

Рисунок 4 – расположение данных в памяти

В классе BMP уже имеется такой тип RGBApixelArr, а также метод GetColorsArr, возвращающий свойство класса BMP ColorsStructOfArray, которое хранит цвета палитры для заданной глубины цвета (данное свойство определяется в методе SetBitDepth). Обратите внимание, что в данном методе память под массивы структуры выделяется выровненной по значению 64 с помощью функции _mm_malloc.

Для того чтобы воспользоваться этим массивом, сначала необходимо заменить строчку кода

```
RGBApixel* Colors = GetColors();
```

на

```
RGBApixelArr Colors = GetColorsArr();
```

```
int* Blue = Colors.Blue;
int* Green = Colors.Green;
int* Red = Colors.Red;
int ired = input.Red;
int igreen = input.Green;
int iblue = input.Blue;
```

Соответственно цикл вычисления расстоянийидолжен у вас записать следующим образом

```
for (int i = 0; i < NumberOfColors; i++)
{
    TempMatchArr[i] = (Red[i] - ired)*(Red[i] - ired) + (Green[i] - igreen)*(Green[i] - igreen) + (Blue[i] - iblue)*(Blue[i] - iblue);
}
```

Если вы сейчас поставите максимальный уровень диагностических сообщений, как показано на рисунке 5, то после того как перестроите свой проект, увидите, что компилятор сообщает о том, что цикл был векторизован, но обращение к массивам происходит по невыровненному указателю. Для того чтобы цикл вычисления расстояний все время векторизовался, и подсказать компилятору, что данные выровнены, воспользуйтесь директивой simd и опцией данной директивы aligned. Также организуйте выделение памяти под массив `TempMatchArr` с использованием функции _mm_malloc и в конце освободите память с использованием функции _mm_free.

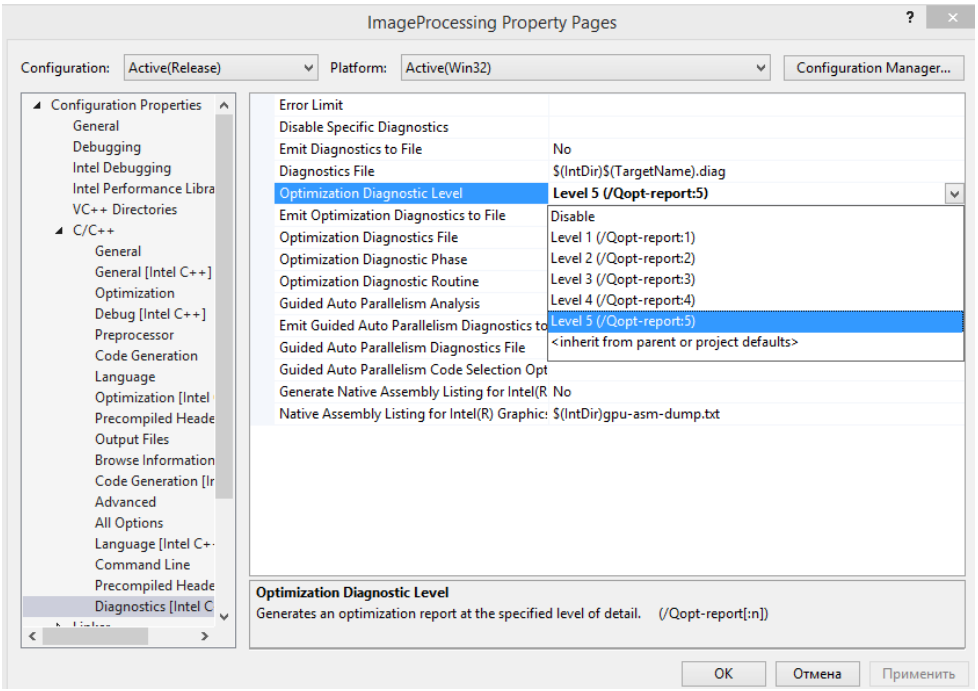

Рисунок 5

Также не забудьте добавить опцию компиляции /QxHost, которая позволит сгенерировать векторные инструкции под ваш тип процессора. Добавить ее необходимо во вкладке C/C++ - Command Line (Рисунок 6).

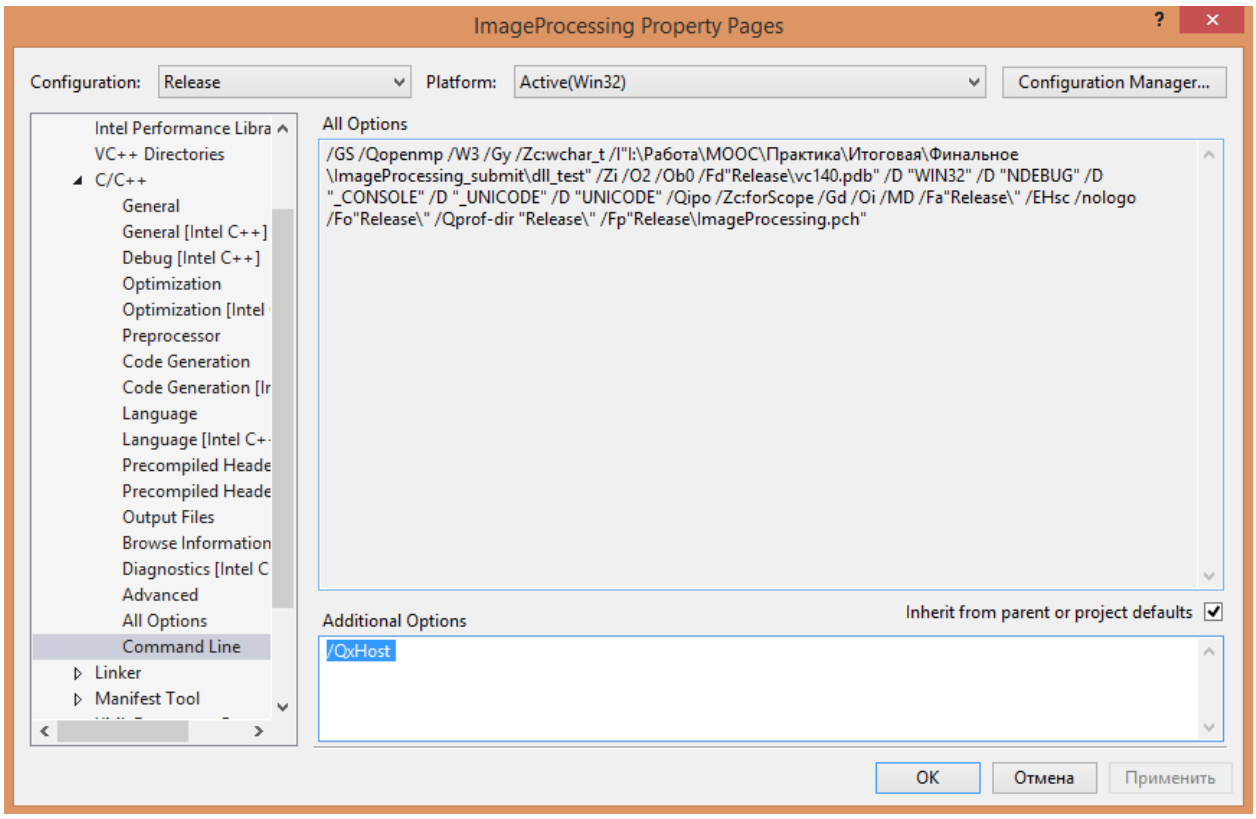

Рисунок 6

После этого соберите программу и запустите ее. Запомните или запишите время выполнения программы и сравните с тем временем, что было раньше. Также выполните еще раз hotspot анализ и сравните результаты с предыдущим анализом. При сравнении обращайте внимание на время выполнения функции FindClosestColor.

После всех преобразований проверьте правильность работы программы, сравнив рисунки output_image8bpp.bmp и output_image8bpp_ref.bmp. Если картинки совпадают, то для проверки задания закомментируйте Submit_Part1() и раскомментируйте вызов функции `Submit_Part2()` в файле end_task.cpp, после завершения программы в папке ImageProcessing будет создан файл submit2.bin, который надо загрузить на платформу Coursera для проверки во второй части задания.

3. Параллельная реализация функции записи Write8bitRow

В этой части задания вы должны еще сократить время выполнения программы за счет параллельности.

Мы достаточно сильно сократили время выполнения функции FindClosestColor, но саму эту функцию распараллелить эффективно очень сложно. Для уменьшения времени выполнения программы попытаемся распараллелить место, где данная функция используется, так как она вызывается для обработки каждого пикселя.

Откройте последний проведенный basic hotspot analysis или запустите новый. После этого перейдите во вкладку Bottom-Up. В моем случае данная функция на 4 позиции по времени выполнения (у вас будет отличаться время и, возможно, немного позиция этой функции). На рисунке представлены результаты при обработке файла test_image_large.bmp) (рисунок 7).

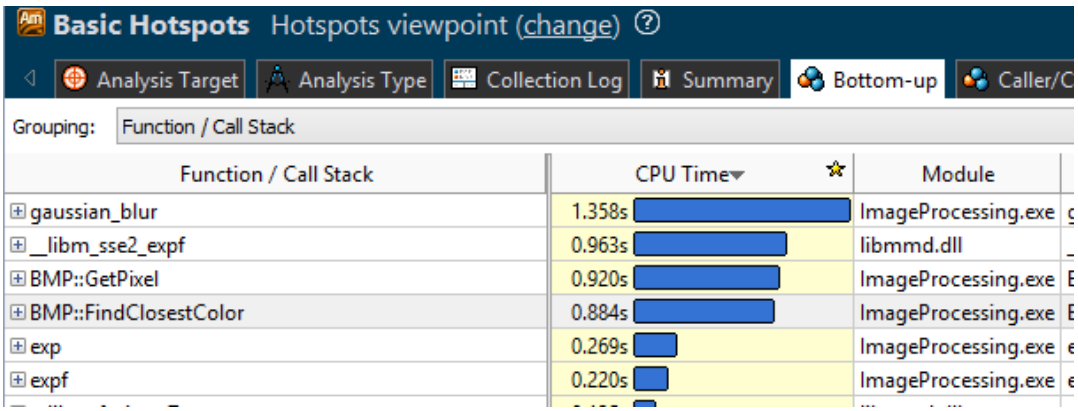

Рисунок 7 – Базовый анализ «горячих» функций.

Для более удобного анализа где данная функция вызывалась, перейдем во вкладку Top-down Tree (рисунок 8).

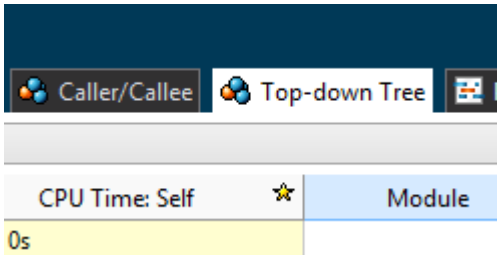

Рисунок 8

В данном представлении вы увидите, что функция FindClosestColor вызывается в функции Write8bitRow (рисунок 9).

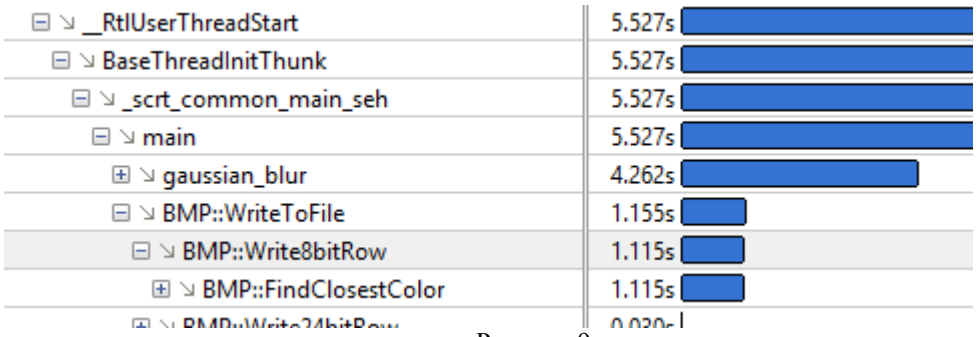

Рисунок 9

Дважды нажав на строчку с этой функцией, вы перейдете в окно исходного кода.

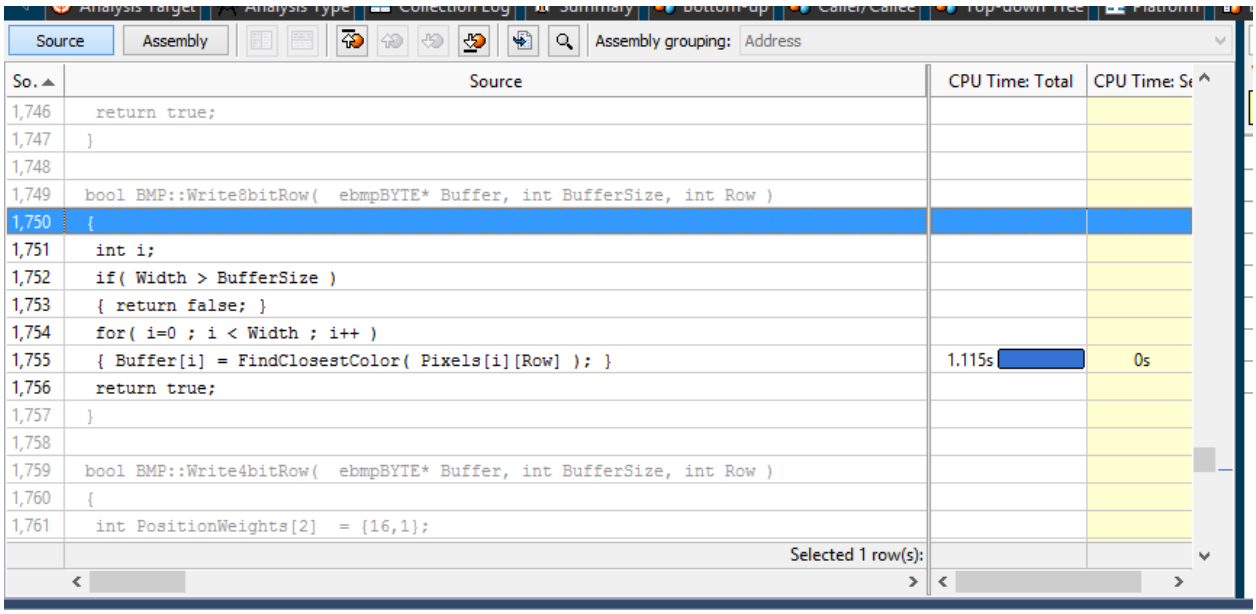

Рисунок 10

Здесь вы увидите, что данная строчка с вызовом этой функции занимает все время выполнения и вызывается в цикле. Данный цикл и необходимо распараллелить. Для распараллеливания воспользуйтесь OpenMP. Для перехода непосредственно в файл с исходным кодом дважды нажмите на строчку кода с вызовом функции FindCloseColor. Теперь вам необходимо провести распараллеливание данного цикла.

После всех преобразований проверьте правильность работы программы, сравнив рисунки output_image8bpp.bmp и output_image8bpp_ref.bmp. Если картинки совпадают, то для проверки задания закомментируйте Submit_Part1(), Submit_Part2() и раскомментируйте функцию `Submit_Part3();` в файле end_task.cpp, после завершения программы в папке ImageProcessing будет создан файл submit3.bin, который надо загрузить на платформу Coursera для проверки.

4. Параллельная реализация функции gaussian_blur

В этой части задания вы должны сократить время выполнения программы за счет параллельной реализации функции `gaussian_blur.` Данная функция проводит Гауссово размытие. Подробнее про него вы можете почитать информацию по следующей ссылке «Gaussian blur». Для успешного выполнения данного задания пользуйтесь всеми возможностями OpenMP по сбалансированному распределению работы.

После всех преобразований проверьте правильно работы программы, сравнив рисунки output_image8bpp.bmp и output_image8bpp_ref.bmp. Если картинки совпадают, то для проверки задания закомментируйте Submit_Part1(), Submit_Part2(), Submit_Part3() и раскомментируйте функцию `Submit_Part4();` в файле end_task.cpp, после завершения программы в папке ImageProcessing будет создан файл submit4.bin, который надо загрузить на платформу Coursera для проверки.

Оценка заданий

После выполнения всех частей убедитесь, что вы загрузили все файлы в систему проверки заданий и все задания у вас были засчитаны.
В таблице 1 представлены названия заданий и баллы за каждую часть.

Таблица 1

Название части	**Баллы**
Первая компиляция приложения и создание исходной точки сравнения	10
Оптимизация и векторизация первой hotspot функции FindClosestColor	30
Параллельная реализация функции записи Write8bitRow	30
Параллельная реализация функции gaussian_blur	30
Итого	100