Здравствуйте.

Вашему вниманию предлагается дипломный проект «ПРОГРАММНЫЙ МОДУЛЬ ДЛЯ РЕАЛИЗАЦИИ ХААРОВСКОГО КЛАССИФИКАТОРА НА ПАРАЛЛЕЛЬНОЙ АРХИТЕКТУРЕ GPU».

Хааровский классификатор – набор признаков, которые используются для распознавания объектов на цифровом изображении. В рамках дипломного проекта производилось распознавание лиц. Примеры признаков, которые могут применяться, показаны на *плакате*. Некоторый набор признаков объединяется в каскад, который, в свою очередь, объединяются в классификатор.

Алгоритм распознавания сводится к следующему (*показывается на рисунке с названием «Каскад», плакат Хааровский классификатор, кроме пункта 1*):

* Для каждого признака каскада:
  + Выбирается определенный признак, который накладывается на каждый пиксель изображения (*рисунок «Используемая область»*).
  + Для области изображения, на которую накладывается данный признак, вычисляется итоговое значение признака, которое сравнивается с пороговым значением;
  + Данное пороговое значение определяет, какая из констант идет на суммирование
* Все значения, полученные от признаков, складываются между собой и сравниваются с эталонным значением, на основе которого определяется, прошел данный пиксель или нет.

Если пиксель не прошел данный каскад, то последующие каскады не рассматриваются, иначе берется следующий каскад (*рисунок «Классификатор»*). После того, как отработает весь классификатор остаются области, на которых предположительно найдено лицо.

Т.к. размеры признаков фиксированные, то дополнительно применяется масштабирование признаков или размера изображения, чтобы была возможность найти лицо, которое по размерам больше, чем используемый признак.

Результат работы программы представлен на рисунке «Пример работы программы».

Как видно из описания, алгоритм простой, но требует много вычислений, что не позволяет применять его для обработки видеопотока высокого разрешения. Поэтому была предпринята попытка переложить основную часть алгоритма на графический процессор. Это связано с тем, что они обладают пиковой производительностью, значительно превышающую современные процессоры (рисунок «Сравнение производительности центрального и графического процессоров»).

Для разработки была выбрана архитектура CUDA. Быстродействие GPU связано с его архитектурой (*рисунок Архитектура графического процессора*). Видеокарта содержит общую память (глобальную), контроллер ОЗУ, обеспечивающий доступ к глобальной памяти и множество мультипроцессоров, подключенных к общей шине. Каждый мультипроцессор состоит из множества вычислительных блоков (ВБ), которые подключены к общей разделяемой памятью и блоку регистров. Особенность ВБ – одновременно они могут исполнять одну и ту же команду.

Для программирования используются следующие понятия – решетка блоков, блоки и треды (*рисунок Модель программирования CUDA*). Тред соответствует одному вычислительному блоку, блок – мультипроцессору, решетка блоков – графическому процессору.

В проекте используется **3 типа памяти** – глобальная, разделяемая и константная. **Константная** располагается в глобальной ОЗУ устройства, но обладает собственным кэшем, что делает ее достаточно быстрой.

**Глобальная** память – самая медленная, но и самая большая память. Единственный способ ускорить – использование механизма транзакций. При этом должны использоваться следующие условия – обращения всех 16 тредов попадают в один 64 или 128 байтный сегмент, который выровнен в памяти; тип используемый при обращении должен занимать 32, 64 или 128 бит, k-ый поток должен читать соответствующее ему k-ое слово из сегмента данных в памяти.

**Разделяемая** память одна из самых быстрых. Основной способ для передачи данных между тредами. Вся память разбивается на банки одинакового размера. Для достижения максимальной производительности требуется, чтобы 16 тредов обращались к различным банкам памяти (*рисунок Примеры обращений к разделяемой памяти*).

Учитывая все вышесказанное, простейший вариант, когда каждому пикселю ставится в соответствие свой тред, не является эффективным, т.к. число каскадов, которые тред должен обрабатывать, значительно отличаются и нагрузка на ВБ становится не равномерной.

Поэтому была введены следующие условия:

* Все изображение разбивается на участки по 32х32 либо 16х16 пикселей в зависимости от обрабатываемого каскада;
* Обрабатываются не все каскады классификатора, а лишь один, за исключением первых двух каскадов;
* Перед тем, как обрабатывается очередной каскад, применяется планировщик

Тогда упрощенный вид программы представлен на рис. «*Упрощенная схема программы*»

**Инициализация** исполняется только для первого изображения и определяет видеокарту, с которой будет работать программа, инициализирует память для работы.

**Обработка каскада** разделяется на 2 основных типа – универсальный и на основе генератора кода. Их задача – исполнение очередного каскада классификатора. Каждый из этих подходов включает в себя 4 типа ядер – на 1, 2, 4 и 8 тредов на один пиксель. Такое разбиение связано с тем, что данные для обработки очень быстро сокращаются (*график Зачем использовать 4 варианта функций*) и нагрузка на вычислительные блоки становится не равномерной. Генератор кода применяется из-за того, что ядра на его основе обладают максимальной производительностью (*таблица Сравнение производительности вариантов обработки*). Однако из-за максимальной производительности теряется универсальность кода, т.к. все значения, которые заданны в параметрах классификатора, вставляются в текст программы. Поэтому дополнительно требуется перед использованием другого классификатора перекомпиляция программы.

**Планировщик**. Так как каждый каскад отсеивает около 50% всех пикселей изображения, то данные становятся разреженными. Соответственно, оптимальным образом нагрузить графический процессор становится невозможным. Задача планировщика заключается в том, чтобы составить специальный массив-план, элементы которого – индексы в маске изображения, которые требуется обрабатывать

После того, как отработает планировщик, осуществляется переход либо к обработке следующего каскада классификатора, либо к обработке оставшихся каскадов на CPU. Необходимость данной стадии объясняется с помощью рисунка «Почему переходим на CPU».

**Анализ результата** завершает обработку.

Тестирование проводилось с помощью Intel Core i7, NVIDIA GTX285.

В результате были получены следующие результаты (*таблица Суммарная производительность*):

* Время работы OpenCV – 220 мс;
* Время исполнения на GPU – 82 мс.

Выигрыш в скорости составил более чем в 2,5 раза.

/\* функция, которая ничего не возвращает \*/

void function(){ printf( "Hello wrold" ); } /\* обьявление функции \*/

void ( \*pointer )( void ); /\* обьявление указателя \*/

int main( int argc, char \*argv[] ){

pointer = &function; /\* берем адрес функции \*/

pointer(); /\* вызов функции function() через ее адрес \*/

return 0;

}

/\* функция, которая что-то возвращает \*/

char\* function(){ /\* обьявление функции \*/

static char \*str = "HelloWorld";

return str;

}

char\*( \*pointer )( void ); /\* обьявление указателя \*/

int main( int argc, char \*argv[] ){

pointer = &function; /\* берем адрес функции \*/

printf( "%s\n", pointer() ); /\* вызов функции function() через ее адрес \*/

return 0;

}

/\* функция, которая что-то принимает в качестве аргумента \*/

void function(const char \*\_s){ /\* обьявление функции \*/

printf( "%s\n", \_s );

}

void ( \*pointer )( const char \*); /\* обьявление указателя \*/

int main( int argc, char \*argv[] ){

pointer = &function; /\* берем адрес функции \*/

pointer( "Hello world" ); /\* вызов функции function() через ее адрес \*/

return 0;

}