Meetrapport snelheid bilineaire interpolatie bij schaling afbeeldingen

Chris Smeele (1681546), Philippe Zwietering (1685431)

04-04-2017

Contents

1 Doel **1**
1.1 Hypothese 1

2 Werkwijze **1**

3 Resultaten **2**

4 Verwerking **2**

5 Conclusie **3**

6 Evaluatie **3**

7 Appendix **3**
7.1 Code bilineaire schaling 3

1 Doel

We willen in dit onderzoek kijken wat het verschil is in snelheid van twee verschillende implementaties van bilineaire interpolatie om afbeeldingen te schalen. Hierbij willen we graag het verschil bekijken tussen onze eigen implementatie en de implementatie die al wordt gebruikt in de bestaande implementatie van dit project, die gebruik maakt van de OpenCV library. Het verschil in snelheid kan gemakkelijk gemeten worden door het verschil in tijd te meten tussen het begin en einde van de schalingsstap in beide implementaties.

1.1 Hypothese

Wij denken dat onze eigen implementatie sneller zal zijn omdat deze multi-threading toepast, waardoor er meerdere rijen tegelijkertijd kunnen worden geschaald.

2 Werkwijze

Onze code voor de bilineaire interpolatie en het schalen staan in appendix 7.1.

Voor het verschil in snelheid tussen de twee algoritmes is gekeken naar de gemiddelde tijdsduur voor een test afbeelding om te worden geschaald. We hebben de volgende testafbeelding bekeken, deze afbeelding geeft voor beide algoritmen een succesvolle gezichtsherkenning:

Om het tijdsverschil tussen het begin en einde van de schalingsstap te bepalen is gebruik gemaakt van de chrono namespace van de standaard library van c++ 2014. Aan het begin en einde van de schalingsstap wordt gekeken wat de tijd is met de `now()` methode op een `std::chrono::steady_clock` en het verschil hiertussen wordt uitgeprint op het scherm in milliseconden, met 6 cijfers achter de komma.

Figure 1: Onbwerkte testafbeelding

3 Resultaten

We hebben beide implementaties 10.000 keer gedraaid op Philippe zijn laptop, met behulp van een Python script. Hieruit krijgen we het volgende gemiddelde, zie tabel 1. De significantie van dit resultaat is platform afhankelijk en daarom zal de nauwkeurigheid per systeem afwijken. We gebruiken echter de `steady_clock`, en zoals te zien is in de C++-manual is deze specifiek bedoeld voor het meten van tijdsintervallen. We gaan er daarom vanuit dat de nauwkeurigheid minstens goed genoeg is om het verschil in microsecondes goed te kunnen onderscheiden. Dit betekent dat alleen de eerste drie cijfers achter de komma betekenis hebben en dat we de rest weg kunnen gooien. We controleren at compile time dat de resolutie van de tijdmeting nauwkeurig genoeg is, met behulp van een `static_assert`. Als we alle tienduizend metingen middelen, dan krijgen we:

Table 1: Tijdsduur verschillende implementaties

OpenCV algoritme	Eigen algoritme
Tijdsduur (ms)	Tijdsduur (ms)
1.344	5.457

4 Verwerking

Aangezien alle metingen onder dezelfde omstandigheden op hetzelfde systeem zijn uitgevoerd is het logisch om aan te nemen dat de nauwkeurigheid van alle metingen ook hetzelfde is. In de Resultaten 3 was al bepaald dat onze meetresultaten vier significante cijfers hebben, aangezien de meeste metingen niet boven de 10 milliseconden uitkomen. Ook was het kleinste cijfer achter de komma gegarandeerd, wat betekent dat de standaarddeviatie van iedere meting hoogstens 0.00005 ms is. Dit is verwaarloosbaar ten opzichte van de metingen.

Het verschil tussen beide meetseries is aanzienlijk. Ons eigen algoritme is 4.060 keer langzamer dan het OpenCV algoritme.

5 Conclusie

Uit ons onderzoek is gebleken dat ons bilineaire interpolatie algoritme voor het schalen van afbeeldingen 4 keer langzamer is dan het algoritme dat gebruikt wordt door de OpenCV library. Dit is gemeten door beide algoritmes 10.000 keer te draaien en de tijdsduur te meten met behulp van een standaard C++-library. Dit resultaat is nogal verassend en opvallend groot, we verwachtten zelf immers ook dat ons eigen algoritme sneller zou zijn, vooral omdat het OpenCV algoritme niet multi-threaded is. We weten zelf niet waardoor dit komt, omdat de methode van het algoritme hetzelfde is, het zijn allebei algoritmen voor bilineaire interpolatie.

6 Evaluatie

Het maken van de implementatie van bilineaire interpolatie ging goed. Het multi-threaded maken hebben we zelfs in een template klasse kunnen bouwen. Ook het maken van de andere interpolatie-algoritmen ging relatief soepel. Voor het schrijven van de tekst stelde vooral Philippe steeds uit, wat ertoe leidde dat dit enigszins laat ingeleverd werd. Over de inhoud echter zijn we allebie wel tevreden.

Wat ons nog wel stoort is dat we er niet achter zijn gekomen waarom het OpenCV algoritme sneller is dan het onze. We hebben het niet expres langzaam gemaakt of anderszins negatief proberen beïnvloeden en we gebruiken zelfs meer resources dan het OpenCV algoritme, juist om te zorgen dat het sneller gaat. Waarschijnlijk zijn de mensen die de OpenCV library hebben ontwikkeld meer ervaren programmeurs dan wij die betere methoden hebben, maar we zouden zelf niet weten hoe ze het daadwerkelijk voor elkaar hebben gekregen.

7 Appendix

7.1 Code bilineaire schaling

~static IntensityImage *scaleBilinearMt(const IntensityImage &image, std::tuple<unsigned,unsigned,double> dim, IntensityImage *out) {
const double origScale = 1 / std::get<2>(dim);
draadificeer(std::get<1>(dim), [&image, &dim, origScale, out](size_t rowStart, size_t rowCount) {
for (uint y = rowStart; y < rowStart + rowCount; ++y) {
double origY = origScale * y; unsigned y1 = floor(origY); unsigned y2 = ceil(origY);
double yFrac = origY - (long)origY;
for (uint x = 0; x < std::get<0>(dim); ++x) { double origX = origScale * x; unsigned x1 = floor(origX); unsigned x2 = ceil(origX);
double xFrac = origX - (long)origX;
auto px1y1 = image.getPixel(x1, y1); auto px1y2 = image.getPixel(x1, y2); auto px2y1 = image.getPixel(x2, y1); auto px2y2 = image.getPixel(x2, y2);
auto h1 = px1y1 + xFrac * (px2y1 - px1y1); auto h2 = px1y2 + xFrac * (px2y2 - px1y2);
auto v = h1 + yFrac * (h2 - h1);
out->setPixel(x, y, v); } } });
return out; }~