Отчет по лабораторной работе:

«Масштабирование изображений на основе сверток»

Дисциплина «Параллельное и распределенное программирование».

Выполнили:

студент 2 курса магистратуры, группы 1467

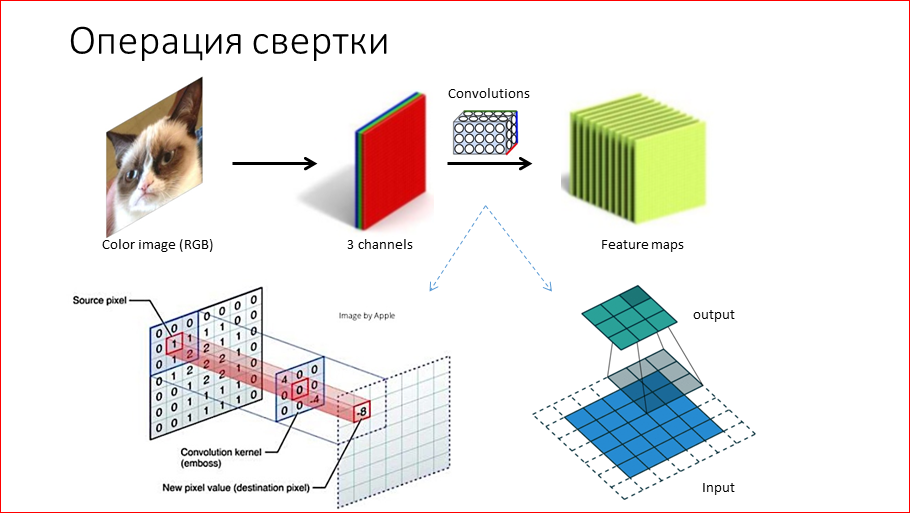
Пешкичев Роман

студент 2 курса магистратуры, группы 1469

Ревенский Александр

**Задание.** Дано некоторое изображение. Требуется реализовать последовательный и два параллельных алгоритма, позволяющие масштабировать изображение. Язык реализации программы С/C++. Параллельные алгоритмы должны использовать библиотеку openMP для распараллеливания на центральном процессоре и CUDA на графическом процессоре.

**Теория.**



Из вышеприведенного изображения следует вывод, что операция свёртки зависит только от ядра свертки и пикселя/пикселей исходного изображения, что в свою очередь говорит о том, что операция свертки не опирается на результаты предыдущих операций и может быть выполнена множеством потоков, то есть параллельно.

**OpenMP.**

Наиболее очевидное и верное решение распараллеливания масштабирования некоторого изображения на основе сверток заключается в том, что обход некоторого изображения выполняется по столбцам и строкам, и это можно распараллелить.

**CUDA.**

**\*/**

**/**

**/**

**/\***

**Результаты:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Исполнитель | Результат | Количество ПЭ | Время | Ускорение(S) | Эффективность(E) |
| CPU | ??? | 1 | 21.398000 | 1.000 | 1.000 |
| CPU | ??? | 8 | 4.279000 | 5.000 | 0.625 |
| GPU | ??? | ??? | 6.729000 | 3.179 |  |

**Последовательный код масштабирования изображения на основе билинейной свертки:**

int\* resizeBilinear\_cpu(int32\_t\* pixels, int w, int h, int w2, int h2)

{

int32\_t\* temp = new int32\_t[w2\*h2];

int32\_t a, b, c, d, x, y, index;

float x\_ratio = ((float)(w - 1)) / w2;

float y\_ratio = ((float)(h - 1)) / h2;

float x\_diff, y\_diff, blue, red, green;

int offset = 0;

for (int i = 0; i<h2; i++)

{

for (int j = 0; j<w2; j++)

{

x = (int)(x\_ratio \* j);

y = (int)(y\_ratio \* i);

x\_diff = (x\_ratio \* j) - x;

y\_diff = (y\_ratio \* i) - y;

index = (y\*w + x);

a = pixels[index];

b = pixels[index + 1];

c = pixels[index + w];

d = pixels[index + w + 1];

// blue element

// Yb = Ab(1-w)(1-h) + Bb(w)(1-h) + Cb(h)(1-w) + Db(wh)

blue = (a & 0xff)\*(1 - x\_diff)\*(1 - y\_diff) + (b & 0xff)\*(x\_diff)\*(1 - y\_diff) +

(c & 0xff)\*(y\_diff)\*(1 - x\_diff) + (d & 0xff)\*(x\_diff\*y\_diff);

// green element

// Yg = Ag(1-w)(1-h) + Bg(w)(1-h) + Cg(h)(1-w) + Dg(wh)

green = ((a >> 8) & 0xff)\*(1 - x\_diff)\*(1 - y\_diff) + ((b >> 8) & 0xff)\*(x\_diff)\*(1 - y\_diff) +

((c >> 8) & 0xff)\*(y\_diff)\*(1 - x\_diff) + ((d >> 8) & 0xff)\*(x\_diff\*y\_diff);

// red element

// Yr = Ar(1-w)(1-h) + Br(w)(1-h) + Cr(h)(1-w) + Dr(wh)

red = ((a >> 16) & 0xff)\*(1 - x\_diff)\*(1 - y\_diff) + ((b >> 16) & 0xff)\*(x\_diff)\*(1 - y\_diff) +

((c >> 16) & 0xff)\*(y\_diff)\*(1 - x\_diff) + ((d >> 16) & 0xff)\*(x\_diff\*y\_diff);

temp[offset++] =

0xff000000 |

((((int32\_t)red) << 16) & 0xff0000) |

((((int32\_t)green) << 8) & 0xff00) |

((int32\_t)blue);

}

}

return temp;

}

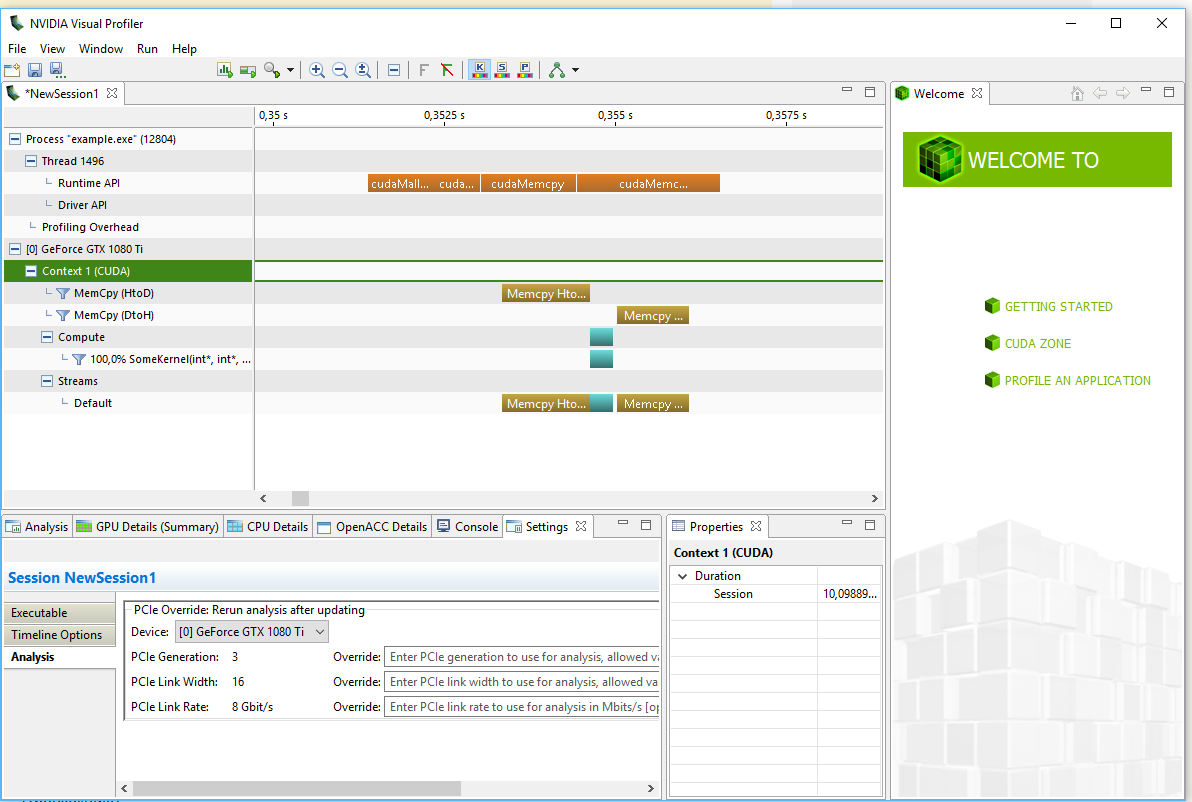
**Вывод:** распараллеливающие алгоритмы позволяют получить более быстрые результаты, но требуют очень четкого понимания того, как это можно достичь в деталях, так как повышение эффективности такого рода алгоритмов напрямую зависит от этого.

Оптимизация CUDA:

Архитектура Pascal

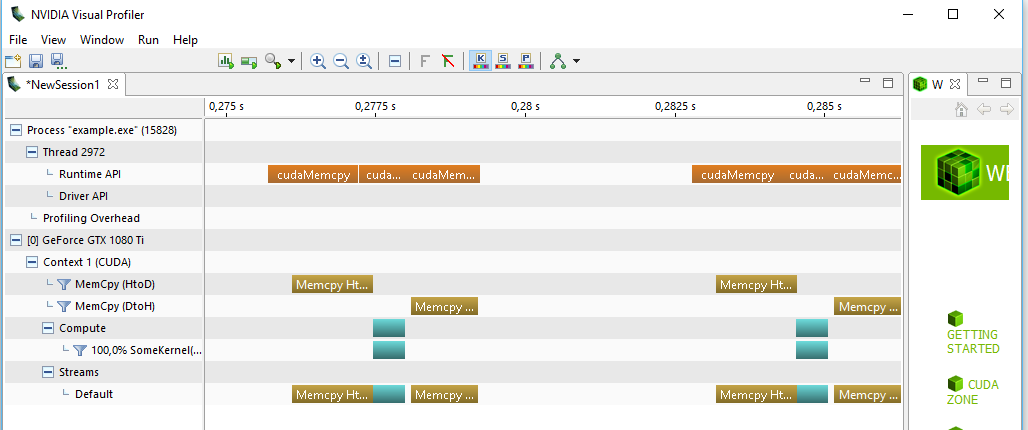
Первый вариант: 1024 потока, 1 поток обрабатывает 1 пиксель.

Время: 6.729000 секунд на 1000 вызовов (1920x1080)->(1280x1280)

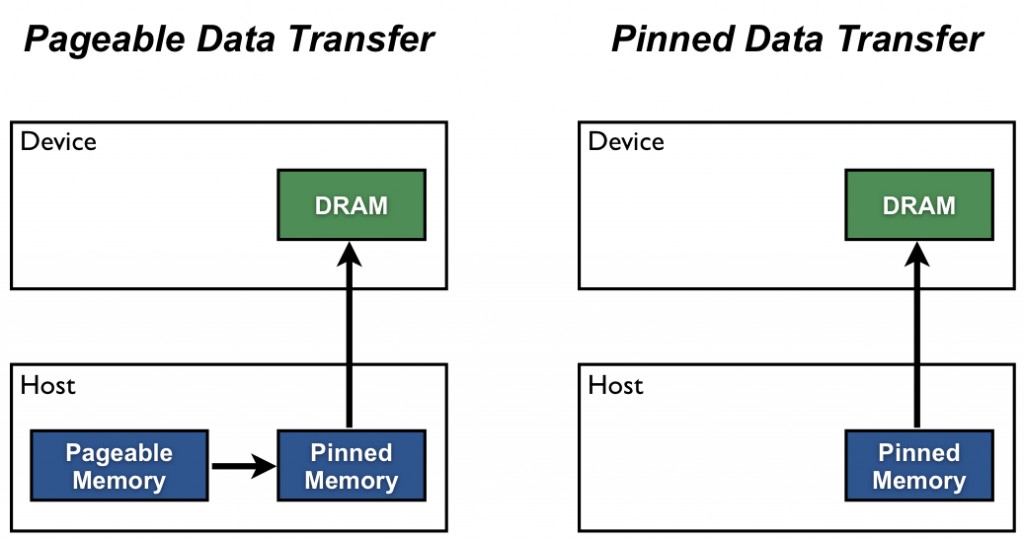


Оптимизация программ, использующих CUDA подразумевает увеличение отношения времени непосредственного выполнения полезного действия к операциям с памятью.

Используя инструменты профилирования можно увидеть, что очень много времени занимает первый вызов функции ресайза, т. к. требуется инициализация устройства, поэтому логично инициализировать устройство заранее. Также можно увидеть, что значительная часть времени тратится на операции выделения памяти на устройстве, копирование данных с хоста на устройство и освобождение памяти на устройстве. Для улучшения этих параметров логично выделить память заранее при инициализации устройства (пусть функция инициализации принимает на вход размеры максимально допустимого для ресайза изображения). Тогда освобождать выделенную память можно при завершении работы программы в функции деинициализации. Таким образом сократилось число операций с памятью и время выполнения составило 5.024000 секунд.



Далее, можно заметить, что операции копирования из памяти хоста в память устройства занимают и обратно слишком много времени. Это происходит т. к. для такого копирования необходимо использовать закрепленную память и cudaMemCpy() создает буфер с закрепленной памятью, который служит посредником между памятью устройства и хоста. Чтобы избежать этого необходимо выделять память под изображения на хосте не с помощью malloc(), а с помощью специальной функции cudaMallocHost(). Использование такого механизма позволило улучшить время работы до 1.346000 секунд.



Дальнейшие пути улучшения:

Использование текстурной памяти GPU.

Тестирование:

CPU: Ryzen 1800x (16 ядер, 3600 МГц)

GPU: Nvidia 1080Ti (3584 ядер cuda, 1569 Мгц)

Измеряется суммарное время в секундах за 1000 вызовов.

1.Ресайз 1920x1080 → 1280x1280

CPU: 21.667

CPU(OMP): 3.637

GPU: 1.372

2. Ресайз 1920x1080 → 3840x2160

CPU: 108.581

CPU(OMP): 18.080

GPU: 3.602

3. Ресайз 1920x1080 → 960x540

CPU: 6.812

CPU(OMP): 1.144

GPU: 0.979