МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

**Лабораторная работа №2**

**по курсу «Программирование графических процессоров»**

**Обработка изображений на GPU. Фильтры.**

Выполнил: Н.А. Зайцев

Группа: 8О-408Б

Преподаватели: К.Г. Крашенинников,

А.Ю. Морозов

Москва, 2016

**Условие**

Описание задачи:

1. **Цель работы​.** ​ Научиться использовать GPU для обработки изображений.
2. **Вариант 2.** Медианный фильтр.

Необходимо реализовать медианный фильтр для изображения. Медианным

элементом считает элемент с номером n / 2, для его нахождения использовать

сортировку подсчетом.

Входные данные. ​На первой строке задается путь к исходному изображению,

на второй, путь к конечному изображению. На следующей строке, целое число r ­­

радиус размытия, r ≤ 100.

**Программное и аппаратное обеспечение**

**GPU:**

Name : GeForce GTX 650

Compute capability : 3.0

Total Global Memory : 2147483648

Shared memory per block : 49152

Registers per block : 65536

Warp size : 32

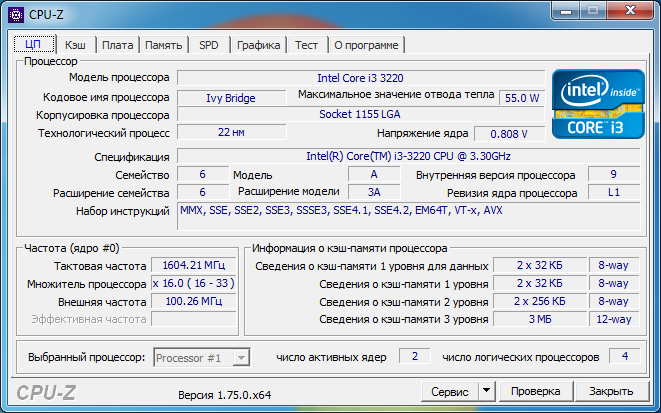
Max threads per block : (1024, 1024, 64)

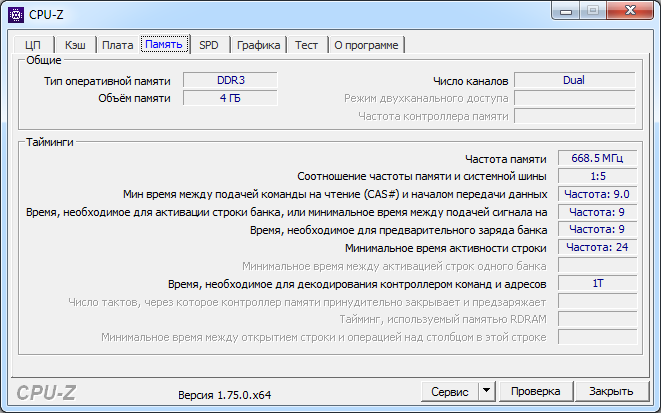
Max block : (2147483647, 65535, 65535)

Total constant memory : 65536

Multiprocessors count : 2

**CPU:**





**HDD:**

Производитель: Seagate

Модель: ST2000DM001-1CH164 ATA Device

Размер : 1,82 ТБ

**OS:**

Windows 7 Professional SP1 64-bit

**IDE:**

Microsoft Visual Studio 2013 + CUDA Plugins

**Compiler:**

nvcc

**Profilers:**

nvprof

nvvp

**Debugger**:

Nvidia CUDA Debugger

**Метод решения**

Программа была реализована при помощи одного ядра. Каждый пиксель изображения обрабатывается одним потоком. Каждый поток заполняет соответствующий пикселю частотный массив элементов матрицы определённого радиуса, и подсчитывается сумма до медианного элемента, после чего этот элемент заносится в переменную в соответствующий цвету байт (линейная сортировка). Так повторяется для всех трёх цветных компонент пикселя. После чего значение переменной (наш итоговый пиксель) заносится в выходное изображение. Была использована двумерная сетка и двумерные координаты для потоков. Количество блоков по каждому измерению задаётся делением соответствующего измерения входного изображения на количество потоков этого же измерения. Количество потоков: по 16 на каждое измерение.

**Описание программы**

Программа состоит из одного файла, но разбита на функцию main и одно ядро.  
  
Ядро kernel\_main:

\_\_global\_\_ void kernel\_main(int height, int width, int r, unsigned int \*src, unsigned int \*dst)

{

int tid\_y = threadIdx.y + blockIdx.y \* blockDim.y;

while (tid\_y < height)

{

int tid\_x = threadIdx.x + blockIdx.x \* blockDim.x;

while (tid\_x < width)

{

int arr\_size;

arr\_size = (MIN(tid\_y + r, height - 1) - MAX(tid\_y - r, 0) + 1) \* (MIN(tid\_x + r, width - 1) - MAX(tid\_x - r, 0) + 1);

int median = arr\_size / 2 + 1;

unsigned int answer = 0;

for (int j = 0; j < 3; j++)

{

unsigned short int C[256];

memset(&C, 0, sizeof(unsigned short int) \* 256);

int y\_up = MAX(tid\_y - r, 0);

int y\_down = MIN(tid\_y + r, height - 1);

int x\_left = MAX(tid\_x - r, 0);

int x\_right = MIN(tid\_x + r, width - 1);

for (int k = y\_up; k <= y\_down; k++)

{

for (int l = x\_left; l <= x\_right; l++)

{

C[(src[k \* width + l] >> (8 \* j)) & 0xFF]++;

}

}

if (C[0] < median)

{

for (int i = 1; i < 256; i++)

{

C[i] += C[i - 1];

if (C[i] >= median)

{

answer |= i << (8 \* j);

break;

}

}

}

}

dst[tid\_y \* width + tid\_x] = answer;

tid\_x += blockDim.x \* gridDim.x;

}

tid\_y += blockDim.y \* gridDim.y;

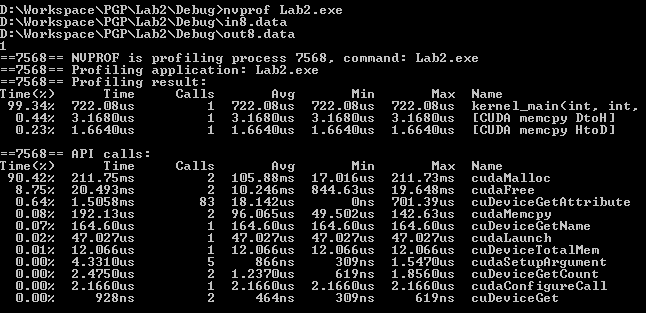
}

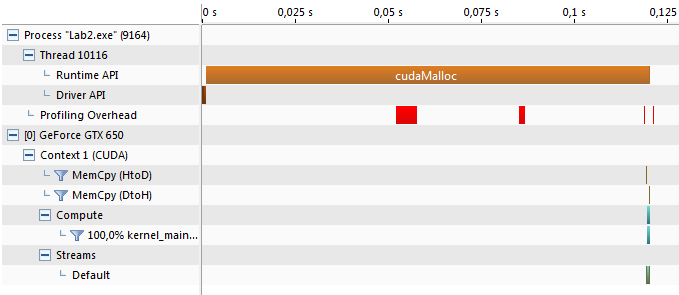
}

Реализует в себе алгоритм, описанный в пункте «Метод решения». Каждый поток может обработать и несколько пикселей, но последовательно, если количества блоков и потоков не хватает, чтобы «покрыть» всё изображение.

**Результаты**

В процессе написания кода я снова попытался использовать профилятор nvprof, но он оказался гораздо менее информативным в данном случае, так как просто показывал время работы единственного ядра моей программы и это не способно сказать мне ничего конкретного, кроме времени работы алгоритма в целом. Список вызовов функций API также не показал ничего полезного. Пример работы профилятора на картинке размером 42х53:

  
  
Я решил попробовать Nvidia Visual Profiler, но он оказался просто графической визуализацией выводимой в консоль информации nvprof’ом и тоже мне не помог, на том же тесте:



Хотя он уже может показать последовательность выполнения выводимой nvprof информации.

В процессе написания программы я узнал, что операция деления достаточно дорогостоящая, и чтобы выделить необходимые биты лучше использовать битовые операции.

Кусочек кода ядра с делениями:

for (int k = y\_up; k <= y\_down; k++)

{

for (int l = x\_left; l <= x\_right; l++)

{

C[(src[k \* width + l] & (0xFF \* mul)) / mul]++;

}

}

if (C[0] < arr\_size / 2 + 1)

{

for (int i = 1; i < 256; i++)

{

C[i] += C[i - 1];

if (C[i] >= arr\_size / 2 + 1)

{

answer |= i \* mul;

break;

}

}

}

И без:

int median = arr\_size / 2 + 1;

…

for (int k = y\_up; k <= y\_down; k++)

{

for (int l = x\_left; l <= x\_right; l++)

{

C[(src[k \* width + l] >> (8 \* j)) & 0xFF]++;

}

}

if (C[0] < median)

{

for (int i = 1; i < 256; i++)

{

C[i] += C[i - 1];

if (C[i] >= median)

{

answer |= i << (8 \* j);

break;

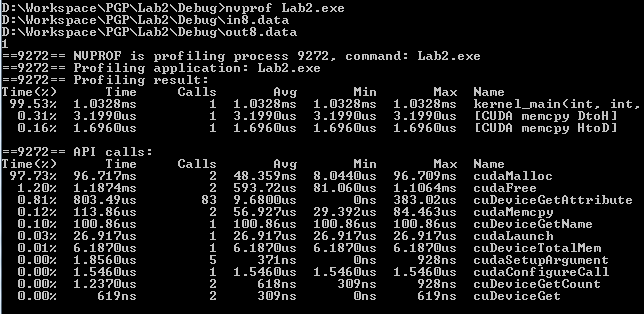
}

}

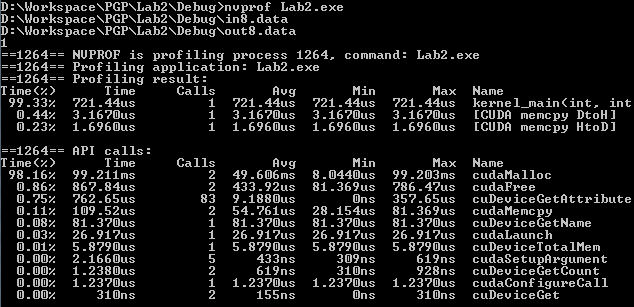
}

Это самые часто используемые инструкции в ядре. Сравним работу на примере предыдущего теста.

До:



И после:



Разница очень заметна. Программа на каждой итерации цикла совершала деления, которые можно было избежать, и тем самым терялось время.

В процессе написания этой лабы мне пришлось освоить Nvidia Cuda Debugger в VS 2013,

и он позволил мне в, казалось бы, проверенном на CPU коде искать ошибки, связанные с тем, что память для статических массивов перестала выделяться, начиная с 4-го потока первого блока. Чтобы понять это, у меня ушёл день. Чтобы этого избежать пришлось отказаться от полноценной реализации линейной сортировки подсчётом с выделением дополнительной памяти и оставить только подсчёт частот и нахождение медианного элемента, а не всей последовательности в целом.

Конечно огромным минусом разработки CUDA приложений на Windows является автоматическое отключение операционной системой графического драйвера в случае, если ядро проработало чуть дольше пары секунд. Что конечно делает тестирование и дебаггинг программ, написанных при помощи одного ядра в особенности, сущим мучением. Я, например, перезагружал Windows hard-reset’ом раз 60, когда писал эту лабораторную работу. Надеюсь, что система не сломается от такого.

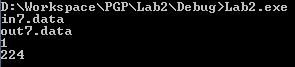
Достоинством разработки на Windows является очень удобная среда Visual Studio 2013 с плагином CUDA, позволяющим использовать CUDA дебаггер.

Вообще большую часть времени занимает не написание кода, а поиск ошибок в нём с сопутствующими перезагрузками, что делает разработку на CUDA сущим мучением. Я уже начинаю посматривать в сторону Linux, но я стараюсь отгонять эти мысли подальше.

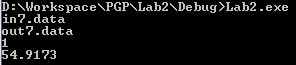
Давайте же сравним производительность программы, реализованной на CPU и GPU с использованием CUDA на более объёмном тесте и будем надеяться, что Windows не отключит драйвер. Вставил в код таймеры непосредственно перед и после выполнения алгоритма.

Использую изображение размером 540х960. Время в миллисекундах.

CPU:



GPU:



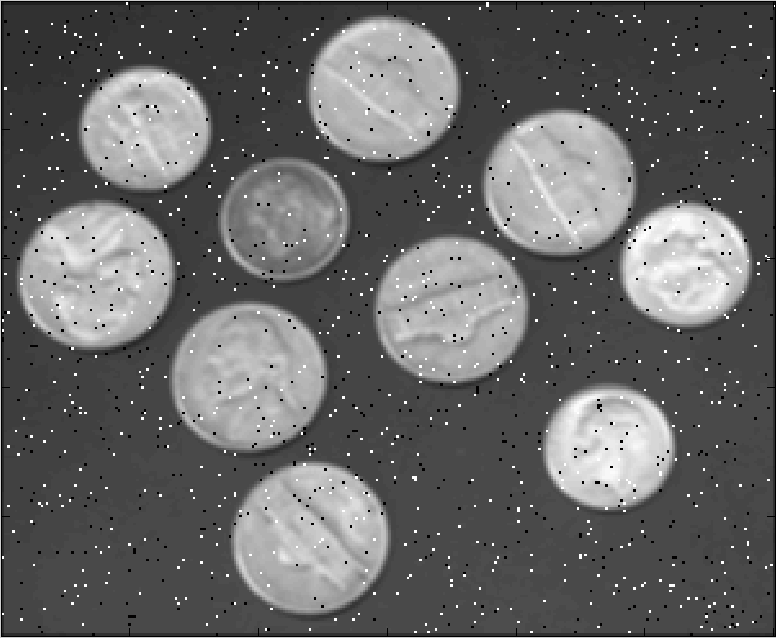
Распараллеленная программа на данном тесте работает в 4 раза быстрее. Разница очень ощутима.

Вообще медианный фильтр широко используется в цифровой обработке сигналов и изображений для уменьшения уровня шума. Продемонстрируем работу алгоритма.

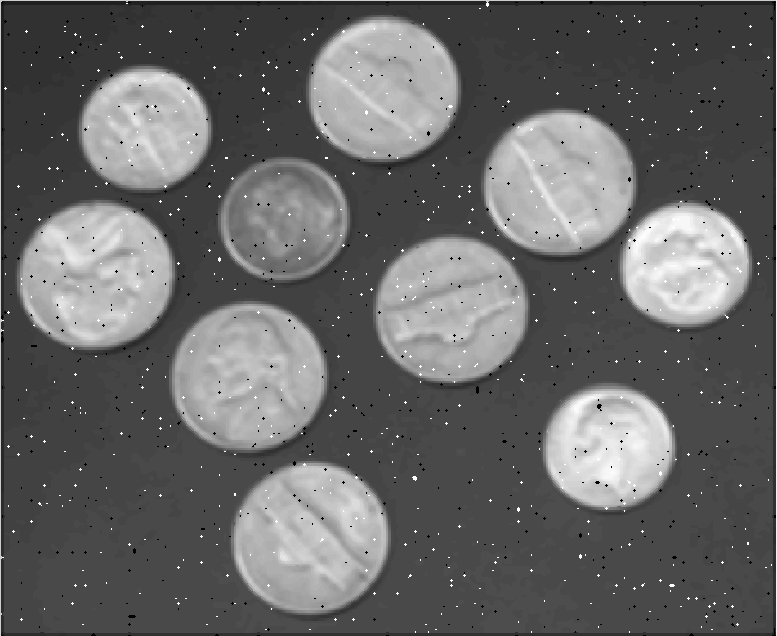
Исходное изображение:



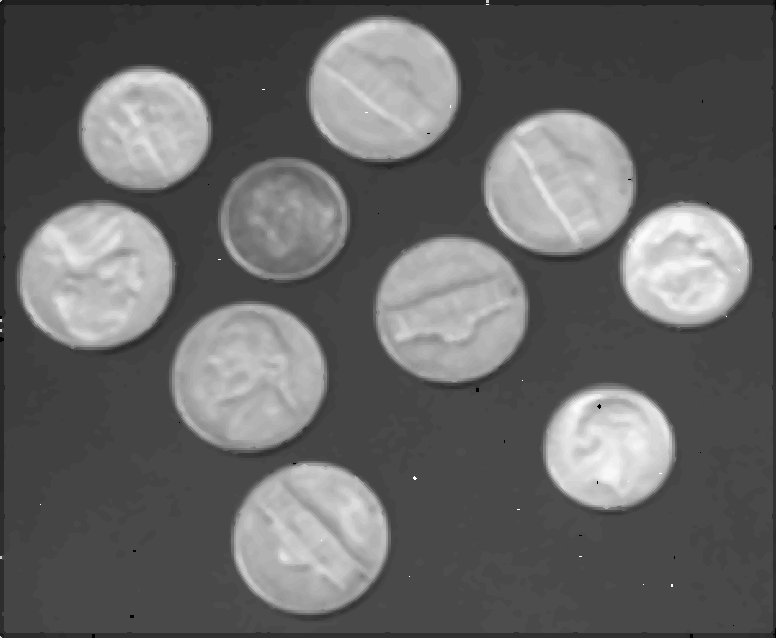
Зашумлённое:



Обработанное при r = 1:



При r = 2:



При r = 3:



Также этот фильтр может использоваться для придания эффекта размытия.

Исходное изображение:



Обработанное при r = 5:



**Выводы**

В результате выполнения данной работы я научился распараллеливать на видеокарте медианный алгоритм фильтрации изображений, а также улучшил свои навыки работы с дебаггером. Наглядно продемонстрировал разницу в производительности программ, написанных с использованием CUDA и без.

**Использованные источники**

1. <https://habrahabr.ru/post/142818/>

2. <https://ru.wikipedia.org/wiki/Медианный_фильтр>

3. https://habrahabr.ru/post/114551/