Table of Contents

10.3.2 Writing Autovectorizable Code

It works!: the recipe for the impatient user

Take your gcc compiler, also **gcc version 4.3.4**. Take this trivial code:

```
int main(){
const unsigned int ArraySize = 10000000;
float* a = new float[ArraySize];
float* b = new float[ArraySize];
float* c = new float[ArraySize];

for (unsigned int j = 0; j<200 ; j++) // some repetitions
   for ( unsigned int i = 0; i  <ArraySize; ++ i)
      c[i] = a[i] * b[i];
}
```

Compile it and run it:

```
g++ vectorisable_example.cpp -o vectorisable_example -O2
time ./vectorisable_example
```

Easy. Tell now the compiler to autovectorize it:

```
g++ vectorisable_example.cpp -o vectorisable_example_vect -O2 -ftree-vectorize
time ./vectorisable_example_vect
```

Faster, eh? Let's see in depth how and why.

Introduction

Starting in the late nineties, Intel integrated Single-Instruction-Multiple-Data (SIMD) machine instructions into their commodity CPU product line. These SIMD instructions allow to apply the same mathematical operation (multiply, add ...) to 2, 4 or even more data values at a time. The set of data values is also called "vector" (not to be confused with an algebraic vector). The theoretical performance gain is equal to the amount of vector units a CPU can hold.

Although the SIMD instruction sets have been around in every-day CPUs for quite some time, theses extended functionalities could only be accessed by programmers willing to interleave their high-level C or C++ source code with sections of quasi-assembler instructions. Modern compilers, like the GNU Compiler Collection (GCC), now include the ability to transform regular C or C++ source code into vector operations. This process is called **auto-vectorization**.

This page will provide an introduction to the necessary software tools and programming techniques and will furthermore give exemplary source code which take advantage of the auto-vectorize feature of GCC.

Hardware

To estimate the performance gains you can achieve with auto-vectorized code, you first need to understand the capabilities of the CPU you are compiling for and how many vectors it can process. Therefore, run the following command:

```
$ cat /proc/cpuinfo
...
     flags              : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
```

```
                    rep_good nopl xtopology nonstop_tsc aperfmperf pni pclmulqdq dtes64 mon
                    ida arat epb xsaveopt pln pts dts tpr_shadow vnmi flexpriority ept vpid
...
```

Among other CPU properties, you will find information about the SIMD capabilities (highlighted in bold). If a certain string is listed here, your CPU supports this feature:

Name	Register Size	Amount of floats	Amount of doubles
mmx (*)	64 bit	0	0
sse2	128 bit	4	2
ssse3	128 bit	4	2
sse4_1	128 bit	4	2
sse4_2	128 bit	4	2
avx	256 bit	8	4

* the first vector units on Intel processors could hold only two itegers.

All Intel and AMD 64-bit CPUs at least support the sse2 instruction set. The AVX instruction set has been introduced in 2011 with the Sandy Bridge architecture and is available on the latest generation of Mac Book Pro laptops.

The AVX instruction set can apply mathematical operations on four double precision floating point values simultaneously. Thus the maximum performance gain your application can achieve is times 4 compared to the version which is not using the vector units.

The steady evolution of the size of the vector units is a symptom of the new trend of the microarchitecture evolution. Ten years ago, the clock frequency was almost doubling on a yearly basis. Somehow a clock frequency limit has been reached and manufacturers opted for bigger and bigger vector units and an increasing number of cores per die.

Which code can be auto-vectorized ?

In order to distribute calculations to the vector units of the CPU, the compiler has to have a solid understanding of the dependencies and side effects of your source code. But foremost, the compiler must be able to detect sections of your source-code where the Single-Instruction-Multiple-Data paradigm can be applied. The simplest case is a loop which performs a calculation on an array:

```
for ( unsigned int i = 0; i <ArraySize; ++ i)
{
     c[i] = a[i] * b[i];
}
```

Try this example yourself using at least **gcc461** (available within the slc5_amd64_gcc461 scram architecture). The command line argument you have to use is `-ftree-vectorize`.

Note: Loops which profit the most from auto-vectorization are the ones containing mathematical operations. Loops which are merely iterating over a collection of objects won't profit and are not even auto-vectorizable in most cases.

A list of loop constructs which can be auto-vectorized can be found here:
http://gcc.gnu.org/projects/tree-ssa/vectorization.html

Once you compile your code with the option `-ftree-vectorizer-verbose=7` GCC will give you a detailed report of all loops in your program and whether they were auto-vectorized or not. The following report is the result of a successfully vectorized loop:

```
autovect.cpp:66: note: vect_model_load_cost: aligned.
autovect.cpp:66: note: vect_get_data_access_cost: inside_cost = 1, outside_cost = 0.
autovect.cpp:66: note: vect_model_load_cost: aligned.
autovect.cpp:66: note: vect_get_data_access_cost: inside_cost = 2, outside_cost = 0.
autovect.cpp:66: note: vect_model_store_cost: aligned.
autovect.cpp:66: note: vect_get_data_access_cost: inside_cost = 3, outside_cost = 0.
autovect.cpp:66: note: vect_model_load_cost: aligned.
autovect.cpp:66: note: vect_model_load_cost: inside_cost = 1, outside_cost = 0 .
autovect.cpp:66: note: vect_model_load_cost: aligned.
autovect.cpp:66: note: vect_model_load_cost: inside_cost = 1, outside_cost = 0 .
autovect.cpp:66: note: vect_model_simple_cost: inside_cost = 1, outside_cost = 0 .
autovect.cpp:66: note: vect_model_simple_cost: inside_cost = 1, outside_cost = 1 .
autovect.cpp:66: note: vect_model_store_cost: aligned.
autovect.cpp:66: note: vect_model_store_cost: inside_cost = 1, outside_cost = 0 .
autovect.cpp:66: note: Cost model analysis:
  Vector inside of loop cost: 5
  Vector outside of loop cost: 11
  Scalar iteration cost: 5
  Scalar outside cost: 0
  prologue iterations: 0
  epilogue iterations: 2
  Calculated minimum iters for profitability: 3

autovect.cpp:66: note:   Profitability threshold = 3

autovect.cpp:66: note: Profitability threshold is 3 loop iterations.
autovect.cpp:66: note: LOOP VECTORIZED.
```

If a loop could not be vectorized, GCC outputs the reason:

```
autovect.cpp:133: note: not vectorized: control flow in loop.
```

In this case, a call to `std::cout` introduced a control flow within the loop, which prevents the vectorization.

Getting GCC to auto-vectorize your code

A requirement to auto-vectorize source code with GCC is to use a recent version of the compiler. We suggest using at least **GCC 4.6.1**, but in general one can say "the newer, the better".

Compiler Flags

To turn on auto-vectorization use

```
-ftree-vectorize
```

This option is implicitly enabled if you compile with the optimization level `-O3` or more.

To get a report which loops have been auto-vectorized and which failed and why, you can increase the verbosity of the vectorization using

```
-ftree-vectorizer-verbose=7
```

On how to read this output, have a look at the *Real-World code examples* section below.

Some loop constructs (e.g. reduction of float values) can only be vectorized if the compiler is allowed to change the order of mathematical operations. To allow this, use the flag

```
-ffast-math
```

This option is also enabled if you use the optimization level `-Ofast`. Be aware, that the fast-math option slightly modifies the error behavior in floating point operations. For details, please see http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Optimize-Options.html

Furthermore, you can explicitly state, which SIMD instruction set the compiler should use when generating the binary file. When compiling for the `x86-64` architecture, GCC will use the SSE2 instruction set by default. If you want to enable the AVX instruction set, use the compiler flag:

```
-mavx
```

Be aware, that the machine you want to run the binary on must support the AVX instruction set. You can also let the compiler decide which instruction set is supported by the machine you are compiling on by using the compiler flag

```
-march=native
```

If you want to use C++11 features, like lambda expressions, be sure to enable the new standard

```
-std=gnu++0x
```

Best-Practices to enable auto-vectorization

Use Structure-of-Arrays

Not only needs the compiler be able to analyze your loop iterations, but also the patterns in which the memory is accessed. Complex classes with nested arrays are therefore hard or impossible to auto-vectorize by the compiler. Use simple c-arrays or `std::arrays` (introduced in C++11) to hold your data and allow the compiler to access your data in a continuous manner. This will also enable your program to take advantage of the various caches the CPU has.

If you need variable-sized arrays, use `std::vector` and extract a pointer to the first element to access the elements during a loop:

```
std::vector
```

An complete example on Structure-Of-Arrays can be found in the "Comprehensive Code Example" section below

Limit branching

Limit the branching within the for-loop to a minimum. GCC is able to translate some if statements to vectorized code, but only to a limited extend. When doing so, all possible branches are evaluated and the values of the branch not taken are discarded.

Isolate functional parts

If you have complex calculations within the for-loop you want GCC to vectorize, consider splitting them into more simpler fragments, called *kernels*, by using the C++11 feature of lambda expressions. An introduction to this new C++ feature can be found here: http://en.wikipedia.org/wiki/Anonymous_function#C.2B.2B

Here is an example of a lambda function defined to compute and return the square of a double variable.

Definition:

```
auto kernel_square =
```

```
        [] // capture nothing
        ( double const& a) // take 1 parameter by reference
        ->double // lambda function returns the square
{
        return ( a * a );
};
```

And the invocation of the lambda function within a loop:

```
for ( unsigned int i = 0; i <size; ++ i)
{
        ptr_square[i] = kernel_square( ptr_x[i] ) + kernel_square( ptr_y[i] ) + kernel_square( pt
}
```

Note that this code will be auto-vectorized by GCC. The call to the lambda function will not result in the overhead associated with regular functions, as the code will be fully inlined.

Another, more comprehensive example includes a for-loop within the lambda function. This loop is also auto-vectorized by GCC:

```
// Defining a compute kernel to encapsulate a specific computation
auto kernel_multiply =
        [ &cFixedMultiply ] // capture the constant by reference of the scope of the lambda expre
        ( DataArray const& a, DataArray const& b, DataArray & res ) // take 3 parameters by refer
        ->void // lambda function returns void
{
        // simple loop vectorized
        for( unsigned int i<a.size(); ++ i)
        {
                res[i] = a[i] * b[i];
        }
};
```

No calls to external functions

Calls to external functions, like `exp()`, `log()` etc, break the vectorization of a loop. Therefore, you have to decide if the mathematical operations within your loop are complex enough to justify a split of the loop. This means, in the first loop you calculate the parameter for the `exp()` call and store this result in a temporary array. If done right, GCC will auto-vectorize this loop and you can profit from the speedup. The second loop will simply perform the calls to `exp()` and store the result.

If the function you want to call is controlled by you, attempt to formulate this function as a C++11 lambda-expression. An introduction to this new C++ feature can be found here: http://en.wikipedia.org/wiki/Anonymous_function#C.2B.2B

Use plain-old integer counters in loops

Use plain-old integers counters to construct the for-loop and not the famed std::iterators, as these make it hard for GCC to analyze memory access and the properties of the the loop, like iteration count.

```
for (vector
```

becomes

```
for (unsigned int i = 0; i <y.size(); ++i)
{
    y[i] += 23;
}
```

Comprehensive Code Example

The following code gives a few examples on how to write auto-vectorizable C++ code with GCC. Copy and paste the source code and compile it with the command

```
g++ -Ofast -ftree-vectorizer-verbose=7 -march=native -std=c++11 -o autovect autovect.cpp
```

Be sure to use at least GCC 4.7

```
/*

   Some examplary loops for GCC Auto-Vectorization

   by Thomas Hauth ( Thomas.Hauth@cern.ch )

   Compile with ( use at least gcc 4.7 ):
   g++ -Ofast -ftree-vectorizer-verbose=7 -march=native -std=c++11 -o autovect autovect.cpp

*/

#include <math.h>

#include <string>
#include <iostream>
#include <array>
#include <vector>

// Sturcture-Of-Array to hold coordinates
struct Vector3
{
   std::vector<double> x;
   std::vector<double> y;
   std::vector<double> z;

   // final result of the distance calcualtion
   std::vector<double> distance;

   void add( double _x, double _y, double _z )
   {
      x.push_back( _x );
      y.push_back( _y );
      z.push_back( _z );
      distance.push_back( 0.0f );
   }

};


int main()
{

// Fixed Size Arrays

   typedef std::array<double, 10> DataArray;

   DataArray vect_a = { 0,1,2,3,4,5,6,7,8,9 };
   DataArray vect_b = {0.5,1,2,3,4,5,6,7,8,9 };
   DataArray vect_res_plain = { 0,0,0,0,0,0,0,0,0,0};
   DataArray vect_res_lambda = { 0,0,0,0,0,0,0,0,0,0};

   constexpr double cFixedMultiply = 23.5f;

   // simple loop vectorized
   // -- auto-vectorized --
```

```
   for( unsigned int i = 0; i < vect_a.size(); ++ i)
   {
      vect_res_plain[i] = vect_a[i]  + vect_b[i];
   }

   // Defining a compute kernel to encapsulate a specific computation
   auto kernel_multiply =
      [ &cFixedMultiply ] // capture the constant by reference of the scope of the lambda express
      ( DataArray const& a, DataArray const& b, DataArray & res ) // take 3 parameters by referen
      ->void // lambda function returns void
   {
      // simple loop vectorized
      // -- auto-vectorized --
      for( unsigned int i = 0; i < a.size(); ++ i)
      {
         res[i] = a[i] * b[i] * cFixedMultiply;
      }
   };

   // call the lambda function
   // this call is autovectorized
   kernel_multiply ( vect_a, vect_b, vect_res_lambda );


   // This kernel will be called multiple times and performs the quadrature
   auto kernel_square =
      [] // capture nothing
      ( double const& a) // take 1 parameter by reference
      ->double // lambda function returns the square
   {
      return ( a * a );
   };

   // create struct and fill with dummy values
   Vector3 v3;
   for ( unsigned int i = 0; i < 50 ; ++ i)
   {
      v3.add( i * 1.1, i * 2.2,  i* 3.3 );
   }

   // store the size in a local variable. This is needed to GCG
   // can estimate the loop iterations
   const unsigned int size = v3.x.size();

   // -- auto-vectorized --
   for ( unsigned int i = 0; i < size; ++ i)
   {
      v3.distance[i] = sqrt( kernel_square( v3.x[i] ) + kernel_square( v3.y[i] ) + kernel_square(
   }

   // output the result, so GCC won't optimize the calculations away
   std::cout << std::endl << "Computation on std::array" << std::endl;
   for( unsigned int i = 0; i < vect_a.size(); ++ i)
   {
      std::cout << vect_res_plain[i] << std::endl;
      std::cout << vect_res_lambda[i] << std::endl;
   }

   std::cout << std::endl << "Computation on Structure-of-Array with variable sized std::vectors"
   for( unsigned int i = 0; i < v3.x.size(); ++ i)
   {
      std::cout << "sqrt( " << v3.x[i] << "^2 + " << v3.y[i] << "^2 + " << v3.z[i] << "^2 ) = "
                << v3.distance[i] << std::endl;
   }

   return 0;
}
```

Autovectorisation in real applications

Here you can find a non comprehensive list of products taking advantage of the autovectorisation capabilities of the GCC compiler:

- The VDT mathematical library

This topic: CMSPublic > WorkBookWritingAutovectorizableCode
Topic revision: r12 - 2012-11-30 - ThomasHauth