2020

Meetrapport Efficiëntie Data Containers

Jeffrey de Waal & Diego Nijboer

Vision, Diederik Yamamoto-Roijers

4/4/2020

Inhoudsopgave

1.1. Doel

Het doel van onze meting is uitzoeken wat de efficiëntste manier voor het opslaan van data is. Er is keuze uit: de std::vector, std::array, std::map en een dubbele pointer

1.2. Hypothese

Wij denken dat de dubbele pointer uit onze test als snelste uit de bus komt. Dat denken wij omdat de dubbele pointer geen onnodige objecten aanmaakt en omdat dit geen onnodige abstractie vereist.

1.3. Werkwijze

Om uit te zoeken welke manier van opslaan het meest efficiënt is, hebben wij een programma geschreven dat iedere beschikbare container vult met 100000 RGB-objecten en deze vervolgens uitleest. We timen hoelang het duurt om alle containers uit te lezen. Aan de hand van deze data kunnen we de snelheid opmeten.

1.4. Resultaten

De volgende resultaten zijn een gemiddelde over 20 test runs.

Container type	Tijd
Double pointer	0.001 seconden
Std::array	0.007 seconden
Std::vector	0.146 seconden
std::map	2.427 seconden

1.5. Verwerking

We hebben de test resultaten afgelezen uit ons eigen testprogramma, in dit testprogramma hebben we de snelheden van de 4 verschillende containers gemeten. De code voor het testprogramma is in de bijlage onder aan dit document te vinden.

1.6. Conclusie

Als we de hier bovenstaande resultaten vergelijken, dan wordt het duidelijk dat de dubbele pointer tot wel 7x sneller is dan elke andere methode en hiermee dus als winnaar uit de test komt.

1.7. Evaluatie

Zoals wij eerder in de hypothese gesteld hebben, is de dubbele pointer de snelste manier om data op te slaan. Wij zullen dit container type dan ook gaan toepassen in onze code. We hebben hier de kans op meetfouten geprobeerd zo veel mogelijk te minimaliseren door de container met een hoog aantal objecten te vullen, dit zorgt ervoor dat de meettijden verder uit elkaar liggen.

1.8. Code voor de snelheidstest

```
class containers {
private:
  RGB black = RGB(255, 255, 255);
  RGB* blackPointer = &black;
  RGB returnObject;
  RGB* returnObjectPointer;
  clock_t beginTime;
  std::array<RGB, 100000> arrayContainer;
  std::map<int, RGB> mapContainer;
  std::vector<RGB> vectorContainer;
  RGB** doublePointerContainer;
  vector mapVector = vector(100, 100);
public:
  float execute_array() {
    std::cout << "std::array: ";
    beginTime = clock();
    for (int i = 0; i < 100000; i++) {
      arrayContainer[i] = black;
    }
    for (int i = 0; i < 100000; i++) {
      returnObject = arrayContainer[i];
    }
    return float(clock() - beginTime) / CLOCKS_PER_SEC;
  }
  float execute_double_pointer() {
    std::cout << "double pointer: ";
    beginTime = clock();
    for (int i = 0; i < 100000; i++) {
      doublePointerContainer = &blackPointer;
      doublePointerContainer++;
    }
    for (int i = 0; i < 100000; i++) {
      doublePointerContainer = &blackPointer;
      returnObjectPointer = *doublePointerContainer;
      doublePointerContainer++;
    }
    return float(clock() - beginTime) / CLOCKS_PER_SEC;
  }
  float execute_map() {
    std::cout << "std::map: ";
    beginTime = clock();
    for (int i = 0; i < 100000; i++) {
      mapContainer[i] = black;
    }
    for (int j = 0; j < 100000; j++) {
      returnObject = mapContainer[j];
    }
    return float(clock() - beginTime) / CLOCKS_PER_SEC;
    }
  float execute_vector() {
    std::cout << "std::vector: ";
    beginTime = clock();
    for (int i = 0; i < 100000; i++) {
      vectorContainer.push_back(black);
    } for (int i = 0; i < 100000; i++) {
      returnObject = vectorContainer.at(i);
    }
    return float(clock() - beginTime) / CLOCKS_PER_SEC;
  }
};
```