Scout: A Source-to-Source Transformator for SIMD-Optimizations

Olaf Krzikalla, Kim Feldhoff,
Ralph Müller-Pfefferkorn, and Wolfgang E. Nagel

Technische Universität, Dresden, Germany
{olaf.krzikalla,kim.feldhoff,ralph.mueller-pfefferkorn,
wolfgang.nagel}@tu-dresden.de

Abstract. We present Scout, a configurable source-to-source transformation tool designed to automatically vectorize C source code. Scout provides the means to vectorize loops using SIMD instructions at source level. Our main focus during the development of Scout is a maximum flexibility of the tool in two ways: being capable of vectorizing a wide range of loop constructs and being capable of targeting various modern SIMD architectures. Scout supports several SIMD instructions sets like SSE or AVX and is easily extensible to upcoming ones.

In the second part of the paper we present results of applying Scout's vectorizing capabilities to CFD production codes of the German Aerospace Center. The complex loops used in these codes often inhibit the automatic vectorization of usual C compilers. In contrast, Scout is able to vectorize most of these loops. We measured the resulting speedup for SSE and AVX platforms.

1 Introduction

Most modern CPUs provide SIMD units in order to support data-level parallelism. One important method of using that kind of parallelism is the vectorization of loops. However, programming using SIMD instructions is not a simple task. SIMD instructions are assembly-like low-level intrinsics and often steps like finalization computations after a vectorized loop become necessary. Thus tools are needed in order to efficiently exploit the data-level parallelism provided by modern CPUs.

In the context of the HI-CFD project [4] we needed a mean to comfortably vectorize loops written in C. We are going to target various HPC platforms with different instruction sets and different available compilers.

2 Related Tools

Naturally, a vectorization tool is best built in a compiler. Indeed, all current C compilers provide auto-vectorization units. But a compiler must reason about the correctness of the vectorized program automatically. This reasoning can be done by an extensive dependency and aliasing analysis and a lot of approaches

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part II, LNCS 7156, pp. 137–145, 2012.

are available to vectorize various forms of codes, especially loops [7]. However in practice it is not possible to always reason about the absence of dependencies (e.g. in a loop with indirect indexing). Thus means are needed in order to provide meta information about a particular piece of code. For instance the Intel compiler allows a programmer to augment loop statements with pragmas to designate the absence of inner-loop dependencies.

We have tested some compilers with respect to their auto-vectorization capabilities. For some loops in our codes the available means to provide meta information were insufficient (see Sect. 3.3). Sometimes subtle issues arose around compiler-generated vectorization. For instance in one case a compiler suddenly rejected the vectorization of a particular loop just when we changed the type of the loop index variable from `unsigned int` to `signed int`. A compiler expert can often reason about such subtleties and can even dig in the documentation for a solution. But an application programmer normally concentrates on the algorithms and cannot put too much effort in the peculiarities of each used compiler. The vectorization of certain (often more complex) loops was rejected by all compilers regardless of inserted pragmas, given command-line options aso.

We have checked other tools specifically targeting loop vectorization. In [6] a retargetable back-end extension of a compiler generation system is described. Being retargetable is an interesting property (see also Sect. 3.2) but for our project it did not come into consideration due to its tight coupling to a particular compiler system. *SWARP* [9] seems to depend solely on a dependency analysis – something we could not rely on.

3 The Vectorizing Source-to-Source Transformator Scout

We decided to develop a new tool in order to comfortably exploit the parallel SIMD units. The tool shall transform C source code. The output is also C source code, but with vectorized loops augmented by SIMD intrinsics. The respective SIMD instruction set is configurable. Thus the tool is usable as an universal vectorizer and it is aimed to become an industrial-strength vectorizing preprocessor. We have called this vectorization tool *Scout*.

Scout exposes a command line interface as well as a graphical user interfaces. Internally it uses the *clang* parser [1] to transform C source code to an abstract syntax tree (AST). The vectorization and other optimizations are then performed on that AST. Eventually the transformed AST is rewritten to C code. Scout is published under an Open Source license and available via `http://scout.zih.tu-dresden.de`

We have opted for a strict semi-automatic vectorization. That is, as with compilers, the programmer has to annotate the loops to be vectorized with `#pragma` directives. The directive `#pragma scout loop vectorize` in front of a `for` statement triggers the vectorization of that loop. Before the actual vectorization starts, the loop body is simplified by function inlining, loop unswitching (moving loop-invariant conditionals inside a loop outside of it [3]) and unrolling of inner loops whereever possible. The resulting loop body is then vectorized using the unroll-and-jam technique.

3.1 Unroll-and-Jam

Various approaches to vectorize loops exist. Traditional loop-based vectorization transforms a loop so, that every statement processes a possible variable-length vector [5]. With the advent of the so-called multimedia extensions in commodity processors the *unroll-and-jam* approach became more important [8]. In [7] this approach is descibed mainly as a mean to resolve inner-loop dependencies. However, we use this approach in a more general way. First, we partially unroll each statement in the loop according to the vector size. Then we test whether the unrolled statements can be merged to a vectorized statement. Unvectorizeable statements (e.g. if-statements including their bodies) remain unrolled. Only their memory references to vectorized variables are accordingly adjusted. All other statements are vectorized by decomposing them to vectorizeable expressions. Scout allows the user to vectorize arbitrarily complex expressions (see Sect. 3.2).

A nice consequence of using the unroll-and-jam approach is the possibility to vectorize different data types (e.g. `float` and `double`) in one loop simultaneously. The vector sizes of vectorized data types may differ, but the largest vector size has to be a multiple of all other used vector sizes. The loop is then unrolled according to that largest vector size and vectorizeable statements of other data types are then only partially merged together and remain partially unrolled.

Listing 1 demonstrates the vectorization of different data types for a SSE platform. The vector size for `float` is 4 and for `double` it is 2. Hence the loop is unrolled four times. Then all operations for `float` values can be merged together (in the example only the load/store operations). In contrast only two unrolled consecutive operations for `double` values (one load and the division) are merged to a vectorized operation leaving the `double` operations partially unrolled. Vectorized conversion operations are generated automatically whenever needed.

```
float a[100]; double b[100];
double x;
#pragma scout loop vectorize
for (int i=0; i<100; ++i) {
  x    = a[i];
  x    = x / b[i];
  a[i] = x;
}
```

```
float a[100]; double b[100];
__m128 av;
__m128d xv1, xv2, bv1, bv2;
for (int i=0; i<100; i+=4) {
  av  = _mm_loadu_ps(a + i);
  xv1 = _mm_cvtps_pd(av);
  xv2 = _mm_cvtps_pd(
          _mm_movehl_ps(av, av));
  bv1 = _mm_loadu_pd(b + i);
  bv2 = _mm_loadu_pd(b + i + 2);
  xv1 = _mm_div_pd(xv1, dv1);
  xv2 = _mm_div_pd(xv2, dv2);
  av  = _mm_movelh_ps(
          _mm_cvtpd_ps(xv1),
          _mm_cvtpd_ps(xv2));
  _mm_storeu_ps(a + i, av);
}
```

Listing 1. Mixing types in vectorization

3.2 Configuring Scout

A central requirement is the configurability of Scout with respect to existing as well as upcoming SIMD architectures. This aspect is controlled by supplying a configuration file to Scout which describes the properties of the target SIMD platform. A configuration file is written in pure C++. Actually C++ is not designed as a configuration language and thus we had to stretch the semantics. However the choice of C++ has a lot of advantages: it is not necessary to learn yet another configuration syntax, it is possible to use the usual preprocessing means (conditional compilation, includes) in the configuration thus alleviating the maintenance costs, and the AST of the configuration file can be generated and processed by clang. The actual intrinsics are wrapped up in string literals making the configuration valid C++ even if the headers for the actual target SIMD platform are not available on the translation machine. Listing 2 shows an excerpt of a configuration file for the data type `float` targeting the SSE architecture.

```
namespace scout {

template<class, unsigned> struct config;

template<>
struct config<float, 4> {
  typedef __m128 type;    // target SIMD type
  enum { align = 16 };    // alignment requirement

  static void store_aligned(float*, type) { // function name
    "_mm_store_ps(%1%, %2%)";               // predefined by Scout
  }

  static float add(float a, float b) { // expression mapping
    a + b;                             // statement is an expression
    "_mm_add_ps(%1%, %2%)";
  }

  static float condition_lt(float a, float b, float c, float d) {
    a < b ? c : d;
    "_mm_blendv_ps(%3%, %4%, _mm_cmplt_ps(%1%, %2%))";
  }

  static float sqrt(float) { // function mapping
    float sqrtf(float);      // statement is a function declaration
    "_mm_sqrt_ps(%1%)";
  }
}

} // namespace scout
```

Listing 2. Scout configuration for a typical SIMD architecture

For each supported data type the configuration provides a specialized class template named `config` placed in the namespace `scout`. The first template

parameter denotes the underlying base type of the particular vector instruction set. The second integral template parameter denotes the vector size of that set. A set of predefined type names, value names and static member functions are expected as class members of the specialization.

There are two general kinds of static member functions. If the function name is predefined by Scout, then the function body consists of only one statement – the string literal denoting the intrinsic. Load and store operations are defined in this way.

If the function name of the static member functions is not predefined, then the string literal in the function body is preceded by an arbitrary number of expressions and/or function declarations. In that case expressions and function calls in the original source code are matched against these configuration expressions and functions and are vectorized according to the string literal if they fit. This option adds great flexibility to Scout. Indeed it is not only possible to use various instruction sets in their atomic shape but also combine them to more complex or idiomatic expressions a priori.

Listing 3 demonstrates the vectorization capabilites of Scout by using the `condition_lt` and `sqrt` functions of Listing 2.

```
float a[100], b[100];
float x;
#pragma scout loop vectorize
for (int i=0; i<100; ++i) {
  x = a[i] < 0 ? b[i] : a[i];
  a[i] = sqrtf(x);
}
```

```
float a[100], b[100];
__m128 a_v, b_v, x_v, c0_v;
c0_v = _mm_set1_ps(0.0);
for (int i=0; i<100; i+=4) {
  a_v = _mm_loadu_ps(a + i);
  b_v = _mm_loadu_ps(b + i);
  x_v = _mm_blendv_ps(b_v, a_v,
          _mm_cmplt_ps(a_v, c0_v));
  x_v = _mm_sqrt_ps(x_v);
  _mm_storeu_ps(a + i, x_v);
}
```

Listing 3. Vectorization of complex expressions and function calls

```
double a[100], c[100];
int d[100];
#pragma scout loop vectorize
for (int i=0; i<100; ++i) {
  int j = d[i];
  double b = a[j];
// computations

// introduces an inner-loop
// dependency if d[i]==d[i+1]:
#pragma scout vectorize unroll
  c[j] += b;
}
```

```
__m128d b_v;
int j_v[2];
double a[100], c[100];
int d[100];
for (int i=0; i<100; i+=2) {
  j_v[0] = d[i];
  j_v[1] = d[(i + 1)];
  b_v    = _mm_set_pd(a[j_v[0]],
                      a[j_v[1]]);
// vectorized computations

// compute every element separately:
  c[j_v[0]]+=_mm_extract_pd(b_v,0);
  c[j_v[1]]+=_mm_extract_pd(b_v,1);
}
```

Listing 4. Partial vectorization

3.3 Partial Vectorization

Most loops in our codes follow very basic schemes: they read data from several arrays, do some heavy calculations and then either write or accumulate the result in a different array. Hence and under the reasonable assumption, that there are no pointer aliasing issues, pure writes normally don't introduce any dependencies. Accumulation operations however involve a read and write operation to the same memory location and hence can introduce dependencies, especially if there is indirect indexing involved. Such dependencies could prevent whole loops from being vectorized. But actually most of the calculation can be performed in parallel, just the accumulation process itself needs to remain serial. Thus we introduced a pragma directive forcing a statement to compute each vector element separately (Listing 4).

4 Practical Results

Beside the usual test cases we have applied Scout to two different CFD production codes used in the German Aerospace Center. Both codes are written in C using the usual array-of-structure approach. That approach is rather unfriendly with respect to vectorization, because vector load and store operations have to be composite. Nevertheless we did not change the data layout but used the source code as is only augmented with the necessary Scout pragmas. The presented measurements were mainly done on an Intel® Core™ 2 Duo P8600 processor with a clock rate of 2.4 GHz, operating unter Windows 7™ using the Intel® compiler version 11.1. The AVX measurements were done on a an Intel® Sandy Bridge™ processor, using the Intel® compiler version 12.

The first code computes interior flows in order to simulate the behavior of jet turbines. In the loops direct indexing is used meaning array indices are linearly transformed loop indices. We have split the code in four computation kernels and present the splitted results for a better understanding of the overall and detailed speedup in Fig. 1. It shows typical speedup factors of the vectorized kernels produced by Scout compared to the originals.

As expected, we gain more speedup with more vector lanes, since more computations can be executed in parallel. Kernel 2 even outperforms its theoretical maximum speedup, which is a result of the other transformations (in particular function inlining) performed by Scout implicitely.

Table 1 shows the effects of AVX on the performance of a complete run. The first row shows the average time of one run including the computation kernels and some framework activity. Naturally, this measurement method reduces the overall speedup gained due to the vectorization but leads to very realistical results. After all, the application of Scout reduces the runtime automatically by about 10%. We expected a much better speedup by stepping up from SSE4 to AVX because the vector register size has doubled on AVX.

However, the additional gains were rather negligible. The second row shows the main reason for this behavior. The CPI metric (*Clockticks per Instructions*

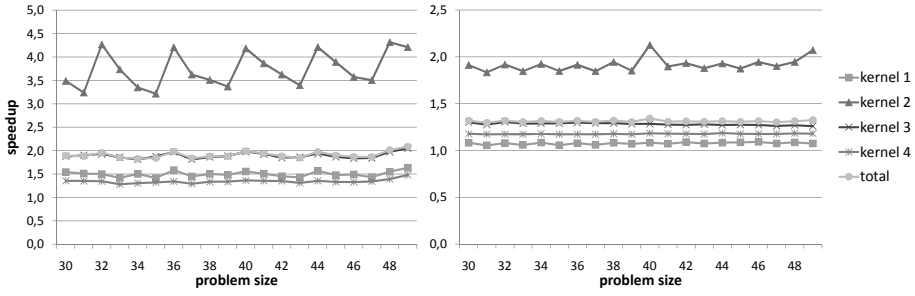

Fig. 1. Speedup of CFD kernels on Intel Core 2 Duo due to the vectorization (left side: single precision, four vector lanes, right side: double precision, two vector lanes)

Retired) is an indication of how much latency affected the execution. Higher CPI values mean there is more latency. In our case the latency is caused mainly by cache misses. This comes with no surprise, because with a doubled vector size also a doubled amount of data gets pumped through the processor during one loop iteration. Even if this effect is well documented [2] a CPI value of 2.0 still means there is a lot of room for improvements. In Sect. 6 we outline a possible approach in order to address that issue.

Table 1. Effects of Scout to a CFD production code on Intel Sandy Bridge

	Intel 12 SSE4	Intel 12 AVX	Scout + Intel 12 SSE4	Scout + Intel 12 AVX
avg. Runtime [sec]	6.31	6.32	5.70	5.65
CPI	0.88	0.88	1.34	2.00

The second CFD code computes flows around an air plane. Unlike the other code it works over unstructured grids. That is, the loops use mostly indirect indexing to access array data elements. Most loops in that kernel could only be partially vectorized (see Sect. 3.3). Nevertheless we could achieve some speedup as shown in Table 2. We had two different grids as input data to our disposal. First we vectorized the original code. However the gained speedup of about 1.1

Table 2. Speedup of a partially vectorized CFD kernel on Intel Core 2 Duo (double precision, two vector lanes)

Relation	*original* to *vectorized*	*merged* to *merged and vectorized*	*original* to *merged and vectorized*
Grid 1	1.070	1.391	1.489
Grid 2	1.075	1.381	1.484

was not satisfying. Then we merged some loops inside the kernel together to remove repeated traversal over the indirect data structures. This made the code more compute-bound and resulted in a much better acceleration of about 1.4 just due to the vectorization. Eventually the overall speedup was nearly 1.5.

5 Summary and Conclusion

Traditionally, auto-vectorization is considered to be a compiler feature. However, for various reasons compilers fail to vectorize a wide range of loops. Thus we have introduced Scout in order to better exploit existing SIMD features supplied by most modern processors. By using the unroll-and-jam approach we were able to extend loop vectorization by some new and unique features and capabilites. We are not aware of a compiler or another vectorization tool which provides the means for a partial vectorization (Sect. 3.3). In addition, at least all compilers available to us refused to vectorize loops with mixed data types (Sect. 3.1). Most compilers have the capability to detect and vectorize common expressions to idiomatic vectorized counterparts. However, these capabilities are mostly hidden in the code of the compiler and cannot be configured by the user. On the other hand the configuration framework of Scout provides a great tool in order to be able to vectorize code for various target platforms, even for user-specific ones (Sect. 3.2).

Section 4 presents the use of vectorization technology from a practitioners point of view. It is worth mentioning that by just augmenting the source code with pragmas and using Scout we could always achieve considerable speedups. Of course, a further hand-tuning of code may lead to even better results (Table 2). However, we emphasize that Scout nowadays is used nearly transparently in the software production process of the German Aerospace Center in order to speedup their codes automatically.

6 Future Work

While the achieved acceleration presented in this paper was already rather good, it was not as exciting as one would expect due to the number of available vector lanes. Of course Amdahl's law plays a rather large role in our results. We did not change the data layout and thus had to live with composite load and store operations. That in turn leads to a smaller parallel portion of code and hence lesser speedup.

But the presented AVX results, especially the raise of the CPI value, indicate memory accesses as another major obstacle for performant SIMD code. Actually, compute-bounded code often gets memory-bound due to vectorization. Of course, the cache pressure can be reduced by a carefully hand-crafted data layout. But the cache size is a hard limit and even hand-crafting sometimes is not worth the rather huge effort. Thus, in order to regain a load balance between memory and computation, we will explore the energy-saving possibilites of memory-bounded computations.

Our approach combines Scout and a performance event governor (*pegov*)[10]. pegov increases a CPUs p-State - thereby reducing its frequency and voltage - during the execution of memory-bound code. As presented in [10] this can lead to substantial energy savings. Thus, first Scout makes the code faster, but also increases the memory burden. Even though memory-bounded regions are rarely speed up by vectorization, one can increase the performance metric "energy efficiency" by using a performance aware governor like pegov. We expect, that the combination of these approaches can produce faster and more energy-efficient code automatically.

Acknowledgments. This work has been funded by the German Federal Ministry of Education and Research within the national research project HI-CFD (01 IH 08012 C) [4].

References

1. clang: a C language family frontend for LLVM, `http://clang.llvm.org` (visited on March 26, 2010)
2. Intel VTune Performance Analyzer Basics: What is CPI and how do I use it? `http://software.intel.com/en-us/articles/intel-vtune-performance-analyzer-basics-what-is-cpi-and-how-do-i-use-it/` (visited on June 6, 2011)
3. Loop unswitching, `http://en.wikipedia.org/wiki/Loop_unswitching` (visited on July 19, 2011)
4. HICFD - Highly Efficient Implementation of CFD Codes for HPC Many-Core Architectures (2009), `http://www.hicfd.de` (visited on March 26, 2010)
5. Allen, R., Kennedy, K.: Automatic translation of fortran programs to vector form. ACM Trans. Program. Lang. Syst. 9, 491–542 (1987), `http://doi.acm.org/10.1145/29873.29875`
6. Hohenauer, M., Engel, F., Leupers, R., Ascheid, G., Meyr, H.: A SIMD optimization framework for retargetable compilers. ACM Trans. Archit. Code Optim. 6(1), 1–27 (2009)
7. Kennedy, K., Allen, J.R.: Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Publishers Inc., San Francisco (2002)
8. Larsen, S., Amarasinghe, S.: Exploiting superword level parallelism with multimedia instruction sets. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI 2000, pp. 145–156. ACM, New York (2000), `http://doi.acm.org/10.1145/349299.349320`
9. Pokam, G., Bihan, S., Simonnet, J., Bodin, F.: SWARP: a retargetable preprocessor for multimedia instructions. Concurr. Comput.: Pract. Exper. 16(2-3), 303–318 (2004)
10. Schöne, R., Hackenberg, D.: On-line analysis of hardware performance events for workload characterization and processor frequency scaling decisions. In: Proceeding of the Second Joint WOSP/SIPEW International Conference on Performance Engineering, ICPE 2011, pp. 481–486. ACM, New York (2011), `http://doi.acm.org/10.1145/1958746.1958819`