Свертка

GPU Teaching Kit – Accelerated Computing

# Цели

Цель данной лабораторной – реализовать блочный алгоритм свертки изображения, используя как разделяемую, так и константную память. Маска свертки будет константной и иметь размер 5x5, но размер изображения будет произвольным (но большим, чем 5x5).

Для того чтобы использовать константную память для маски свертки, в первую очередь вы должны отправить ее на устройство. Допустим, массив, представляющий собой маску, на устройстве называется M. Вы можете использовать const float \* \_\_restrict\_\_ M как один из параметров вашего ядра. Это информирует компилятор, что содержимое массива маски константное и может быть доступно только через указатель M. Тогда компилятор разместит эти данные в константную память и позволит потоковому мультипроцессору агрессивно кэшировать содержимое маски во время выполнения ядра.

Свертка используется во многих областях, в том числе, в наложении фильтров на изображение. Стандартная формула сверки изображения I с фильтром M размером 5x5:

где – выходной пиксель на позиции i,j по каналу c, – входной пиксель на позиции i,j по каналу c (в нашем случае количество каналов будет равно 3, что соответствует значениям RGB) и – маска в точке x,y.

# Предварительные требования

Прежде чем начать, убедитесь, что:

* вы изучили все материалы Модуля 8

# Входные данные

Входные данные представлены перемежеванным изображением размером height x width x channels. Говоря “перемежеванное”, мы подразумеваем, что I[y][x] содержит три значения, соответствующие каналам RGB. Это означает, что, чтобы получить указатель на значение конкретного элемента, нужно вычислить что-то похожее на

index = (yIndex\*width + xIndex)\*channels + channelIndex;

В нашем случае индексы каналов соответствуют 0 для R, 1 для G, 2 для B. Таким образом, чтобы получить значение канала G элемента I[y][x], вам следует использовать линеаризованное выражение I[(yIndex\*width+xIndex)\*channels + 1].

Для простоты вы можете представлять channels как 3.

# Ход выполнения

Отредактируйте код, чтобы выполнить следующие действия:

* выделить память на устройстве;
* скопировать память хоста на устройство;
* инициализировать размерности блока нитей и ядра;
* запустить ядро CUDA;
* скопировать результат на хост;
* освободить память устройства;
* реализовать блочную двумерную свертку с учетом каналов;
* уменьшить количество обращений к глобальной памяти за счет использования разделяемой, обработать граничные условия при загрузке входных блоков в разделяемую память.

Места, куда следует вставить код, помечены комментариями //@@.

# Псевдокод

Последовательный псевдокод выглядит так:

maskWidth := 5  
maskRadius := maskWidth/2 # так как деление целочисленное, результат будет 2  
for i from 0 to height do  
 for j from 0 to width do  
 for k from 0 to channels  
 accum := 0  
 for y from -maskRadius to maskRadius do  
 for x from -maskRadius to maskRadius do  
 xOffset := j + x  
 yOffset := i + y  
 if xOffset >= 0 && xOffset < width &&  
 yOffset >= 0 && yOffset < height then  
 imagePixel := I[(yOffset \* width + xOffset) \* channels + k]  
 maskValue := K[(y+maskRadius)\*maskWidth+x+maskRadius]  
 accum += imagePixel \* maskValue  
 end  
 end  
 end  
 # значения пикселей находятся в диапазоне от 0 до 1  
 P[(i \* width + j)\*channels + k] = clamp(accum, 0, 1)  
 end  
 end  
end

где clamp определено как

def clamp(x, lower, upper)  
 return min(max(x, lower), upper)  
end

# Инструкция по установке

Последнюю версию исходного кода лабораторной, наряду со скриптами сборки, можно найти в репозитории [Bitbucket](https://bitbucket.org/hwuligans/gputeachingkit-labs/src/master/Module8). Инструкции к Cmake и сборки лабораторной можно найти в файле [README](https://bitbucket.org/hwuligans/gputeachingkit-labs/src/master/README.md) в корневой директории репозитория.

Исполняемый файл, являющийся результатом компиляции лабораторной, можно запустить следующей командой:

./Convolution\_Template -e <expected.ppm> \  
 -i <input0.ppm>,<input1.raw> -o <output.ppm> -t image,

где <expected.ppm> – имя ожидаемого результата, <input.ppm> – входной набор данных и <output.ppm> – необязательный путь для результатов. Набор данных можно сгенерировать, используя генератор, собранный во время компиляции.

Изображения хранятся в формате PPM (P6). Вы можете создать свои изображения, конечно же, если вы хотите. Простейший способ создать изображение – использовать сторонние средства, такие как bmptoppm. Маска хранится в формате CSV. Так как размер входных данных небольшой, лучшим вариантом будет редактирование их вручную.

# Вопросы

1. Назовите 3 применения свертки.
2. Сколько операций с плавающей точкой будет выполнено вашим ядром свертки? Объясните.
3. Сколько чтений из глобальной памяти произведет ваше ядро? Объясните.
4. Сколько записей в глобальную память произведет ваше ядро? Объясните.
5. Какими будут максимальное, минимальное и среднее значения количества *реальных* операций, которые произведет одна нить? Под *реальными* подразумеваются операции, которые напрямую влияют на финальный результат.
6. Какой была бы скорость выполнения операций с плавающей точкой для ядер CPU и GPU в данном приложении? Как они зависят от размера входных данных?
7. Сколько дополнительного времени понадобится в качестве накладных расходов на вычисления на GPU? Учтите, весь код выполняется на хосте, за исключением самого ядра. Как будут изменяться эти накладные расходы при увеличении размера входных данных?
8. Как вы думаете, что случится, если вы увеличите размер маски (скажем, до 1024), в то время как размер блока будет 16x16? На что вы потратите больше всего времени? Наложит ли это новые ограничения, которые нужно будет учесть в алгоритме (подумайте о размере разделяемой/константной памяти)?
9. Обязательно ли вам иметь отдельный буфер для результата? Или, с другой стороны, почему нельзя записывать результат свертки в то же самое место?
10. Что такое тождественная маска?

# Шаблон кода

Представленный код предлагается как отправная точка. Импорты, экспорты и проверка решения уже представлены в коде. Требуется вставить свой код в области, обозначенные //@@. Остальной код трогать не нужно. Руководство описывает функционал методов с префиксом wb\*.

#include <wb.h>  
  
#define wbCheck(stmt) \  
 do { \  
 cudaError\_t err = stmt; \  
 if (err != cudaSuccess) { \  
 wbLog(ERROR, "Failed to run stmt ", #stmt); \  
 return -1; \  
 } \  
 } while (0)  
  
#define Mask\_width 5  
#define Mask\_radius Mask\_width / 2  
#define TILE\_WIDTH 16  
#define w (TILE\_WIDTH + Mask\_width - 1)  
#define clamp(x) (min(max((x), 0.0), 1.0))  
  
//@@ ВСТАВЬТЕ СЮДА ВАШ КОД  
  
int main(int argc, char \*argv[]) {  
 wbArg\_t arg;  
 int maskRows;  
 int maskColumns;  
 int imageChannels;  
 int imageWidth;  
 int imageHeight;  
 char \*inputImageFile;  
 char \*inputMaskFile;  
 wbImage\_t inputImage;  
 wbImage\_t outputImage;  
 float \*hostInputImageData;  
 float \*hostOutputImageData;  
 float \*hostMaskData;  
 float \*deviceInputImageData;  
 float \*deviceOutputImageData;  
 float \*deviceMaskData;  
  
 arg = wbArg\_read(argc, argv); /\* чтение аргументов программы\*/  
  
 inputImageFile = wbArg\_getInputFile(arg, 0);  
 inputMaskFile = wbArg\_getInputFile(arg, 1);  
  
 inputImage = wbImport(inputImageFile);  
 hostMaskData = (float \*)wbImport(inputMaskFile, &maskRows, &maskColumns);  
  
 assert(maskRows == 5); /\* размер маски равен 5 в данной лабораторной \*/  
 assert(maskColumns == 5); /\* размер маски равен 5 в данной лабораторной \*/  
  
 imageWidth = wbImage\_getWidth(inputImage);  
 imageHeight = wbImage\_getHeight(inputImage);  
 imageChannels = wbImage\_getChannels(inputImage);  
  
 outputImage = wbImage\_new(imageWidth, imageHeight, imageChannels);  
  
 hostInputImageData = wbImage\_getData(inputImage);  
 hostOutputImageData = wbImage\_getData(outputImage);  
  
 wbTime\_start(GPU, "Doing GPU Computation (memory + compute)");  
  
 wbTime\_start(GPU, "Doing GPU memory allocation");  
//@@ ВСТАВЬТЕ СЮДА ВАШ КОД  
 wbTime\_stop(GPU, "Doing GPU memory allocation");  
  
 wbTime\_start(Copy, "Copying data to the GPU");  
//@@ ВСТАВЬТЕ СЮДА ВАШ КОД  
 wbTime\_stop(Copy, "Copying data to the GPU");  
  
 wbTime\_start(Compute, "Doing the computation on the GPU");  
//@@ ВСТАВЬТЕ СЮДА ВАШ КОД  
 convolution<<<dimGrid, dimBlock>>>(deviceInputImageData, deviceMaskData,  
 deviceOutputImageData, imageChannels,  
 imageWidth, imageHeight);  
 wbTime\_stop(Compute, "Doing the computation on the GPU");  
  
 wbTime\_start(Copy, "Copying data from the GPU");  
//@@ ВСТАВЬТЕ СЮДА ВАШ КОД  
 cudaMemcpy(hostOutputImageData, deviceOutputImageData,  
 imageWidth \* imageHeight \* imageChannels \* sizeof(float),  
 cudaMemcpyDeviceToHost);  
 wbTime\_stop(Copy, "Copying data from the GPU");  
  
 wbTime\_stop(GPU, "Doing GPU Computation (memory + compute)");  
  
 wbSolution(arg, outputImage);  
  
//@@ ВСТАВЬТЕ СЮДА ВАШ КОД  
  
 free(hostMaskData);  
 wbImage\_delete(outputImage);  
 wbImage\_delete(inputImage);  
  
 return 0;  
}