Auto-Vectorization Techniques for Modern SIMD Architectures

Olaf Krzikalla[1], Kim Feldhoff[1], Ralph Müller-Pfefferkorn[1], and Wolfgang E. Nagel[1]

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

Abstract. The current speed of the development of SIMD hardware promises a steady performance enhancement of vectorized code. But more and broader vector registers as well as new instructions require a constant adjustment of the used auto-vectorization techniques in order to avoid obstacles and to exploit the provided means to a maximum effect. The paper introduces approaches targeting particular features of modern and upcoming SIMD Architectures. All presented approaches were recently realized in our source-to-source vectorizer Scout. They address issues encountered during the vectorization of production codes, mainly from the CFD domain. The performance measurements show considerable speedups of codes auto-vectorized by Scout over conventionally vectorized codes. Thus programs can benefit from modern SIMD hardware even more by enhancing auto-vectorization techniques with the introduced methods.

1 Introduction

Scout is a configurable source-to-source transformation tool designed to automatically vectorize C source code. The C source code produced by Scout contains vectorized loops augmented by SIMD intrinsics. The respective SIMD instruction set is easily configurable by the user. In [8] we describe the tool concerning loop vectorization in general.

We have opted for a strict semi-automatic vectorization. That is, 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 [2]) and unrolling of inner loops wherever possible. The resulting loop body is then vectorized using the *unroll-and-jam* approach.

The *unroll-and-jam* approach became especially useful with the advent of the so-called multimedia extensions in commodity processors [10]. It is nowadays used by most modern compilers (e.g. gcc[11]). In fact, a lot of the features and capabilities provided by the vectorization unit of Scout are based on the *unroll-and-jam* approach. These features and capabilities include:

- outer-loop vectorization
- unrolling of unvectorizable conditions
- simultaneous vectorization of different data types (e.g. `float` and `double`)
- vectorization of common reduction operations (addition, multiplication, saturation)
- partial vectorization in case of inner-loop dependencies.

In Sect. 3 we present another technique derived from the *unroll-and-jam* approach.

Scout has become an industrial-strength vectorizing preprocessor and is already used on a day-to-day basis in the software production process of the German Aerospace Center. In particular we have applied Scout to several computation kernels of CFD codes. These codes are written in C using the usual array-of-structure data layout. That layout is rather unfriendly with respect to vectorization, because load and store operations are not streamable. Instead several scalar load operations have to build together the content of a vector register and scalar store operations have to store back a vector register to memory respectively (*composite load/store*). Nevertheless we did not change the data layout but used the source code as is only augmented with the necessary Scout pragmas.

Figure 1 shows the speedups for a SSE and an AVX platform of one of the CFD codes. This code computes interior flows in order to simulate the behavior of jet turbines. It is based on a three-dimensional structured grid. The grid size denotes the problem size, however two cells at each border are computed differently. Thus the iteration range of all main computation loops is 4 less than the problem size. Data is accessed through direct indexing, hence array indices are linearly transformed loop indices. We have split the code in three computation kernels. The SSE measurements were done on an Intel® Core™ 2 Duo P8600 processor with a clock rate of 2.4 GHz, using the Intel® compiler version 12.1. The AVX measurements were done on an Intel® Sandy Bridge™ processor, using the Intel® compiler version 12.0.

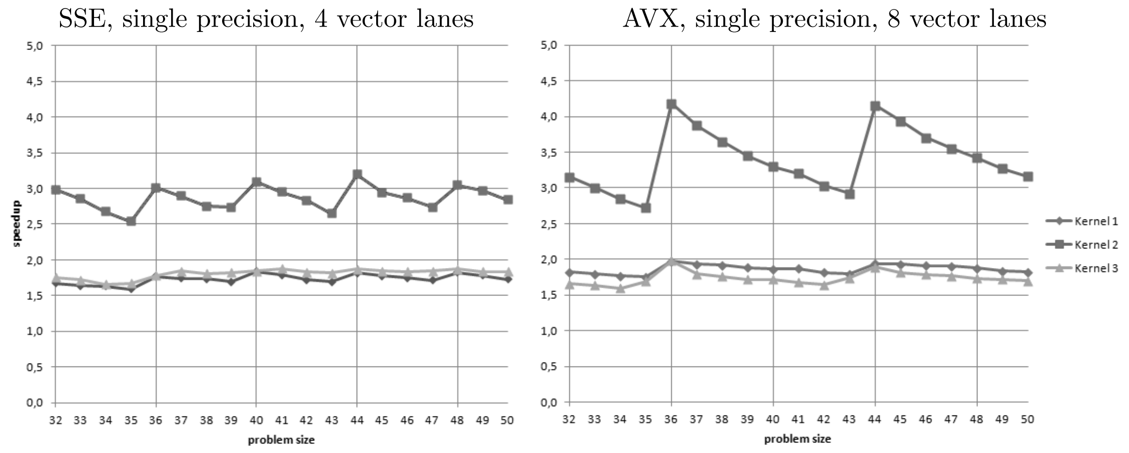

Fig. 1. Speedup of CFD kernels due to the vectorization performed by Scout

There are several conclusions to draw from the comparison of the SSE and AVX performance:

Kernel 1 consists of just some straight-forward matrix computations. But the number of actual computations is low relative to the number of load and store operations. Thus even the SSE version is already memory-bound and thus there is not much to get from simply moving to broader vector registers.

Kernel 2 profits most form the auto-vectorization by Scout. The gained speedup is a sum of the vectorization and other transformations, in particular function inlining. The kernel has rather few memory accesses but plenty of computations. Thus the transition to AVX nets even more speedup. The most limiting factor is an unrolled condition in the vectorized loop. In addition, the scalar computation of the residual loop causes a considerable saw-tooth formation.

Kernel 3 has issues of kernel 1 as well as kernel 2. The SSE version is already memory-bound and in addition there is an unrolled condition in the vectorized loop Thus the speedup increases only from $1,7$ to 2 at best. However, the performance impact of the residual loop computation is notable again.

Beside its use in the software production process Scout also serves as a means to investigate new vectorization techniques. The techniques presented in the following sections address some of the just mentioned issues identified during the transition of code from SSE to AVX platforms. All these techniques are implemented in Scout and ready-to-use. Scout is published under an Open Source license and available via `http://scout.zih.tu-dresden.de`

2 Residual Loop Computation

Comparing the SSE and AVX results of kernel 2 in Fig. 1 shows, that the performance loss due to the residual loop computation grows with the vector register size up to a point where it cannot be ignored furthermore. There are several approaches to address this problem.

A good option is the use of masks for a vectorized residual loop computation. However, modern architectures still do not support masked computations in a way suitable for an automatic vectorization [3]. Another option is the generation of a recursively halved vectorization of the final loop. For example, on Intel's Sandy Bridge processor it is possible to use first AVX and then for the residual loop computation SSE. But this approach highly depends on the target SIMD architecture. In addition, both approaches do not exploit the available width of the vector registers to their full extent.

```
#pragma scout loop vectorize
  for (i=0; i<S; ++i) {
#pragma scout loop vectorize
    for (j=0; j<T; ++j) {
      $s_{i,j}$
    }
  }
```

```
for (i=0; i<S; ++i) {
  for (j=0; j<T-V+1; j+=V) {
    $s_{i,j..j+VS}$
  }
}
for (i=0; i<S-V+1; i+=V) {
  for (j=T-V+1; j<T; ++j) {
    $s_{i..i+VS,j}$
  }
}
for (; i<S; ++i) {
  for (j=T-V+1; j<T; ++j) {
    $s_{i,j}$
  }
}
```

Listing 1: Column Vectorization: general approach

In Scout we have implemented another technique called *column vectorization*. It is derived from our observation, that residual loops have a significant performance impact in nested loops particularly. In fact, kernel 2 is a triply nested loop with a vectorized inner loop. Column vectorization vectorizes the inner loop as usual and then performs an outer-loop vectorization of the residual loop. Listing 1 shows the general loop transformation with a statement s depending on both indices i and j and a vector size V. Scout performs column vectorization, if two nested loops are both marked for vectorization.

The introduced approach exploits the available width of the vector registers much better than a masked or half-vectorized computation. Figure 2 illustrates that difference between a masked and a column-vectorized residual loop computation.

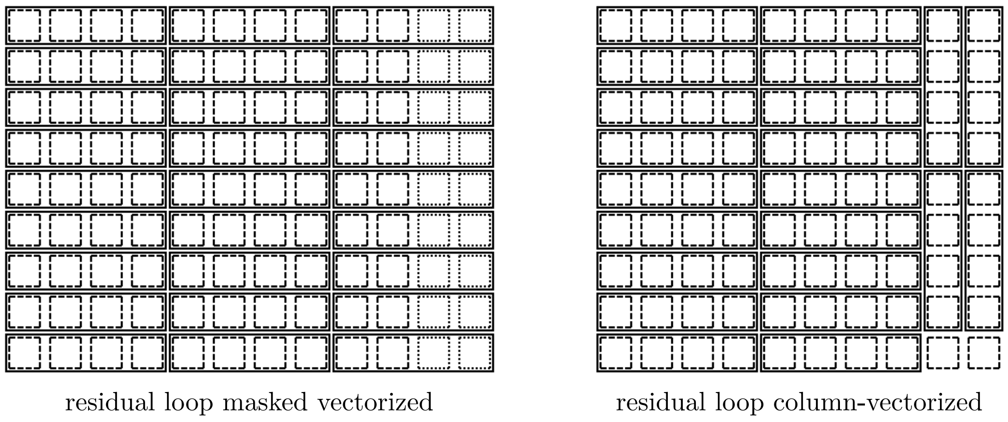
residual loop masked vectorized
residual loop column-vectorized

Fig. 2. Exploitation of vector registers in residual loop computations

Issues can arise due to different memory access patterns in the vectorized main and residual loop. At worst the main loop can use streamed loads and stores while the column-vectorized residual loop has to use composite loads/stores. This introduces sequential parts in the loop body and very probably worsens the cache performance too. Though, normally these are insulated problems of the residual loop.

Another problem is the additional vectorized loop body introduced by the column vectorization. This body enlarges the overall code size of the function and can impair the instruction caching. If the vectorized loop is embedded in yet another loop, the code size can even afflict the overall performance.

Figure 3 shows performance results of a column-vectorized kernel 2[1] compared to the traditional scalar residual loop computation. The performance penalty of the residual loop computation nearly vanishes for the SSE platform and is significantly reduced for the AVX platform. But the column-vectorized AVX version does not reach the peak performance of the conventional vectorized version, which is a result of the just mentioned code bloat. Indeed for that version the compiler issued even a diagnostic that it has turned off high-level optimizations due to the size of the function body. The SSE version does not suffer from this issue mainly because the composite loads generated for that

[1] The SSE version used here was slightly simpler than the version used for Fig. 1

version lead to much smaller code. Section 4 demonstrates the reduction of code size by replacing composite loads with gather operations.

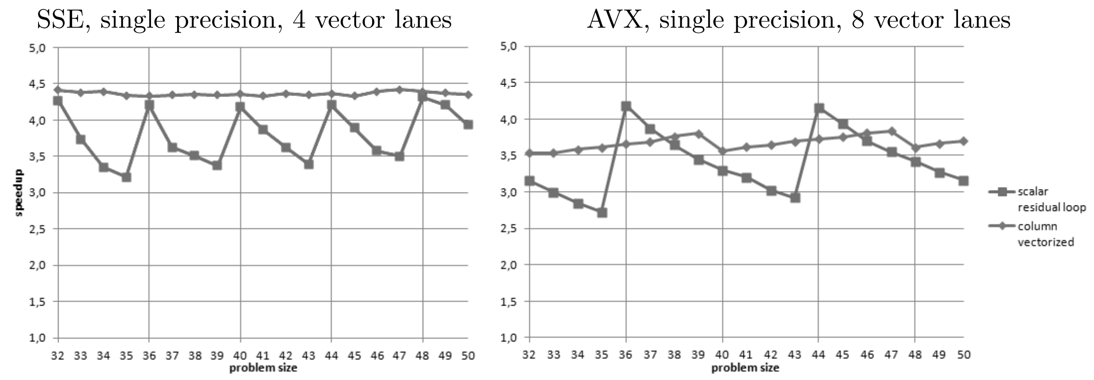

Fig. 3. Performance effect of *column vectorization*

3 Register Blocking

Loop blocking, also known as loop tiling partitions a loop's iteration space into smaller chunks or blocks in order to improve the spatial locality of memory accesses[12, 9]. Thus information in high-speed storage can be used multiple times.

Usually loop blocking is performed by introducing an inner loop iterating over the block size. Our approach differs from the traditional technique, but leads to the same result. Logically, we block every statement. Technically, we do not introduce inner loops but exploit the already implemented unroll feature of the used *unroll-and-jam* approach. Listing 2 demonstrates the idea. According to the value of 8 given to the *size* property, the statements s and t are unrolled 8 times. Since the assumed underlying SIMD architecture provides 4 vector lanes, 4 unrolled statements are merged to one vectorized statement.

```
#pragma loop vectorize size(8)
for (i=0; i<S; ++i) {
   $s_i$
   $t_i$
}
```

```
for (i=0; i<S; i += 8) {
   $s_{i:i+3}$
   $s_{i+4:i+7}$
   $t_{i:i+3}$
   $t_{i+4:i+7}$
}
```

Listing 2: Register Blocking: general approach

A very simple case demonstrating the blocking effect is the loading of loop-invariant variables. Actually, such variables need to be loaded only once in a vector register. However, register spilling becomes inevitable in more complex loop bodies. In that cases register blocking reduces the reloading of spilled registers as shown in listing 3. The variable `c_v` is loaded in a vector register at every start of the loop. Register blocking halves the number of that loads.

```
#pragma loop vectorize size(8)
for (i=0; i<S; ++i) {
  a[i] = b[i] * c;
  // more computations
}
```

```
c_v = { c,c,c,c };
for (i=0; i<S; i+=8) {
  a[i:i+3] = b[i:i+3] * c_v;
  a[i+4:i+7] = b[i+4:i+7] * c_v;
  // c_v is spilled
}
```

Listing 3: Register Blocking: Reuse of loop-invariant variable

Register blocking becomes especially useful, if used in conjunction with outer-loop vectorization. Consider the example in listing 4. In that example `c[j]` is invariant to the vectorized loop, but depends on the inner loop index. Moreover the vectorized variable `c_v` is not spilled but $S * K/VS$ times rebuilt from memory, where VS is the vector size of the target architecture. Blocking the outer loop by a block size of $2VS$ reduces the number of loads by $S * K/(2VS)$ This saves a large absolute number of memory accesses. Thus the presence of inner loops makes register blocking often very worthwhile.

```
#pragma loop vectorize size(8)
for (i=0; i<S; ++i) {
  s = 0;
  for (j=0; j<K; j++) {
    s += x[i+j] * c[j]
  }
  y[i] = s
}
```

```
for (i=0; i<S; i += 8) {
  s_v1 = { 0,0,0,0 };
  s_v2 = { 0,0,0,0 };
  for (j=0; j<K; j++) {
    c_v = { c[j],c[j],c[j],c[j] };
    s_v1 += x[i+j:i+3+j] * c_v;
    s_v2 += x[i+4+j:i+7+j] * c_v;
  }
  y[i:i+3] = s_v1;
  y[i+4:i+7] = s_v2;
}
```

Listing 4: Register Blocking: Reuse of loop-invariant variable

There are several obstacles limiting the positive effect of register blocking. First, the code bloat due to the unrolling can seriously impair the performance of modern microprocessors, since these processors benefit from small loops. Nevertheless register blocking might still be worthwhile for large loop bodies if they contain small time-consuming inner loops. Second, the block size depends on the iteration range. A large block size leads to a large range portion computed in the residual loop. This effect can be alleviated by the generation of several vectorized residual loops with reduced block sizes.[2] And third, a reasonable block size is of course limited by the number of available vector registers. If e.g. the block size in listing 4 becomes too large, the compiler will start spilling some of the unrolled `s_v` variables.

We have measured the effect of register blocking with the computation kernel shown in listing 5, which is derived from production code. As demonstrated in the listing, the value determining the block size can be a symbol or even an expression and is hence adjustable by e.g. compiler switches.

```
#pragma scout loop vectorize size(BLOCK_SIZE)
for (i = 0; i < S; ++i) {
  for (j = 0; j < G; j++) {
    float x = a[j];
    for (d = 0; d < D; d++) {
      x += b[j*D+d] * c[d*S+i];
    }
    output[i * G + j] = x;
  }
}
```

Listing 5: Register Blocking: benchmark code

Figure 4 shows the speedup gained by register blocking. We measured the code on an AVX platform with 8 vector lanes and 16 vector registers. That is, a 4-blocked version will compute 32 elements and a 8-blocked version will compute 64 elements per iteration. The problem size determines the iteration ranges S and G, however S is always rounded down to a multiple of 64 in order to remove the performance effect of the residual loop. The innermost iteration range D is constant. The vectorization replaces the scalar memory accesses with streamed aligned load and store operations. This explains the huge overall speedup. The graph for the unblocked version shows a lot of cache effects, especially for problem sizes in the range from 448 to 512. These effects mostly vanish for all blocked version. They remain notable only in the area just below 512.

The compiled code contains no register spillings in the innermost loop. However, the 8-blocked version contains 5 spilled registers in the G-loop. Thus, especially for larger problem sizes with a higher cache pressure the 8-blocked version does not perform better then the 4-blocked version anymore. All in all, the 4-blocked version exploits the available vector registers most efficiently. This version nets about 40% more speedup than the unblocked version just for the effort of putting the `size` property in the `pragma` directive.

4 Gather and Scatter

Composite load/store operations essentially result in a series of scalar assembly instructions in the compiled code. With only two vector lanes (e.g. `double` at SSE platforms) the ratio of scalar code to vectorized code is still tolerable. However with more vector lanes composite load/store operations become a huge obstacle with respect to the performance effect of the vectorization. While for the

[2] Currently Scout does not perform this optimization automatically.

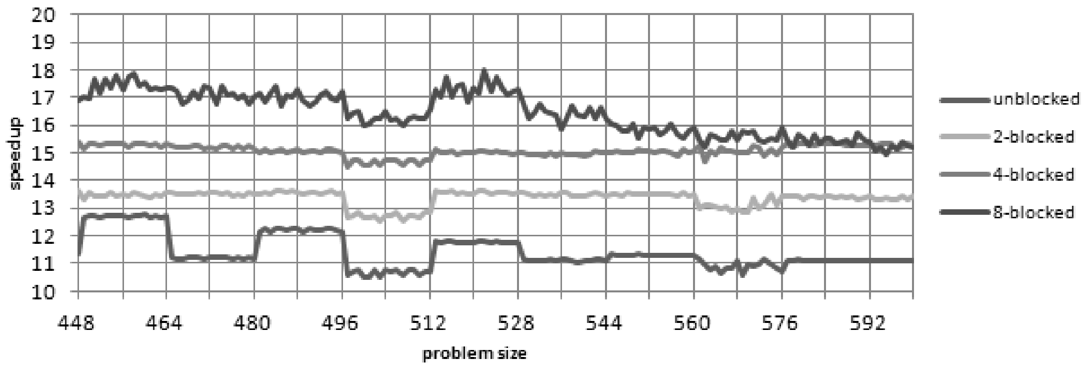

Fig. 4. Performance boost on an AVX processor due to register blocking

same program the parallel portion of code is still the same, the scalar code grows with every vector lane. In fact the AVX platform already supports 8 vector lanes for `float` and other upcoming SIMD architectures have even broader vector registers [7]. Thus targeting these modern SIMD architectures also means, that composite load/store operations must be avoided whenever possible.

Normally the rearrangement of data in a structure-of-arrays layout is not an option due to the huge effort. Fortunately, newer SIMD architectures provide gather instructions, which can load vector elements from distributed memory addresses in a bunch. Currently Scout supports gather instructions, that are implemented by using a base address and a vector register containing indices to compute the actual memory address of each vector element. This is the type of instructions supported by AVX2 [5].

However it is not reasonable to support gather instructions independent of the used data layout. Listing 6 shows two very similar programs. Gather instructions are only useful for the left one, where the distance between two subsequent vector elements is the compile time constant `sizeof(A)`. In the listing on the right side that distance is unpredictable at compile time. Each element of `A` has allocated its own array of `V`s. The vector elements not only end up distributed in memory but the index of each element has to be computed separately. Using a gather instruction would exchange the scalar portion of a composite load for a rather equal sized scalar computation of the indices.

Thus Scout only generates gather instructions, if it can reason at compile time, that all vector elements are members of the same array. The reasoning is implemented using the static analyzer implemented in the *clang* framework [1]. That analyzer provides a taxonomy of memory regions. We investigate each memory access by traversing its super regions. As long as a super region is a record or an array, we continue traversing. Eventually we end up either at an array with a loop-invariant address (the array `a` in the left listing) or we stop at an access through a loop-variant pointer (the access through `v` in the right listing). Only in the first case the generation of a gather instruction is considered.

In the former example, the vector of indices boiled down to a compile-time constant. But gather instructions can also be used if the indices itself are members of another array. Listing 7 shows the technique used in the presence of indirect memory accesses. Again all elements must be members of the same array. In addition the indices must be loadable either by a vectorized load or by another gather instruction. In any other case the index register itself had to be determined by a composite

```
struct V { float x, y; };
struct A { V v[5]; };

A a[100];
for (i=0; i<S; ++i) {
  s = a[i].v[1].x;
}
```

```
struct V { float x, y; };
struct A { V* v; };

A a[100];
for (i=0; i<S; ++i) {
  s = a[i].v[1].x;
}
```

Listing 6: Memory Regions: Only the left code benefits from a gather load

load, which would make a subsequent gather instruction rather pointless. Currently, Scout supports only streamable loads of indices as shown in the example listing, but does not generate recursive gather operations.

```
struct V { float x, y; };
V a[1000];
int idx[100]
#pragma scout loop vectrize
for (i=0; i<S; ++i) {
  s = a[idx[i]].x;
}
```

```
struct V { float x, y; };
V a[1000];
int idx[100]
for (i=0; i<S; i+=4) {
  idx_v = idx[i:i+3] * sizeof(V);
  s_v = gather(&a[0].x, idx_v);
}
```

Listing 7: Gather load in the presence of indirect indexed arrays

Regrettably at the time of the writing of this paper we had no hardware to our disposal supporting gather and scatter instructions. Therefore the results presented here are based on Intel's Software Development Emulator[6]. That is, we have checked the soundness of our approach and our implementation. Furthermore we can give an approximation of the gains by comparing the number of instructions with and without gather loads. However we cannot present any real-time measurements yet.

Table 1 shows the instruction counts of the innermost loop bodies of the three aforementioned CFD kernels. We have traced all executed instructions during one iteration including instructions of called functions. We have then used the trace to estimate the sequential portion of executed code. This portion consists of all instructions dealing with a single vector lane only. The grow of the parallel portion in the transition from AVX to AVX2 is the interesting point with respect to Amdahl's Law.

As expected, the effect highly depends on the properties of the code. Kernel 1 with just its matrix computations benefits most from the use of gather. The only remaining sequential part results from the fact, that AVX2 does not provide a scatter operation. The code of kernel 2 includes not only few load operations but also an unrolled condition. Therefore the parallel portion of that

Table 1. Executed number of instructions on AVX (composite loads) and AVX2 (gather loads)

	total # of instructions		# of sequential instructions		parallel portion	
	AVX	AVX2	AVX	AVX2	AVX	AVX2
Kernel 1	1244	793	715	80	43%	90%
Kernel 2	2451	2232	1553	1219	37%	45%
Kernel 3	2885	2666	1406	1158	51%	57%

loop cannot be increased significantly by replacing the few sequential loads with gather operations. The structure of kernel 3 is actually similar to kernel 1. However, some computations inside that kernel use a form of indirect indexing, that needs a recursive generation of gather operations. And as already mentioned, the generation of recursive gather instructions is not supported by Scout yet. Hence, only a few loads could be transformed to gather operations, other loads remained composite. Very probably a future version of Scout will be able to increase the parallel portion of this kernel.

5 Conclusions

Section 1 presents the use of auto-vectorization technology from a practitioners point of view. It is worth mentioning that by just augmenting source code with pragmas and using Scout we could always achieve considerable speedups. However, by moving the code from SSE to AVX the achieved acceleration was not as exciting as one would expect. We have identified several issues and some of them are addressed in the paper.

Section 2 has discussed the increasing influence of residual loops to the performance. Upcoming architectures [7] will have even broader vector registers. Hence dealing with the problem becomes inevitable. Our *column vectorization* technique addresses the problem at least in the presence of nested loops. This has helped us to solve our practical issues as shown in our performance measurements. A more general solution would be preferable though. The masked execution of residual loops will be realized in Scout once platforms supporting masked instructions suitable for auto-vectorization become available.

Broader vector registers also mean, that more amount of data gets pumped through the processor during the computation. This increases the cache pressure and eventually compute-bounded code gets just memory-bound due to vectorization. In Sect. 3 we have demonstrated, how an extension of the unroll-and-jam approach can reduce the memory burden. With *register blocking* we can bypass the limitations posed by memory transfers in certain codes, especially in small inner loops. The just released version 12.1 of the Intel compiler provides a very similar feature in order to support register blocking.

In Sect. 4 we have investigated the benefits of gather and scatter instructions. With these instructions a programmer can use the usual array-of-structures approach for his or her data layout but nevertheless load or store whole vector registers in one instruction. We have pointed out some preconditions in order to use gather and scatter instructions. We have also given a first approximation of the usefulness of gather instructions in production code. The comparison of the parallel portions of the three kernels has identified some remaining problems. The recursive generation of gather instructions as well as a possible vectorized dealing with conditions is necessary future work in Scout. Eventually we eagerly await the possibility to do some real-time measurements using gather instructions in the very near future.

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]. We would like to thank Alexander Heinecke from Technische Universität München for his valuable input regarding register blocking.

References

1. clang: a C language family frontend for LLVM. Website, available online at `http://clang.llvm.org`; visited on March 26th 2010.
2. Loop unswitching. Website, available online at `http://en.wikipedia.org/wiki/Loop_unswitching`; visited on July 19th 2011.
3. Targeting avx-enabled processors using pgi compilers and tools. Website, available online at `http://www.pgroup.com/lit/articles/insider/v3n2a4.htm`; visited on September 23th 2011.
4. HICFD – Highly Efficient Implementation of CFD Codes for HPC Many-Core Architectures. Website (2009), available online at `http://www.hicfd.de`; visited on March 26th 2010.
5. Intel advanced vector extensions programming reference. Website (2011), available online at `http://software.intel.com/file/36945`; visited on December 3rd 2011.
6. Intel software development emulator. Website (2011), available online at `http://www.intel.com/software/sde`; visited on December 3rd 2011.
7. Abrash, M.: A First Look at the Larrabee New Instructions (LRBni) (2009), available online at `http://www.ddj.com/hpc-high-performance-computing/216402188`; visited on March 26th 2010.
8. Krzikalla, O., Feldhoff, K., Müller-Pfefferkorn, R., Nagel, W.: Scout: A Source-to-Source Transformator for SIMD-Optimizations. In: 4th Workshop on Productivity and Performance (PROPER 2011). Bordeaux, France (August 2011), accepted for publication.
9. Lam, M.D., Rothberg, E.E., Wolfe, M.E.: The cache performance and optimizations of blocked algorithms. In: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems. pp. 63–74. ASPLOS-IV, ACM, New York, NY, USA (1991), `http://doi.acm.org/10.1145/106972.106981`
10. 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. pp. 145–156. PLDI '00, ACM, New York, NY, USA (2000), `http://doi.acm.org/10.1145/349299.349320`
11. Nuzman, D., Zaks, A.: Outer-loop vectorization: revisited for short SIMD architectures. In: PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques. pp. 2–11. ACM, New York, NY, USA (2008)
12. Wolfe, M.: Iteration space tiling for memory hierarchies. In: Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing. pp. 357–361. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1989), `http://dl.acm.org/citation.cfm?id=645818.669220`