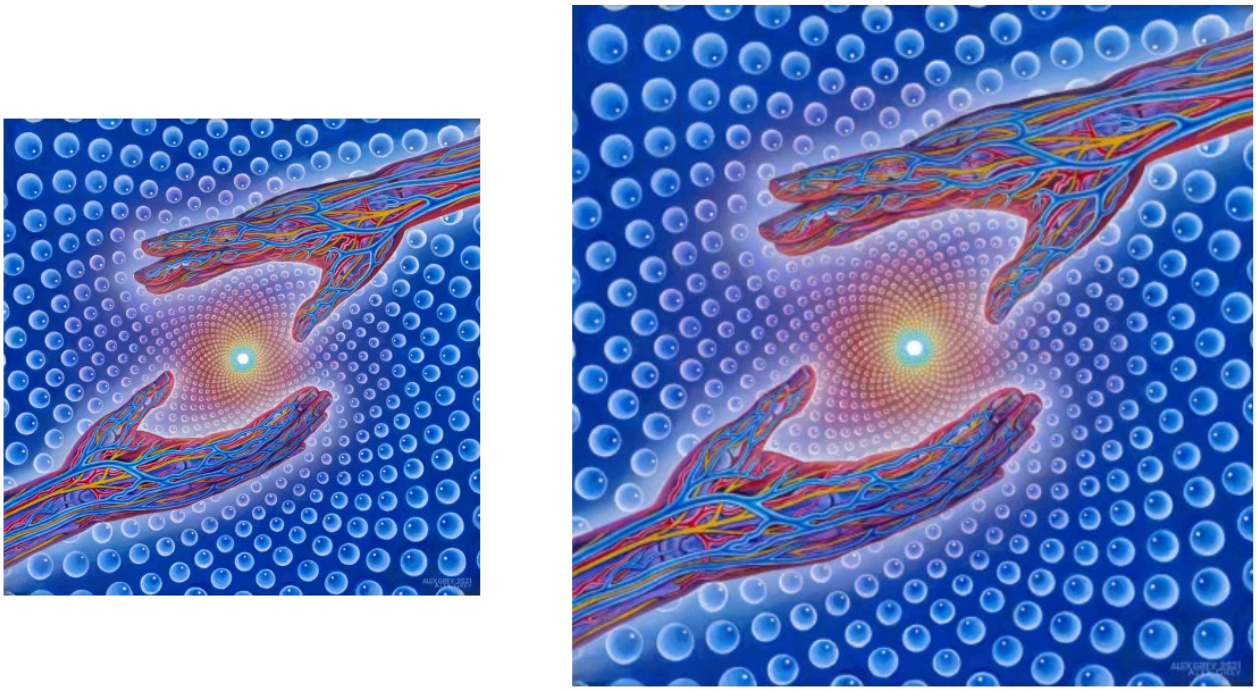
Interpolacja Bilinearna przy pomocy karty graficznej i frameworka OpenCL

Marek Kasprowicz



Przeskalowany obraz z 420x420 na 600x600 przy pomocy karty graficznej i frameworka OpenCL

Interpolacja Bilinearna [^1][^2][^3][^4]

Proces stosowany przy skalowaniu zbiorów danych, najczęściej obrazów. Polega on na wyznaczniu wartości dla nowej struktury danych w oparciu o informacje zawarte w pierwszej. Należy dla każdego nowego elementu tablicy znaleźć cztery wartości odpowiadające mu w początkowej tablicy i wyznaczyć nowe na podstawie odległości od tych punktów.

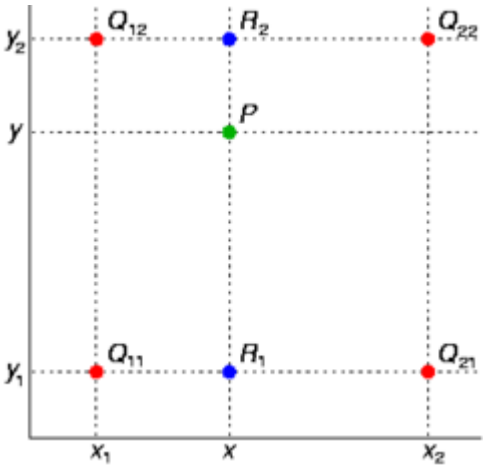
W pierwszej kolejności należy stworzyć pusty obraz odpowiednio przeskalowany, a następnie należy wyznaczyć skalę powiększenia.

```
ratio_x = input.width / output.width
ratio_y = input.height / output.height
```

Pozwoli to na wyznaczanie odpowiadających indeksów w drugim obrazie.

```
x = x_ratio * j
y = y_ratio * i
```

Ponieważ wartością x i y będą zmienne zmiennoprzecinkowe, to po zastosowaniu ceil i floor w łatwy sposób wyznaczamy punkty przegowe, pomiędzy którymi znajduje się interesujący nas punkt.



Następnie wyznacza się zmienną ciężkości, określającą drogę przebytą między dwoma bocznymi pikselami.

```
xw = (x_ratio * j) - x0
yw = (y_ratio * i) - y0
```

Na końcu zastosowuje się dla każdego koloru piksela wyprowadzony wzór[^8] na interpolację między czterema punktami:

$$f(x,y) \approx f(0,0)\,(1-x)(1-y) + f(1,0)\,x(1-y) + f(0,1)\,(1-x)y + f(1,1)xy.$$

OpenCL [^5][^6]

Opensourcowy framework stosowany do tworzenia oprogramowania na różnych ALU. W przeciwieństwie do CUDA nie jest ograniczony tylko do konkretnego rodzaju hardware. Raz napisany kod w OpenCL można uruchomić na niemalże każdym sprzęcie, wliczając w to GPU, CPU lub FPGA). Aktaulnie jest w posiadaniu grupy Khronos i najwyższą jego wersją jest 3.0. Natywnie kod OpenCL tworzy się za pomocą języka C, jednak Khronos wprowadził headery interpretujące polecenia na obiekty C++, co znacznie poprawiło czytelność kodu i zmniejszyło "biolerplate". Istnieją również nakładki na inne języki takie jak:

1. Python - https://pypi.org/project/pyopencl/
2. Java - https://github.com/gpu/JOCL
3. C# - https://www.nuget.org/packages/OpenCL.Net
4. C++ - https://github.com/KhronosGroup/OpenCL-CLHPP

Omówienie kodu

Projekt składa się z 5 plików:

- *main.cpp* - Wczytuje obraz, wywołuje skalowanie na GPU i CPU. Zbiera czas wykonywania i zapisuje do plików .csv.
- *CPU.cpp*- Algorytm dwuliniowej interpolacji wykonywany sekwencyjnie na procesorze, przy pomocy obiektów biblioteki OpenCV.
- *GPU.cpp* - Plik przygotowywujący GPU i konwertujący obraz OpenCV do formatu możliwego do przekazania dla GPU.
- *bilinear_interpolation.cl* - Algorytm dwuliniowej interpolacji wykonywalny na GPU, jest wczytywany w GPU.cpp.
- */test/plot.py* - Interpretuje dane zebrane podczas testów i pokazuje je w formie wykresu.

Oprócz tego znajdują się tu pliki:

- *image.png* - Testowy obraz do skalowania.
- *test/cpu.csv* - Wyniki testów dla cpu.
- *test/gpu.csv* - Wyniki testów dla gpu.
- *test/gpuVScpu.png* - Wykres porównania.
- *README.md*, *README.html*, *README.pdf* - dokumentacja projektu (html i pdf zostały wygenerowane z README.md).

Najważniejszą częścią projektu są pliki main.cpp, GPU.cpp i bilinear_interpolation.cl, dzięki nim wywołuje się algorytm interpolacji na GPU.

Wczytanie obrazu z dysku twardego

Do manipulacji obrazami wykorzystana została biblioteka OpenCV, importowalna przez bibliotekę *<opencv2/opencv.hpp>*.

```
cv::Mat load_image(string file_name) {
    cv::Mat image = cv::imread(file_name);
    if (image.empty()) {
        cout << "Image not loaded" << endl;
    }

    cv::cvtColor(image, image, cv::COLOR_BGR2RGBA);
    return image;
}
```

Metoda zwraca obraz w obiekcie cv::Mat, który jest strukturą danych przechowywującą kolory pikseli. Warto zwrócić uwagę, że format jest konwertowany z BGR na RGBA, gdyż wczytywanie obrazów przez OpenCL jest zoptymalizowane dla 4 wartości.

Platformy i urządzenia

Na początku bloku kodu OpenCL należy zadeklarować z jakiej platformy i jakiego urządzenia będzie się korzystać.

```
void setup_device(cl::Platform &platform, cl::Device &device) {
    std::vector<cl::Platform> all_platforms;
```

```
        cl::Platform::get(&all_platforms);
        if (all_platforms.size() == 0) {
            std::cout << " 0 platforms \n";
            exit(1);
        }

        platform = all_platforms[0];

        std::vector<cl::Device> all_devices;
        platform.getDevices(CL_DEVICE_TYPE_ALL, &all_devices);
        if (all_devices.size() == 0) {
            std::cout << " 0 devices\n";
            exit(1);
        }

        device = all_devices[0];
    }
```

Metoda ta zwraca pierwszą platformę i pierwsze urządzenie w niej zawarte. Jest to łatwy sposób na testowanie programu na wielu urządzeniach, jeśli tylko posiada się urządzenia wspierające OpenCL.

Aby wyświetlić informacje o platformach i urządzeniach, należy wykonać na nich metodę getInfo.

```
    std::cout << platform.getInfo<CL_PLATFORM_NAME>() << ": " <<
device.getInfo<CL_DEVICE_NAME>() << "\n"
```

Kontekst, kolejka zadań, program i kod jądra

W poniższym kodzie występuje deklaracja urządzenia na którym program ma wykonywać zadania, oraz do źródła zostaje wczytany plik *bilinear_interpolation.cl* posiadający instrukcje dla GPU. Program zostaje zbudowany, a potem tworzony jest Kernel, ze wskazaniem która metoda z wczytanego pliku zostanie wywołana.

```
    cl::Context context({ device });
    cl::CommandQueue queue(context, device, CL_QUEUE_PROFILING_ENABLE);

    cl::Program::Sources sources;
    string kernel_code = get_kernel("bilinear_interpolation.cl");
    sources.push_back({ kernel_code.c_str(), kernel_code.length() });

    cl::Program program(context, sources);
    if(program.build({ device }) != CL_SUCCESS) {
        std::cout << "Error building: " <<
program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(device) << "\n";
        exit(1);
    }

    cl::Kernel kernel(program, "bilinear_interpolation");
```

Kernel

Polecenia wykonywalne przez urządzenie OpenCL, realizujące interpolację. Oznaczony jest flagą *__kernel*

sampler_t

```
  __constant sampler_t sampler =
CLK_NORMALIZED_COORDS_FALSE|CLK_ADDRESS_CLAMP|CLK_FILTER_NEAREST;
```

Sampler jest obiektem koniecznym do manipulacji obrazem. Ustawia się na nim flagi odnoszące się np. do normalizacji koordynatów, przybliżania lokaliazcji czy specyfikacji adresowej.

image2d_t

Reprezentacja obiektu *cl::Image2D* wewnątrz kodu kernela. Aby manipulować danymi sotsuje się dwie metody:

```
read_imagef(image, sampler, (int2)(x, y))
write_imagef(image, (int2)(x, y), color)
```

Kolor jest przestawiony jako typ *float4*, czyli wektor czterech floatów, gdzie:

```
red:    float4.x
green:  float4.y
blue:   float4.z
alpha:  float4.w
```

get_global_id(n)

Używany do uzyskania indeksu "obrotu pętli" w kernelu, która jest deklarowany w *cl::NDRange*. Ponieważ w kodzie została wywołana opcja dwuargumentowa, więc dostępne są dwa identyfikatory pod indeksem 0 i 1.

```
  int i = get_global_id(0);
  int j = get_global_id(1);
```

Modyfikatory

Modyfikatory zmiennych pozwalają na przyspieszenie obliczeń, gdyż urządzenie odwołuje się do konkretnych adresów.

```
__read_only
__write_only
const
```

Załadowanie i konwersja obrazu dla GPU

Projekt stosuje OpenCV[^7] do pracy na obrazach. Aby obiekt cv::Mat mógł być zrozumiany przez GPU musi on być zmapowany do wektora o wielkości **height * width * 4**.

```
  Piksele w obrazie 2d:   [0, 0]      [0, 1]          [n, n]
  Piksele w tablicy 1d: { r, g, b, a, r, g, b, a, ... r, g, b, a }
```

Po konwersji należy zadeklarować obiekt *cl::Image2D* służacy do przekazania danych do GPU. Do obiektu należy przekazać kontekst, stwierdzić z jaką flagą ma być odczytywany w funkcji kernelowej, określić format przy pomocy *cl::ImageFormat* (format kolorów i ty zmiennych w tablicy 1d), oraz rozmiar zadeklarowanego obrazu.

```
  cl::ImageFormat format(CL_RGBA, CL_UNORM_INT8);
  cl::Image2D Input_Image(context, CL_MEM_READ_ONLY, format,
input_image.size().width, input_image.size().height);
  cl::Image2D Output_Image(context, CL_MEM_WRITE_ONLY, format, image_size,
image_size);
```

Aby załadować obraz do jądra, należy jeszcze określić koordynaty początku i końca obrazu.

```
  queue.enqueueWriteImage(Input_Image, CL_TRUE, origin, input_region, 0, 0,
&input_arr[0]);
```

Wywołanie programu jądra

Przekazywanie obiektów do jądra wykonuje się przy pomocy metody setArg, gdzie należy wskać na indeks obiektu i go przekazać. Program rozpocznie się gdy wywołamy na kolejce *enqueueNDRangeKernel*, która uruchamia pętlę o wielkości zadeklarowanej w *cl::NDRange*.

```
  kernel.setArg(0, Input_Image);
  kernel.setArg(1, Output_Image);
  kernel.setArg(2, sizeof(float), &x_ratio);
  kernel.setArg(3, sizeof(float), &y_ratio);

  queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(image_size,
image_size), cl::NullRange, NULL);
```

cl::NDRange(image_size, image_size) jest odpowiednikiem pętli:

```
  for(int i = 0; i < image_size; i++) {
    for(int j = 0; j < image_size; j++) {
      ...
    }
  }
```

Odczyt danych i zapis obrazu

Odczytanie obrazu realizuje metoda enqueueReadImage, zachowująca się bliźniaczo do enqueueWriteImage. Obraz jest zapisany w zmiennej output_arr, a potem przekonwertowany na obiekt cv::Mat.

```
  std::vector<uchar> output_arr(image_size * image_size * 4);
  queue.enqueueReadImage(Output_Image, CL_TRUE, origin, output_region, 0, 0,
&output_arr[0]);
  cv::Mat output_image(image_size, image_size, CV_8UC4, output_arr.data());
```

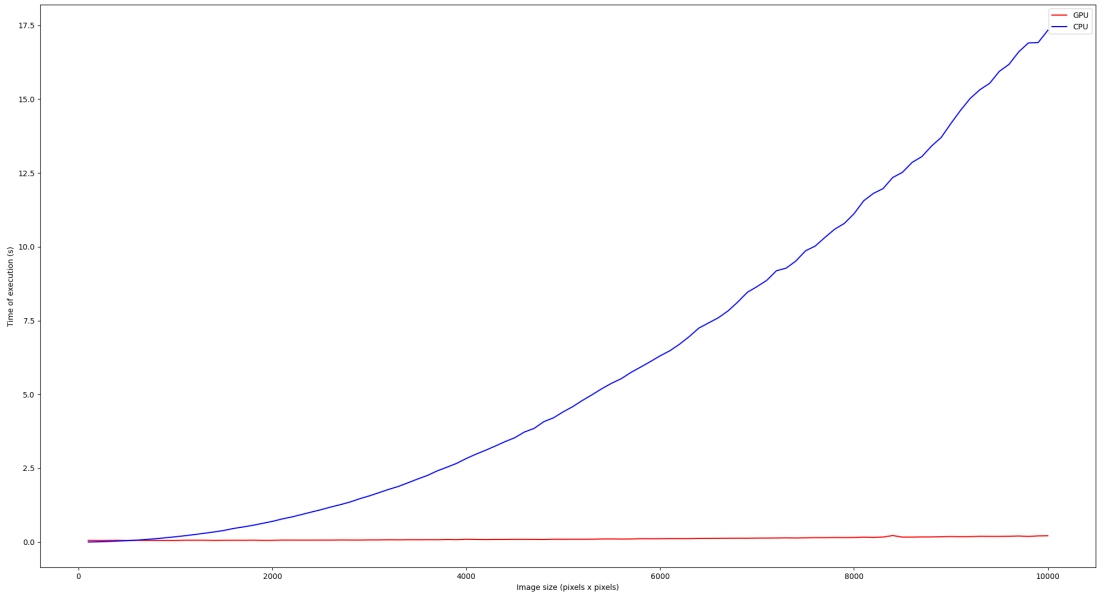
Pomiar czasu

Czas wykonania mierzony jest przy pomocy biblioteki *chrono* i nie zawiera on wczytywania i zapisywania obrazu na dysk, ponieważ nie są to istotne dla równoległości elementy.

Porównanie GPU i CPU

Czasy wykonania

W trybie testu projekt tworzy pliki gpu.csv i cpu.csv zawierające czase wykonania skalowania obrazów z zakresu 100-10000, wykonywanywanego co 100 pikseli. Przy pomocy skryptu */test/plot.py* są interpretowane i konwertowane do wykresu.



Czasy wykonywania z początku są równe, bądź nawet lepsze dla CPU. Trend ten zmienia się gdy należy przeskalować obraz do formatu 300x300. Od tego czasu wykładniczo zwiększa się czas wykonania dla CPU, podczas gdy GPU oscyluje na podobnym poziomie, jednak może to być spowodowane operacjami tworzenia wektorów obrazów i zamiany ich na cv::Mat.

W finalnym punkcie czas wykonania dla CPU jest aż *81,20* razy większy niż programu na GPU.

Poprawność obrazów

Obrazy wykonane przez cpu i gpu są identyczne. Można to sprawdzić przy pomocy narzędzia wykrywającego zmiany w obrazach online np. https://products.groupdocs.app/comparison/compare?FolderName=0b8244d6-fd84-4c2b-9180-36429426cc98&FirstFileName=cpu_out.png&SecondFileName=gpu_out.png

Biblioteki, kompilacja i uruchamianie

OpenCL

```
sudo apt install opencl-headers ocl-icd-opencl-dev -y
```

OpenCV

```
sudo apt install python3-opencv libopencv-dev
```

Kompilacja

```
g++ main.cpp -lOpenCL `pkg-config --cflags --libs opencv4`
```

Wywołanie

```
./a.out [plik_wejsciowy] [plik_wyjsciowy] [rozmiar_nowego_pliku] [tryb_testowy]
```

- [plik_wejsciowy] - Nazwa pliku wejściowego. Np. "in.png"
- [plik_wyjsciowy] - Nazwa pliku wyjściowego. Np. "out.png"
- [rozmiar_nowego_pliku] - Miara do której będzie skalowany obraz, podany w pixelach. Np. 100
- [tryb_testowy] - W trybie testowym nie są produkowane obrazy wyjścia, jednak tworzą się pliki z danymi.
    - 0 - brak testów
    - 1 - testowanie

Biblioteki do wykresów

```
pip install pandas
pip install matplotlib
```

Wywołanie kodu rysującego wykres

```
python3 plot.py
```

Komentarz osobisty

Praca z biblioteką OpenCL była bardzo trudna. Istniejące materiały są przestarzałe i odnoszą się najczęściej do wersji C. Ogólnie rzecz biorąc istnieje bardzo mała ilość użytecznych przykładów, a dochodzenie do sensownego rozwiązania zajmuje sporo czasu. Najwięcej czasu zajęła mi konwersja obiektu cv::Mat na formę odczytywalną przez GPU, problemy z którymi się spotkałem to: niepoprawny format danych (BGR zamiast RGBA), konwersja uchar na float czy problemy z odczytaniem poprawnych wartości przez GPU.

Źródła

[^1]: Bilinear Interpolation Calculator: https://www.omnicalculator.com/math/bilinear-interpolation [^2]: Zmiana wielkości obrazu - Interpolacja dwuliniowa: http://www.algorytm.org/przetwarzanie-obrazow/zmiana-wielkosci-obrazu-interpolacja-dwuliniowa.html [^3]: The AI Learner Bilinear Interpolation: https://theailearner.com/2018/12/29/image-processing-bilinear-interpolation/ [^4]: Understanding Bilinear Image Resizing: https://chao-ji.github.io/jekyll/update/2018/07/19/BilinearResize.html [^5]: Khronos OpenCL Guide: https://github.com/KhronosGroup/OpenCL-Guide [^6]: OpenCL Programming by Example By Ravishekhar Banger, Koushik Bhattacharyya: https://www.packtpub.com/product/opencl-programming-by-example/9781849692342 [^7]: OpenCV C++ Tutorial: https://www.opencv-srf.com/2017/11/load-and-display-image.html [^8]: Interpolacja dwuliniowa: https://pl.wikipedia.org/wiki/Interpolacja_dwuliniowa