

Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Пояснительная записка к курсовому проекту по дисциплине  
«Операционные системы и сети»

Тема: Распараллеливание задачи обработки изображений

Выполнил:  
Студент гр. 753504  
Краснов И. А.

Проверила:  
Ассистент кафедры  
Шнейдер В. В.

Минск 2020

## СОДЕРЖАНИЕ

СОДЕРЖАНИЕ .....	2
ВВЕДЕНИЕ.....	3
ЗАДАНИЕ .....	4
РАЗРАБОТКА СТРУКТУРЫ ПРОГРАММЫ .....	5
ФОРМАТ ИЗОБРАЖЕНИЯ.....	7
СВЕДЕНИЯ О ПРОГРАММИРОВАНИИ НА CUDA .....	8
МОДЕЛЬ ПАМЯТИ CUDA .....	10
ОСОБЕННОСТИ РЕШЕНИЯ ЗАДАЧИ НА CUDA .....	12
РАСЧЕТ КОНФИГУРАЦИИ.....	14
РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ.....	15
ЗАКЛЮЧЕНИЕ .....	24
СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ.....	25
ПРИЛОЖЕНИЕ А. ИНТЕРФЕЙС ПРОГРАММЫ.....	26

## ВВЕДЕНИЕ

В современном мире использование графического изображения возрастает с каждым днем. Ведь большинство людей легче воспринимают какую-либо информацию визуально, нежели на слуху. При этом наряду со значительным повышением уровня развития техники, весьма существенную роль играют методы обработки изображения.

Приблизительно на протяжении двухсот лет протекает процесс улучшения фотографий и не останавливается, по сей день. При переходе от аналоговых к цифровым методам обработки изображений упростились алгоритмы для этого. С каждым днем рождаются новые формы обработки изображения, некоторые совершенствуются, другим на смену приходят новее и интереснее. Благодаря современным технологиям, методы обработки изображения полностью меняют представление об изображении. Тем не менее, на сегодняшний день нет почти ни одной области, которую не затрагивала бы цифровая обработка изображений. Области применения цифровой обработки изображений достаточно разнообразны. Свое распространение оно имеет во многих науках, в создании рекламы и т.д.

Если мы вернемся на несколько лет назад, то вспомним, что фотографии, того времени в основном были черно-белыми, только постепенно начали приобретать цвет. Но с годами, они утрачивают свое прежнее состояние, теряют свое качество. С совершенствованием технологий удалось добиться того, что вернуть фотографию в первоначальный вид дело нескольких минут.

Обработка изображений обеспечивает улучшение изображений, сжатие данных для хранения и передачи по каналам связи, а также анализ, распознавание и интерпретацию зрительных образов.

В связи с распространением использования графики в компьютерной среде необходимы были и специализированная аппаратура для ускорения обработки графики. Результатом было создание GPU – процессора, который специализировался на обработке графических данных и вычислений с плавающей точкой. Особенностью таких процессоров является наличие большого числа процессоров, которые параллельно выполняют одни и те же действия (SIMD).

Основным содержанием данной работы будет знакомство с матричными фильтрами для обработки изображений. Результатом будет программа, которая позволяет накладывать матричный фильтр размером 5 на 5 и меньше на изображение.

## ЗАДАНИЕ

Разработать программу, которая будет накладывать фильтр с заданной матрицы (до 5 на 5) на изображение. Отфильтрованное изображение сохраняется в новый файл. Должна быть возможность сравнения обработки на GPU с CPU.

## РАЗРАБОТКА СТРУКТУРЫ ПРОГРАММЫ

Для реализации поставленной задачи необходимо программа на C/C++, компилируемая под CUDA. Графический интерфейс программы будет написан на C# с использованием технологии WPF.

Отдельно должны быть реализованы функции для обработки матриц 5 на 5 и 3 на 3, так как, несмотря на то, что при занулении крайних строк в матрице 5 на 5 мы получим матрицу 3 на 3 и тот же результат, обработка матрицы меньшего размера осуществляется значительно быстрее и фильтры с матрицей 3 на 3 встречаются чаще.

Узкое место обработки данных на GPU – работа с памятью и с необходимостью копирования из оперативной памяти на память устройства. Это можно увидеть на примере умножения матриц (см. рис. 1).

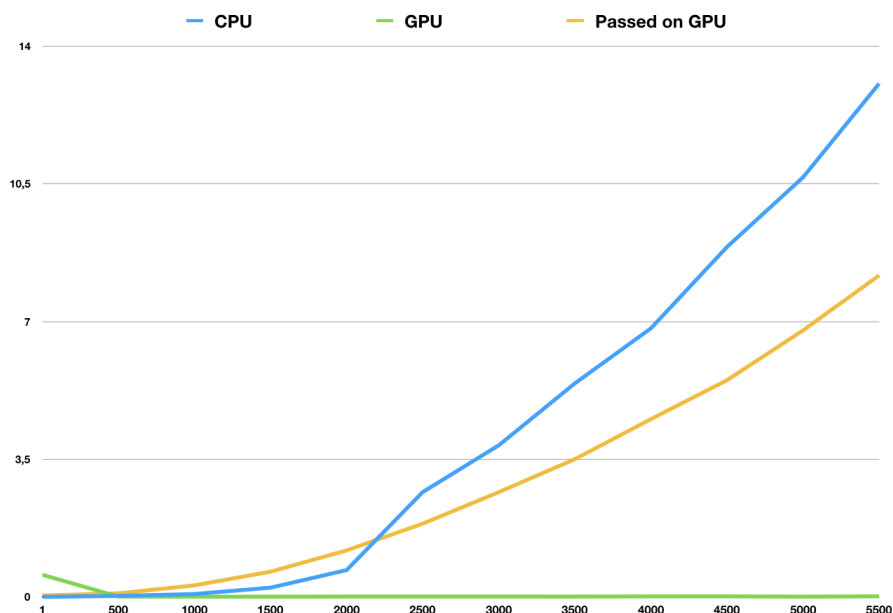


Рис. 1. График сравнения умножения матриц на CPU и GPU.

Линия оранжевого цвета – время, которое требуется для создания данных в обычном ОЗУ, передачу их в память GPU и последующие вычисления. Зеленая линия – время, которое требуется на вычисление данных, которые были сгенерированы уже в памяти видеокарты (без передачи из ОЗУ). Синяя – время подсчета на центральном процессоре.

Программа на C++ будет состоять из кода для GPU (описывает необходимые вычисления и работу с памятью устройства) и кода для CPU (управляет памятью GPU – выделение/освобождение, обменивает данные между GPU/CPU, запускает код на GPU, обрабатывает результаты и другой последовательный код). GPU рассматривается как периферийное устройство,

управляемое центральным процессором, т.е. оно не может само себя загрузить работой.

В итоге программа будет представлять из себя следующую структуру:

– на CPU:

- 1) переслать данные с CPU на GPU;
- 2) запустить вычисления на нитях GPU;
- 3) скопировать результат с GPU на CPU.

– на GPU:

- 1) код для нити.

Программа на C# будет подключать код на C++ в виде библиотеки dll и будет собирать данные для их асинхронного выполнения и проверять данные на корректность.

## ФОРМАТ ИЗОБРАЖЕНИЯ

Для обработки были выбран графический формат PPM с глубиной цвета 256 (данная глубина наиболее распространена и помещается в 1 байт) на 3 цвета в формате RGB. Формат PPM прост для чтения и записи, но при этом не использует сжатия данных и поэтому объемный.

Спецификация формата:

- 1) «магическое число» P6 для идентификации типа файла;
- 2) пробельные символы;
- 3) ширина, как десятичное число из ASCII символов;
- 4) пробельный символ;
- 5) высота, как десятичное число из ASCII символов;
- 6) пробельный символ;
- 7) максимальная глубина цвета, как десятичное число из ASCII символов (в нашем случае больше 0 и меньше 256);
- 8) один пробельный символ;
- 9) растровое изображение как строки общим числом равным высоте изображения. Каждый ряд содержит пиксели, общим числом равным ширине изображения. Каждый пиксель – 3 байта представляющих красный, синий, зеленый цвета (именно в таком порядке);
- 10) после «магического числа» до ширины изображения могут быть строки, начинающиеся с символа '#' и состоящие из ASCII символов, содержащие комментарии к изображению.

## СВЕДЕНИЯ О ПРОГРАММИРОВАНИИ НА CUDA

В упрощенном виде вычислительный блок графического процессора представляет собой совокупность потоковых мультипроцессоров (streaming multi-processor, SMX), управляемые с помощью GigaThread Engine. Каждый SMX состоит из множества вычислительных CUDA-ядер (рис. 2).

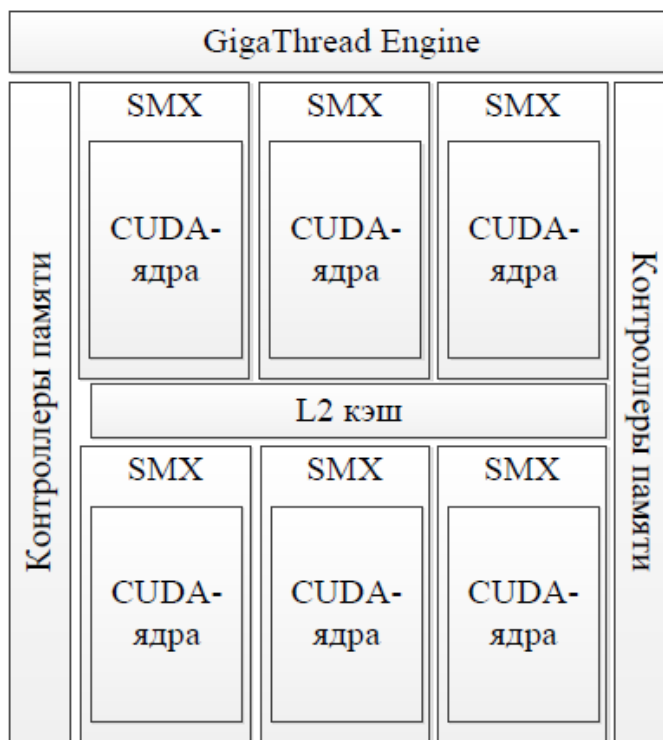


Рис. 2. Упрощенная структура графического процессора.

Такая архитектура легла в основу базовых понятий в CUDA – *нити* (*thread*), *блока* (*block*) и *решетки блоков* (*grid*).

*Нить* (*thread*) – базовая абстракция, представляющая собой легковесный поток, отвечающий за исполнение инструкций. Именно нить соответствует одному исполняющему ядру CUDA мультипроцессора. Максимальное число нитей, которое можно определить, – 2048 (для архитектуры NVIDIA© Kepler). Легковесность нити заключается в том, что время, необходимое на его создание, разрушение и переключение между нитями, ничтожно мало. Для упрощения аппаратной схемы каждые 32 CUDA ядра соединены с одним счетчиком команд, поэтому данные блоки ядер всегда выполняют одну и ту же инструкцию. Данные 32 нити называются *варпом* (*warp*).

Множество нитей объединяется в *блок* (*block*), который проецируется на мультипроцессор. Это означает, что все нити блока всегда будут выполняться на выделенном для него мультипроцессоре.



*Решетка блоков (grid)* представляет собой самый высокий уровень абстракции и, как правило, представляет собой всю задачу, которую требуется решить.

Каждая из абстракций является 3-мерной. Для ее определения в NVIDIA® CUDA определена специальная структура `dim3`.

Базовая задача, которую программист должен решить – правильное разбиение решаемой задачи между нитями и блоками для обеспечения эффективной нагрузки всех имеющихся вычислительных ресурсов.

*Ядро (kernel)* – функция, описывающая последовательность операций, выполняемых каждой нитью параллельно.

Для определения устройства исполнения функции (ядра) в CUDA вводятся следующие ключевые слова:

- `__host__` – функция может быть вызвана и выполнена только на стороне хоста (на центральном процессоре). Если не указывается спецификатор, то функция считается объявленной как `__host__`.
- `__global__` – ядро, предназначенное для выполнения на устройстве (графическом процессоре) и может быть вызвано с хоста. На устройствах с Compute Capability 3.5 (способ описания версии архитектуры графического процессора и поддерживаемого CUDA API) и новее допускается вызов с устройства. Ядра данного типа не возвращают никаких данных (тип `void`);
- `__device__` – ядро выполняется на устройстве, вызывается из кода, выполняемого на устройстве.

Все запуски ядер CUDA являются асинхронными: CPU запрашивает разрешение на запуск ядра путем записи в специальный буфер команд без проверки выполнен код или нет, после этого продолжает выполнять код в соответствии с алгоритмом для хоста.

## МОДЕЛЬ ПАМЯТИ CUDA

Технология CUDA использует следующие типы памяти: регистры, локальная, глобальная, разделяемая, константная и текстурная память (рис. 3). Однако разработчику недоступны к управлению регистры и локальная память. Все остальные типы он вправе использовать по своему усмотрению.

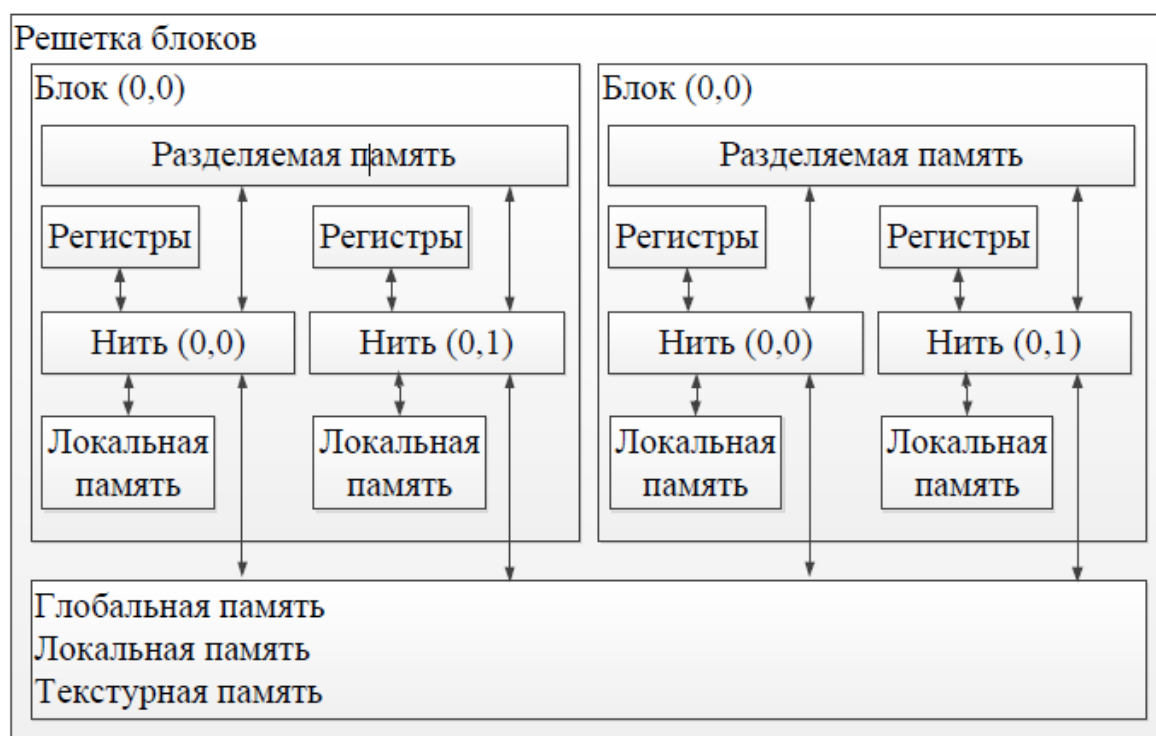


Рис. 3. Иерархия памяти CUDA.

*Глобальная память (global memory)* – тип памяти с самой высокой латентностью, из доступных на GPU. Переменные в данном типе памяти можно выделить с помощью спецификатора `__global__`, а так же динамически, с помощью функций из семейства `cudaMalloc`. Глобальная память в основном служит для хранения больших объемов данных, над которыми осуществляется обработка, и для сохранения результата работы. Данные перемещения осуществляются с использованием функций `cudaMemcpy`. В алгоритмах, требующих высокой производительности, количество операций с глобальной памятью необходимо свести к минимуму.

При копировании из глобальной памяти стоит помнить, что сразу отдается по 4 байта для нити варпа (32 нити), т.е. максимальная пропускная способность  $32 \cdot 4 = 128$  байт. При уменьшении числа требуемых байтов одной нитью, возникновении конфликтов при обращении к банкам памяти пропускная способность уменьшается.

*Разделяемая память (shared memory)* относится к типу памяти с низкой латентностью. Данный тип памяти рекомендуется использовать для минимизации обращения к глобальной памяти. Адресация разделяемой памяти осуществляется между нитями одного блока, что может быть использовано для обмена данными между потоками в пределах одного блока. Для размещения данных в разделяемой памяти используется спецификатор `__shared__`.

*Константная память (constant memory)* является одной из самых быстрых из доступных на GPU. Отличительной особенностью данного типа памяти является возможность записи данных с хоста, но при этом в пределах GPU возможно лишь чтение из этой памяти, что и обуславливает её название.

Для размещения данных в константной памяти предусмотрен спецификатор `__constant__`. Если необходимо использовать массив в константной памяти, то его размер необходимо указать на этапе компиляции, так как динамическое выделение в отличие от глобальной памяти в константной не поддерживается. Для записи с хоста в константную память используется функция `cudaMemcpyToSymbol`.

*Текстурная память (texture memory)* входит в состав текстурных блоков, используемых в графических задачах для формирования текстур. В текстурном блоке аппаратно реализована фильтрация текстурных координат, интерполяция, нормализация текстурных координат в случаях, когда они выходят за допустимые пределы.

*Регистровая память (register)* является самой быстрой из всех типов памяти. Определить общее количество регистров, доступных GPU, можно с помощью функции `cudaGetDeviceProperties`.

Все регистры GPU 32 разрядные. При этом в технологии CUDA нет явных способов использования регистровой памяти, всю работу по размещению данных в регистрах берет на себя компилятор.

*Локальная память (local memory)* может быть использована компилятором, если все локальные переменные не могут быть размещены в регистровой памяти. По скоростным характеристикам локальная память значительно медленнее, чем регистровая.

## ОСОБЕННОСТИ РЕШЕНИЯ ЗАДАЧИ НА CUDA

Обработка графической информации является одним из тех направлений, для которых технология CUDA может обеспечить существенное ускорение.

Реализация алгоритмов в области обработки изображений связана с решением ряда существенных проблем, актуальных для текущих архитектур графического процессора:

- эффективное распределение нагрузки между мультипроцессорами и вычислительными ядрами;
- формирование транзакций при чтении и записи данных из/в глобальную память (подраздел 2.1);
- специфические особенности конкретных алгоритмов (например, проблема обработки краев при фильтрации изображения);

Одна из проблем, с которой сталкивается разработчик при реализации любого алгоритма обработки изображений, - формирование транзакций при доступе к глобальной памяти. При этом можно выделить две случая, которые могут привести к нарушению шаблона доступа:

- один пиксель представляет собой 3 байта данных (каналы RGB);
- отсутствие выравнивания в памяти для второй и последующих строк исходного изображения.

При чтении данных в ядре из глобальной памяти разработчик должен решить проблему эффективной загрузки в разделяемую память. Проблема заключается в том, что пиксель для большинства алгоритмов представляет собой 3-байтную последовательность данных. Простейший способ загрузки – побайтное чтение данных – приводит к нарушению шаблона доступа к памяти и в результате транзакции не формируются, т.к. требуется последовательное обращение нитей к 4-байтным данным. На современных архитектурах за счет наличия кэша частично данная проблема сглаживается, однако это все равно приводит к существенному замедлению работы алгоритма.

Эффективное решение данной проблемы сводится к одному из двух решений. Первое решение заключается в добавлении канала альфа к каждому пикселю изображения, однако, с одной стороны, это приводит к увеличению требуемой глобальной памяти, а с другой стороны – требуется реализация алгоритма конверсии из 3 байтного формата в 4 байтный и обратно, что увеличивает полное время работы программы.

Второй способ эффективного решения проблемы заключается в том, чтобы одна нить загружала не один пиксель, а сразу 4 (рис. 4). Как видно из рисунка 4, для этого потребуется выполнить 3 загрузки данных типа `int`.

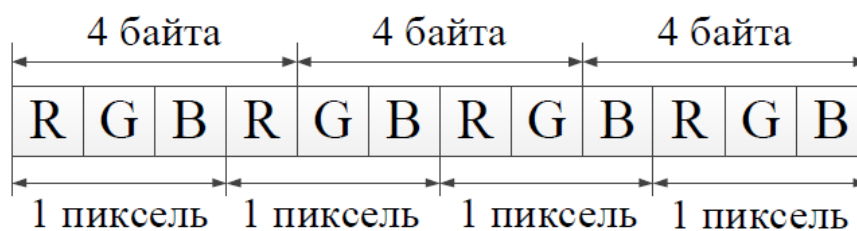


Рис. 4. Демонстрация загрузки и сохранения данных.

Проблема с негарантированным формированием транзакций при чтении второй и последующих строк заключается в том, что данные строки должны быть выравнены на границу 128 байт. В общем случае при произвольной ширине изображения данное условие гарантируется только для первой строки, т.к. функция `cudaMalloc` осуществляет выравнивание стартового адреса.

Решением данной проблемы является искусственное увеличение ширины изображения до границы 128 байт при инициализации памяти устройства (рис. 5). В рамках технологии CUDA предлагается 2 способа решения:

- ручное выравнивание и расширение изображения;
- использование CUDA API.

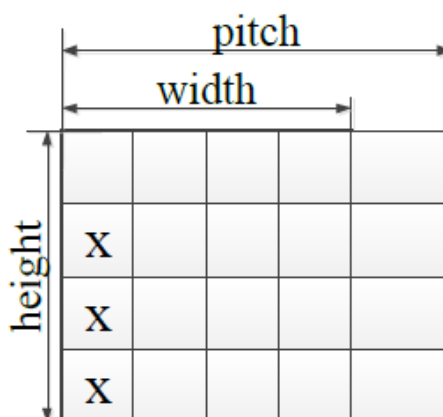


Рис. 5. Выравнивание строк изображения.

Был выбран второй вариант с использованием функции `cudaMallocPitch`. Она выделяет минимум  $\text{width} \times \text{height}$  байт памяти при этом в переменной `pitch` будет возвращено фактически выделенная ширина изображения с учетом выравнивания.

Для решения проблемы границ был выбран способ расширения изображения до размера  $(\text{width} + 2 * \text{kernelSize} / 2, \text{height} + 2 * \text{kernelSize} / 2)$ . Края заполнялись крайними пикселями изображения. При этом все пиксели исходной картинки становятся центральными. Минус – данный способ требует выделения дополнительной памяти для промежуточной картинки.

## РАСЧЕТ КОНФИГУРАЦИИ

Формирование конфигурации исполнения ядра является важной задачей, т.к. даже идеально оптимизированное ядро при неправильной конфигурации способно существенно замедлить работу.

Исходя из практики разработки ядер для обработки цветных изображений, наиболее оптимальным числом нитей по оси X изображения для большинства задач является 32. Данное число нитей позволяет использовать свойства варпа для синхронизации нитей. Обработываемое число пикселей в этом случае – 128, что, позволяет обрабатывать сравнительно небольшие по ширине изображения. Ширину по Y стоит выбирать из соображений, что пиковая нагрузка будет при 256-512 нитей на блок.

При расчетах использовался CUDA Occupancy calculator со следующими параметрами:

### CUDA Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

**1.) Select Compute Capability (click):** 5,0 [\(Help\)](#)  
**1.b) Select Shared Memory Size Config (bytes)**

**2.) Enter your resource usage:** [\(Help\)](#)  
 Threads Per Block 256  
 Registers Per Thread 64  
 Shared Memory Per Block (bytes) 7872

Рис. 6. Параметры для калькулятора.

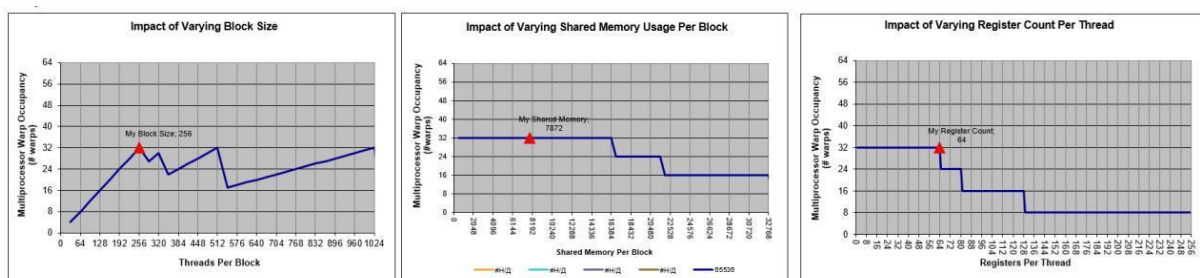


Рис. 7. Графики расчетов.

## РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ

Оригинальное изображение:

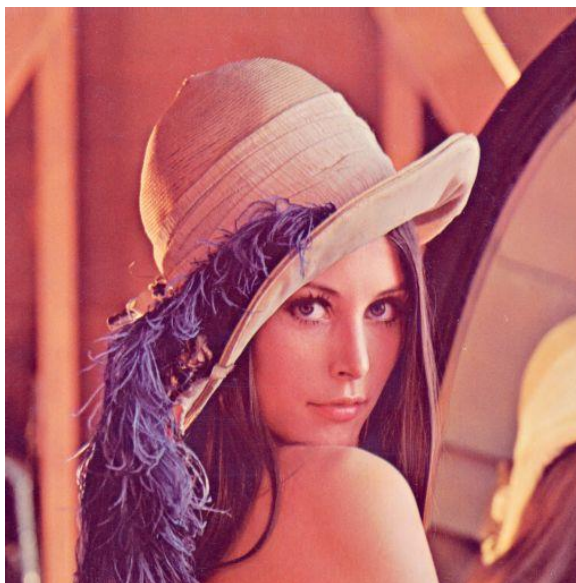


Рис. 8. Оригинальное изображение.



Рис. 9. Оригинальное изображение (черно-белый вариант для тиснения).

Низкочастотные фильтры применяются для сглаживания изображения, уменьшения шума в изображениях.

Фильтр 1:

$$\frac{1}{9} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$





Рис. 10. Результат применения фильтра 1.

Фильтр 2:

$$\frac{1}{10} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$



Рис. 11. Результат применения фильтра 2.

Фильтр 3:



$$\frac{1}{16} \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$



Рис. 12. Результат применения фильтра 3.

Фильтр 4:

$$\frac{1}{4} \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$



Рис. 13. Результат применения фильтра 4.

Высокочастотные фильтры усиливают контрастность, добавляют шумы в изображение.

Фильтр 5:

$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

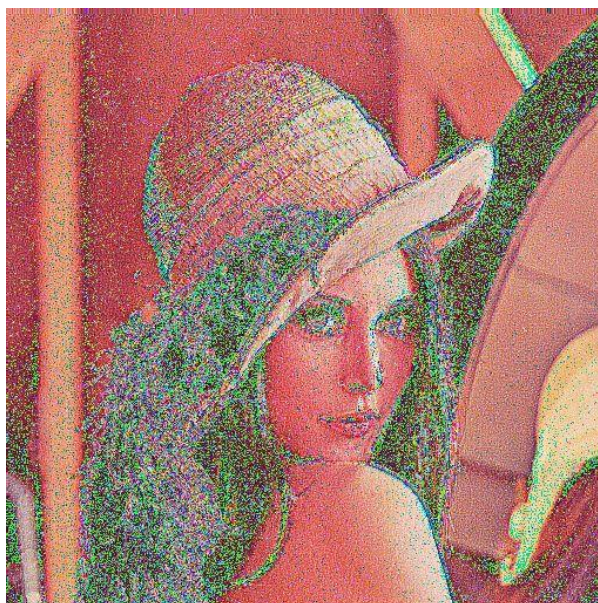


Рис. 14. Результат применения фильтра 5.

Фильтр 6:

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

Фильтр 7:

$$\begin{pmatrix} 0 & -2 & 0 \\ -2 & 5 & -2 \\ 0 & -2 & 0 \end{pmatrix}$$

Фильтр 8:

$$\begin{pmatrix} -1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

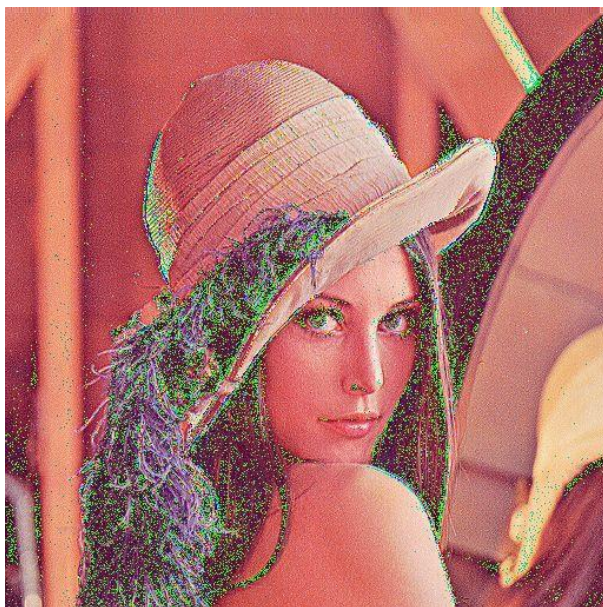


Рис. 15. Результат применения фильтра 6.

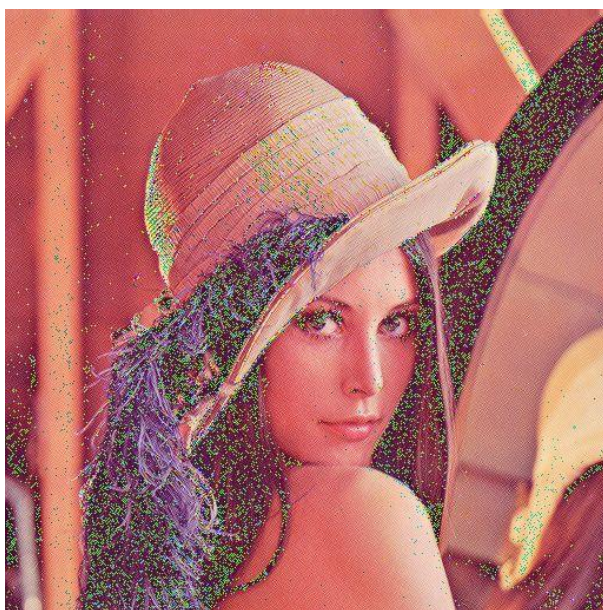


Рис. 16. Результат применения фильтра 7.



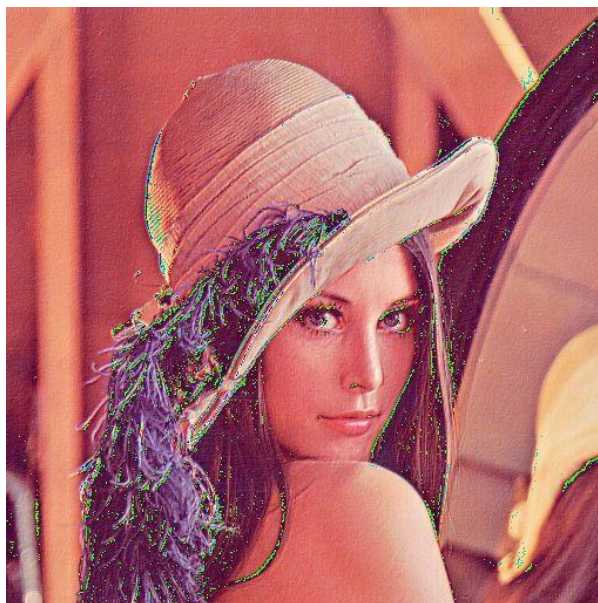


Рис.17. Результат применения фильтра 8.

Оператор Лапласа

Фильтр 9:

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$



Рис. 18. Результат применения фильтра 9.

Фильтр 10:

$$\frac{1}{4} \begin{pmatrix} 0 & 1 & 0 \\ 1 & -8 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$



Рис. 19. Результат применения фильтра 10.

Фильтр 11:

$$\begin{pmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{pmatrix}$$

Эффект тиснения:

Фильтр 12:

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

Фильтр 13:

$$\begin{pmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{pmatrix}$$



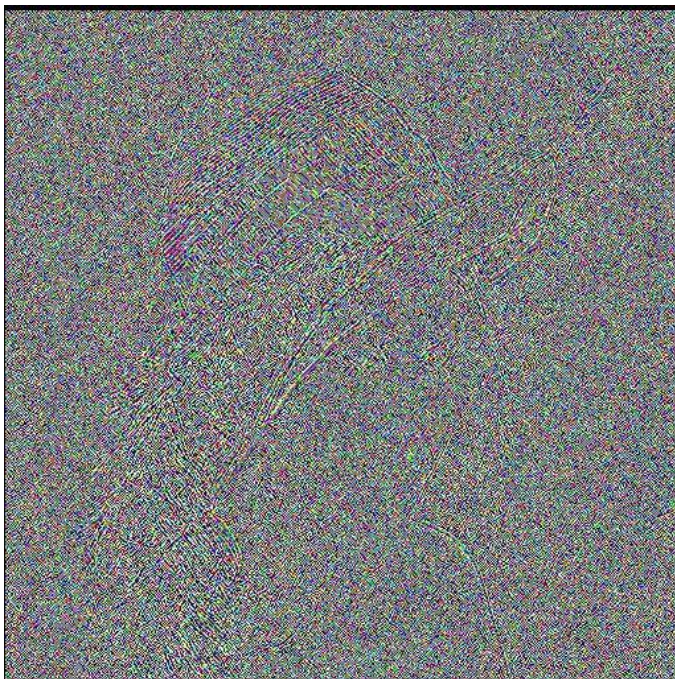


Рис. 20. Результат применения фильтра 11.



Рис. 21. Результат применения фильтра 12.



Рис. 22. Результат применения фильтра 13.

Фильтр 14:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$



Рис. 23. Результат применения фильтра 14.

Удалось добиться ускорения в 30 раз по сравнению с CPU для фильтров 5 на 5 и в 60 раз – для фильтров 3 на 3.

## ЗАКЛЮЧЕНИЕ

На сегодняшний день существует масса средств для цифровой обработки изображений. Но каждое созданное приложение вряд ли окажется лишним. Это связано с тем, что вкусы у всех пользователей разные. Кто-то предпочитает широкий функционал, а кто-то, в свою очередь, доступный интерфейс.

Целью данной курсовой работы было разработать программное средство для фильтрации изображений с использованием технологии CUDA. Разработка данного программного средства требовала знаний в области структуры цифрового хранения изображений, технологии CUDA, разработки приложений на C++.

В ходе разработки были изучены возможности визуальной среды разработки Visual Studio 2017, получены навыки в создании приложений для операционной системы Windows, работе с графическим процессором фирмы NVIDIA.

В результате выполнения данной курсовой работы разработано программное средство для фильтрации изображений с использованием технологии CUDA, рассчитанный на широкий круг пользователей. Простота приложения способствует этому.

Приложение имеет возможность наращивания функционала и расширения возможностей. Со временем данное приложение будет совершенствоваться: будет возможность обработки такими фильтрами, как минимальный, максимальный, минимально-максимальный, медианный, гармонический и другие, обработка каналов по отдельности, учет альфа канала.




## СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Лукашевич М.М, Садыхов Р.Х. Цифровая обработка сигналов и изображений– Минск: БГУИР, 2010 г.
2. ЭРУД по дисциплине «Архитектура высокопроизводительных процессоров» для специальности 1-40 02 01 «Вычислительные машины, системы и сети»
3. Матричные фильтры обработки изображений (электронный ресурс). Электронные данные. Режим доступа: <https://habr.com/ru/post/142818/> - Дата доступа: 15.12.2019.
4. CUDA Toolkit Documentation (электронный ресурс). Электронные данные. Режим доступа: <https://docs.nvidia.com/cuda/index.html> - Дата доступа: 15.12.2019.
5. Convolution (электронный ресурс). Электронные данные. Режим доступа: <http://www.songho.ca/dsp/convolution/convolution.html> - Дата доступа: 15.12.2019.

## ПРИЛОЖЕНИЕ А. ИНТЕРФЕЙС ПРОГРАММЫ

MainWindow



1

☒ Load images

1  1  1

☐ Apply and compare with CPU

1  1  1  1

1  1  1

1

13  0

div offset

Clear

Choose file

Apply