1-arrays

June 3, 2024

1 Choose your data structure

1.1 AoS (Array of Structs)

In the code example below, a “SAXPY” (`y = a*x+y`) calculation is done on a collection of `XY` elements.

[12]:
```
%%file tmp.xy.h

struct XY
 {
  double x, y {0.} ;
  void saxpy( double a )
   { y = a*x + y ; }
 } ;
```

```
Overwriting tmp.xy.h
```

[13]:
```
%%file tmp.aos-functions.h

#include <cstdlib> // for rand

template< typename Itr >
void randomize_x( Itr begin, Itr end )
 {
  for ( Itr itr = begin ; itr!=end ; ++itr )
   { itr->x = std::rand()/(RAND_MAX+1.)-0.5 ; }
 }

template< typename Itr >
void saxpy( Itr begin, Itr end, double a )
 {
  for ( Itr itr = begin ; itr!=end ; ++itr )
   { itr->saxpy(a) ; }
 }

template< typename Itr >
double accumulate_y( Itr begin, Itr end )
 {
```

```
  double res {0.} ;
  for ( Itr itr = begin ; itr!=end ; ++itr )
   { res += itr->y ; }
  return res ;
 }
```

Overwriting tmp.aos-functions.h

[20]:
```
%%file tmp.aos.cpp

#include "tmp.xy.h"
#include "tmp.aos-functions.h"
#include <cassert> // for assert
#include <cstdlib> // for atoi
#include <iostream>

int main( int argc, char * argv[] )
 {
  assert(argc==3) ;
  int size {atoi(argv[1])} ;
  int repeat {atoi(argv[2])} ;
  std::cout.precision(18) ;

  XY * collection {new XY[size]} ;
  auto begin {collection} ;
  auto end {begin+size} ;

  randomize_x(begin,end) ;
  while (repeat--)
    saxpy(begin,end,0.1) ;
  double res {accumulate_y(begin,end)/size} ;
  std::cout<<res<<std::endl ;

  delete [] collection ;
 }
```

Overwriting tmp.aos.cpp

[18]:
```
%%file tmp.aos.bash
echo

rm -f tmp.aos.exe tmp.aos.py
g++ -std=c++17 tmp.aos.cpp -o tmp.aos.exe
./tmp.aos.exe $*

echo "s = 0" >> tmp.aos.py
for i in 0 1 2 3 4 5 6 7 8 9
do \time -f "s += %U" -a -o ./tmp.aos.py ./tmp.aos.exe $* >> /dev/null
```

```
done
echo "print('(~ {:.3f} s)'.format(s/10.))" >> tmp.aos.py
python3 tmp.aos.py

echo
```

```
Overwriting tmp.aos.sh
```

[19]:
```
!bash -l tmp.aos.bash 1024 100000
```

```
67.5053500207703507
(~ 1.380 s)
```

The `main` function is currently using an old-fashioned C array, and the script does not set explicitly the GCC optimization option, which means it is using the default `-O0` (no compiler optimization).

You are asked to try this code, then investigate the alternative arrays `std::array`, `std::valarray`, `std::vector`, `std::list` and the alternative GCC compilation options `-O2` (usual optimisations) and `-O3` (aggressive optimizations, including automatic vectorization). Fill the results below, and try to explain the differences.

Array Option	-O0	-O2	-O3
Classic C array	0.	0.	0.
std::array	0.	0.	0.
std::valarray	0.	0.	0.
std::vector	0.	0.	0.
std::list	0.	0.	0.

1.2 SoA (Struct of Arrays)

Now let's try another approach: instead of creating a structure that groups together `x` andy and making it into an array (as it is naturally done on an object-oriented approach), let's try to make a global structure that contains an array of `x` on one hand, and an array of `y` on the other hand.

This is what the code skeleton below offers, again using C arrays and default -O0. Again, try alternative collections and compilation options. Fill the results table and explain.

[7]:
```
%%file tmp.soa.h

#include "tmp.xy.h"

class SoA
 {
  public :
    SoA( int size ) : m_size(size), m_xs(new double[size]), m_ys(new␣
 ↪double[size]) {}
    ~SoA() { delete [] m_xs ; delete [] m_ys ; }
```

```
    int size() { return m_size ; }
    XY operator()( int indice ) const
     { return { m_xs[indice], m_ys[indice] } ; }
    auto & xs() { return m_xs ; }
    auto & ys() { return m_ys ; }
    void saxpy( double a )
     {
      for ( int i=0 ; i<m_size ; ++i )
        m_ys[i] = a*m_xs[i] + m_ys[i] ;
     }
  private :
    int m_size ;
    double * m_xs ;
    double * m_ys ;
 } ;
```

Overwriting tmp.soa.h

[8]:
```
%%file tmp.soa-functions.h

#include "tmp.soa.h"
#include <cstdlib> // for rand

void randomize_x( SoA & collection )
 {
  for ( int i=0 ; i<collection.size() ; ++i )
   { collection.xs()[i] = std::rand()/(RAND_MAX+1.)-0.5 ; }
 }

double accumulate_y( SoA & collection )
 {
  double res {0.} ;
  for ( int i=0 ; i<collection.size() ; ++i )
   { res += collection.ys()[i] ; }
  return res ;
 }
```

Writing tmp.soa-functions.h

[9]:
```
%%file tmp.soa.cpp

#include "tmp.soa-functions.h"
#include <iostream>
#include <cassert> // for assert
#include <cstdlib> // for atoi

int main( int argc, char * argv[] )
 {
```

```
  assert(argc==3) ;
  int size {atoi(argv[1])} ;
  int repeat {atoi(argv[2])} ;

  SoA collection(size) ;
  randomize_x(collection) ;
  while (repeat--)
    collection.saxpy(0.1) ;
  double res = accumulate_y(collection)/size ;

  std::cout.precision(18) ;
  std::cout<<res<<std::endl ;
 }
```

```
Writing tmp.soa.cpp
```

[10]:
```
%%file tmp.soa.bash
echo

rm -f tmp.soa.exe tmp.soa.py
g++ -std=c++17 tmp.soa.cpp -o tmp.soa.exe
./tmp.soa.exe $*

echo "s = 0" >> tmp.soa.py
for i in 0 1 2 3 4 5 6 7 8 9
do \time -f "s += %U" -a -o ./tmp.soa.py ./tmp.soa.exe $* >> /dev/null
done
echo "print('({:.3f} s)'.format(s/10.))" >> tmp.soa.py
python3 tmp.soa.py

echo
```

```
Writing tmp.soa.sh
```

[11]:
```
!bash -l tmp.soa.bash 1024 100000
```

```
67.5053500207703507
(0.920 s)
```

To help in the analysis, GodBolt can be used, which allows to observe the dose of "inlining", or to look for the presence of vectorial instructions in assembly, such as `addpd` (Add Packed Doubles) or`mulpd` (Multiply Packed Double).

Array Option	-O0	-O2	-O3
Classic C array	0.	0.	0.
std::array	0.	0.	0.
std::valarray	0.	0.	0.

Array Option	-O0	-O2	-O3
std::vector	0.	0.	0.
std::list	0.	0.	0.