C++ Needs Language Support For Vectorization
ISO/IEC JTC1 SC22 WG21 N3774

Axel Naumann (axel@cern.ch)
Sandro Wenzel (sandro.wenzel@cern.ch)

2013-09-19

Contents

Introduction

In this paper we argue that vectorization is a very different concept to parallelization and needs to be supported explicitly by the language. The lack of C++ support for explicit vectorization costs factors (2 to 4 in real code) of performance on current commodity hardware. We demonstrate why we believe vectorization is a much needed language feature. The arguments presented in this paper are based on, and accompanied by, performance measurements of code used at CERN.

Vectorization

Operations can be reshuffled and combined into vector operations, replacing for instance a loop by a single operation on a consecutive set of data. As an example, the non-vectorized code

```
int in0[4] = {0};
int in1[4] = {0, 1, 2, 3};
int out[4];
for (int i = 0; i < 4; ++i)
  out[i] = in0[i] + in1[i];
```

could be transformed into vectorized (pseudo-) code

```
int in0[4];
vector_init(in0, 0);
int in1[4] = {0, 1, 2, 3};
int out[4];
vector_add(out, in0, in1);
```

where `vector_xyz` are hypothetical, architecture-specific instructions (*vector intrinsics*) operating on a set of data.

Vectorization and loop semantics

Vectorized operations do not guarantee the same order of iteration as traditional loops. If the original loop looked like this:

```
int in0[4] = {0, 1, 2, 3};
int out[5] = {0};
for (int i = 0; i < 4; ++i)
  out[i+1] = in0[i] + out[i];
```

then the vectorized version of this example will generally not produce the same (nor even reproducible) result, because the sequencing of reading from and storing to `out` is likely going to be different than in a traditional loop. This is a fundamental semantic change of loops, likely violating 1.9 [intro.execution].

Vectorization versus threading

High Energy Physics, with its embarrassingly parallel problems, has used multi-process parallelism for decades, both for multi-node ("cluster") and single node parallelism. Additionally we selectively employ multithreading to cut down the initialization time and to reduce memory usage (shared memory of threads versus separate process memory). These two concurrency mechanisms are used to schedule tasks on processors; they do not increase the software's efficiency on a single processor. Vectorization, on the other hand, does improve the efficiency (and in particular the throughput) of our software, by using the CPU's vector resources.

Number crunching software, as developed by institutions like CERN, control the parallelization in a very direct way: process parallelism is used in combination with thread parallelization; having the compiler guess what should be good for us would at least be a wasted effort, if not counterproductive.

Also for developers, vector loops are conceptually *very* different from thread parallelization; see for instance (1) contrasting these two. While thread parallelization encourages keeping data in disjoint memory regions and is closer in spirit to multi-process parallelization (with issues such as synchronization, starving, pooling etc), vectorization is a data-centric concept.

Thus, vectorization is orthogonal to parallelization and needs to be handled separately. A picture might contrast these two concepts even more clearly: large trucks are used to remove excavation material in mines. We currently use only a fraction of the size of the trucks; vectorization would allow us to use them to a larger extent. Threading is the concept of using multiple trucks – which is good but different, and not a good reason for running mostly empty trucks.

Types of vectorization

In principle, we can distinguish three different kinds of vectorization:

Auto-Vectorization This is the only type of compiler-generated vectorization available to standard compliant code. Because auto-vectorized and non-vectorized code must yield the same results, the compiler must guarantee that aliasing and order of operations with respect to value dependency cannot interfere with the result. Vector engines of most current hardware require a certain alignment of the groups of input data, that must be guaranteed by the compiler. Any non-vectorizable operation will prevent the compiler from auto-vectorizing code. All of this explains why only very few tight and simple loops with local-only data can possibly be auto-vectorized.

Vectorization through Vector Wrapping Intrinsics An alternative to auto-vectorization is explicit vector-oriented programming. Several libraries exist or are suggested that hide the architecture specific vector intrinsics and expose them through a common interface; see for instance (2–4). Use of intrinsics can lead to significant performance boosts, at the cost of a fundamental change in code layout. In addition, they prevent certain compiler optimizations in code calling into these libraries; memory management and vectorization is defined by the library, which has no knowledge of the code using it.

The fact that the libraries are targeted to a specific set of architectures (which inevitably leaves traces in their interfaces, for instance for alignment) makes them a good *interim* option.

Vectorization Through Code Specifications A third approach uses language extensions. Code can be annotated to guarantee to the compiler that the prerequisites for vectorizing it are met – even though (just as in the previous approach) the compiler cannot determine the correctness of this guarantee and thus would usually not auto-vectorize. Typical examples of the technique are vectorization pragmas (5, 6), OpenMP 4.0 (7) and Intel Cilk Plus (8); a proposal to the committee (9) is similar to the latter.

Our favored approach, annotated loops as in (6, 9, 10), is one of the key language extensions employing this technique. This programming model remains close to that of current C++ making it more accessible. As vectorization through annotations is controlled by the compiler, compiler optimizations can still be leveraged. We have seen cases where the compiler decided against the use of vectorization in annotated loops; a cross-check with explicit vectorization showed that this was indeed the correct optimization.

Compiler-generated vectorization can easily and naturally target the vectorization code for a specific architecture, the same way it currently targets

non-vector code generation. With the emergence of CPU / GPU shared memory regions, targeting the vector code to GPUs will be an obvious next step that in fact has already started (7).

Performance comparisons of different vectorization mechanisms

Algorithm used for performance measurements

For the purpose of this paper, we give an example from the field of detector simulation, which is one of the major consumers of CPU ressources at CERN. Already targeting multi-threading for some time (11), current efforts focus on vectorization support within the simulation software (12) to increase its floating point performance. A very important part of such simulations are simple geometrical calculations, determining for instance whether a particle is inside a particular shape or calculating its distance to the shape's surface. Similar operations are often done in the gaming industry.

As an illustrative example for such calculations we present the following small function that determines if a particle of coordinates `point[0] point[1] point[2]` is located inside a rectangular box of lateral width `boxsize[..]` whose origin is located at `origin`.

```
bool contains(double const* point)
{
   bool inside[3];
   for(int dir=0;dir < 3;dir++){
        inside[dir] = std::abs(point[dir]-origin[dir]) < boxsize[dir];
   }
   return inside[0] && inside[1] && inside[2];
}
```

Auto-vectorization

In future vector-oriented simulations we target such calculations to be executed for several particles per call instead of just one, by introducing a similar function with a slightly modified signature:

```
void contains_v(double const* __restrict__ points,
                bool * __restrict__ isin, int np)
{
   for(int k=0; k<np; k++)
     isin[k] = contains( &points[3*k] );
}
```

Ideally, such code additions should be enough to benefit from CPU vector capabilities. Unfortunately, this simple code never auto-vectorizes and no gain in efficiency is obtained although we know that vectorization is possible.

After a series of refactorings (manual loop unrolling, code inlining, conversion of memory access) and compiler hints (`restrict`) we arrive at code that the compiler agrees to auto-vectorize. However, this approach is very expensive on the developer's side, already for this tiny example, and offers essentially no guarantee of success.

Vectorization using intrinsics

Our currently favored approach (given the state of standard C++, i.e. the lack of language support for vectorization) therefore consists of the use of vector intrinsics. We pay a price for this: we have to manually change the loop structure and to manage all local variables in a vector way. The following snippet shows Vc (3) (see also (4)) code for the above example; similar code could be written using the Intel Cilk Plus array notation (8).

```
void contains_v_Vc( double const* points, bool * isin, int np )
{
  // Architecture dependent "gather" indexes:
  unsigned int const i[4]={0,3,6,9};
  Vc::uint_v const indices(i);

  for(int k = 0; k < np; k += Vc::double_v::Size)
  {
    Vc::double_m mask[3];
    for(int dir=0;dir < 3;dir++)
    {
      Vc::double_v x;
      // Gathers a certain number of coordinates
      // into Vc vector x
      x.gather(&points[3*k+dir], indices);
      x = Vc::abs(x - origin[dir]);
      mask[dir] = (x < boxsize[0]);
    }

    Vc::double_m particleinside;
    particleinside = mask[0] && mask[1] && mask[2];
    for(int j=0;j<Vc::double_v::Size;++j) {
      isin[k+j] = particleinside[j];
    }
  }
}
```

For the Vc example above, using the SSE4 instruction set with GCC version 4.7.2 and compilation flags `g++ -funroll-loops -Ofast` on this code, a speedup factor of 1.93 was seen compared to the non-vectorized version. This factor is close to the size of the SSE4 vector register, i.e. to the number of input data elements that a single vector operation can act on.

We see similar speedup factors in much more complex and complete algorithms, making a dramatic impact in overall performance of our simulation software.

Language-enabled vectorization

Loop syntax and semantics as proposed by (9) enable us to write this code in a much more compact and readable way. We have measured similar (higher or lower) performance gains as using Vc. Most currently available implementations of loop language extensions only offer `#pragma` statements instead of proper keywords. A vector for loop using Intel Cilk Plus (8) could be expressed as:

```
void contains_v(double const* points, bool * isin, int np)
{
#pragma simd
  for(int k=0; k<np; k++)
    isin[k]=contains( &points[3*k] );
}
```

For the vectorization to succeed, this requires `contains()` to be declared as *elemental* function (see also (9)).

This loop is impressively close to traditional C++ code. It encapsulates the vector complexity that is much more exposed in the approach using explicit intrinsics. Nonetheless, the algorithm requires careful design to benefit from vectorization. But once spelled out, it has a very readable form – and unlike current C++ *does* enable vectorization and optimization through compilers.

In summary, use of vectorization drastically increases the performance of number crunching operations. This is not only visible in example code: it can scale to the bulk of algorithms. Using language extensions (for instance that proposed in (9)), vectorized algorithms can be expressed in a way that is very compact and at the same time very similar to current C++.

Motivation for vectorization in C++

Vectorization support should be offered, and it should be offered as a language feature. The following paragraphs explain our reasoning:

Use past, current and future hardware efficiently, with C++

Non-vectorized code uses about 25% of the power of today's commodity hardware – and even less of tomorrow's. It runs instructions on each input data, instead of running an instruction on a vector of input data in the same amount of cycles. As vectorized code is in general more compact (because of the operations on vectors) it usually also improves cache performance.

Vectorization should not be hidden in a library

Vectorized library components

Having a vectorized set of algorithms as suggested by (13) is not a generic answer to the underlying problem: currently C++ does not allow to communicate to the compiler that a loop's requirements for vectorization are fulfilled. For most number crunching applications, vectorized standard library algorithms only help in a tiny fraction of all problems. Changing code to make use of them will often cause additional inefficiencies. The effect of modularization (for instance the call to an elemental function like `contains()` in the above example) on the ability of the standard algorithms to vectorize are not clear at all. It is generally impossible to construct real world algorithms from a set of constituent elements of a hypothetical vectorized standard library.

Vectorizing library components

Focusing on a vectorized `std::for_each` (as for instance suggested by (13)), the differences between a language-based vector loop and a vectorizing library loop seem purely syntactic, with the vectorized `std::for_each` allowing to move vectorization into the library. But this neglects additional information that must be provided to the compiler for vectorization of realistic cases, such as inter-loop dependencies (for instance different variable types like linear versus fixed for all iterations), and on the other hand missing guarantees on the sequencing of the iterations: like regular for loops, vectorized for loops have sequencing guarantees – but they are different. And just as with regular for loops, developers will have to rely on these sequencing guarantees for real world algorithms. Unlike in the parallel for loop case, the sequencing is non-linear. We find this a change too dramatic to be hidden in a library function.

Even if means were provided to convey additional information to a vectorized `std::for_each`, the notion of elemental function will have to be added to enable modularity. And this is a language feature. Furthermore, while `std::for_each` looks like a library-only approach it requires compiler support: only the compiler can convey the information needed from the iterations' body to the vectorization

engine. As an example, at the point of invocation of `std::for_each` the body of the function argument to `std::for_each` must be available to the compiler to be able to vectorize it. Thus, much of the advantage of a library solution is purely perceptional.

Library components wrapping vector intrinsics

Intrinsics-wrapping vectorization libraries on the other hand suffer from (at least implicit) target architecture dependencies, valarray-style changes to algorithms and the fact that compilers are seemingly unable to optimize across intrinsic calls. Just as we prefer writing C++ over assembler code we should be able to rely on compilers to generate optimal machine code.

Prevent escape to non-standard extensions

To leverage the performance of current hardware, people resort to extensions of C++. Those have the usual issue of standard extensions: they are not standard, and where they are (OpenMP might be seen as such) not all C++ compiler vendors implement them – which makes them nonstandard *for C++ users*.

Infrequently used counterarguments

Just for completeness, we have collected a few arguments against vectorization support, and our thoughts about those.

- **Explicit vectorization annotation is not needed; auto-vectorization will get there eventually.**

  All compiler vendors we have talked to disagree.

- **Explicit vectorization annotation is dangerous; it cannot be correct, and if it is correct now it might become incorrect after a code change.**

  The same holds true with thread safety: the compiler cannot guarantee thread safety, it has to trust the developer. Just because you have a powerful tool doesn't mean everyone should use it. In fact the problem is really that currently *nobody* can write code leveraging vector hardware (almost all current commodity chips) in standard C++.

- **A call to any function will cause havoc.**

  The proposal (9) (that we enthusiastically support) does not suffer from this. It does not require a "contagious" annotation run through a call graph. It merely uses calls to non-vectorizable functions as "synchronization" points (in the parallel language).

- **A call to any (for instance math) function will remove the advantage of vectorized code.**

  Many libraries exist that re-implement mathematical functions in a vectorizable way (14, 15); we have our own (16).

Conclusion

We have shown that C++ users currently pay a hefty price for using standard C++. Many extensions exist to leverage vector units in a meaningful way. C++ should react; a language extension is needed.

The proposal (9) goes into the right direction. It would allow us to express our vectorized code as standard compliant code while maintaining its efficiency. Alternative proposals, for instance on an array notation in the spirit of Cilk Plus (8), would be interesting to see and discuss. Whatever solution chosen, vectorization support will have a big impact for all (floating point and integer) number crunching communities.

References

1. GEVA, Robert. *On the difference between parallel loops and vector loops.* N3735

2. ESTERIE, Pierre, GAUNARD, Mathias and FALCOU, Joel. *A Proposal to add Single Instruction Multiple Data Computation to the Standard Library.* N3561

3. KRETZ, Matthias and LINDENSTRUTH, Volker. Vc: A C++ library for explicit vectorization. *Softw: Pract. Exper.* 2012. No. 42.

4. KRETZ, Matthias. *SIMD Vector Types.* N3759

5. MICROSOFT. *Visual Studio 2012 C/C++ Preprocessor Reference: pragma loop* [online]. Available from: http://msdn.microsoft.com/en-us/library/hh923901.aspx

6. INTEL. *Intel(R) C++ Compiler 12.1 User and Reference Guides: pragma ivdep* [online]. Available from: http://software.intel.com/sites/products/documentation/studio/composer/en-us/2011Update/compiler_c/cref_cls/common/cppref_pragma_ivdep.htm

7. OPENMP. *The OpenMP API specification for parallel programming* [online]. Available from: http://openmp.org/wp/openmp-specifications/

8. INTEL. *Cilk Plus* [online]. Available from: https://www.cilkplus.org/

9. GEVA, Robert. *Vector Programming; A proposal for WG21.* N3734

10. TITLE, Analog Devices. [online]. Available from: http://www.analog.com/en/dsp-software/vdsp-bf-sh-ts/sw.html

11. DONG, Xin, COOPERMAN, Gene, APOSTOLAKIS, John, JARP, Sverre, NOWAK, Andrzej, ASAI, Makoto and BRANDT, Daniel. Creating and Improving Multi-Threaded Geant4. *J. Phys.: Conf. Ser.* 2012. No. 396.

12. APOSTOLAKIS, John, BRUN, Rene, CARMINATI, Federico and GHEATA, Andrei. Rethinking particle transport in the many-core era towards GEANT 5. *J. Phys.: Conf. Ser.* 2012. No. 396.

13. HOBEROCK, Jared et al. *A Parallel Algorithms Library.* N3554

14. INTEL. *Short Vector Math Library (SVML) Functions* [online]. Available from: http://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-7580634E-88D9-4B67-94ED-8472CCC5AF68.htm

15. AMD. *LibM* [online]. Available from: http://developer.amd.com/tools-and-sdks/cpu-development/libm/

16. PIPARO, Danilo, INNOCENTE, Vincenzo and HAUTH, Thomas. *The VDT Mathematical Library* [online]. Available from: https://svnweb.cern.ch/trac/vdt