Eurographics Symposium on Parallel Graphics and Visualization (2015)
C. Dachsbacher and P. Navràtil (Editors)

Packet-Oriented Streamline Tracing on Modern SIMD Architectures

B. Hentschel,[1,2] J. H. Göbbert,[1,2] M. Klemm,[3] P. Springer,[4,6] A. Schnorr,[1,2] and T. W. Kuhlen[2,5]

[1]Virtual Reality Group, RWTH Aachen University, Germany
[2]JARA – High Performance Computing
[3]Software and Services Group, Intel GmbH, Germany
[4]Aachen Institute for Advanced Study in Computational Engineering Science, RWTH Aachen University
[5]Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH
[6]HPAC – High Performance and Automatic Computing, RWTH Aachen University

Abstract

The advection of integral lines is an important computational kernel in vector field visualization. We investigate how this kernel can profit from vector (SIMD) extensions in modern CPUs. As a baseline, we formulate a streamline tracing algorithm that facilitates auto-vectorization by an optimizing compiler. We analyze this algorithm and propose two different optimizations. Our results show that particle tracing does not per se benefit from SIMD computation. Based on a careful analysis of the auto-vectorized code, we propose an optimized data access routine and a re-packing scheme which increases average SIMD efficiency. We evaluate our approach on three different, turbulent flow fields. Our optimized approaches increase integration performance up to 5.6× *over our baseline measurement. We conclude with a discussion of current limitations and aspects for future work.*

Categories and Subject Descriptors (according to ACM CCS): C.1.2 [Computer Systems Organization]: Multiple Data Stream Architectures (Multiprocessors)—Single-instruction-stream, multiple-data-stream processors (SIMD) I.3.1 [Computer Graphics]: Hardware Architecture—Parallel Processing I.6.6 [Computing Methodologies]: Simulation and Modeling—Simulation Output Analysis

1. Introduction

The computation of integral lines is both a fundamental building block and one of the computationally more demanding kernels in vector field visualization. In this paper, we investigate the use of vector extensions of modern CPUs for particle tracing. Our approach groups multiple seeds into a *packet*. Advection is then performed for the entire packet at once, following the *single instruction, multiple data* (SIMD) approach. SIMD extensions are available in most contemporary CPU architectures (e.g., SSE or AVX for x86, AltiVec or VMX for Power, NEON for ARM) and their use is key to achieving peak performance. This idea is akin to packet ray tracing [WSBW01] in global illumination. Previous work in that area, however, shows that SIMD-parallel execution is not per se beneficial [WBB08, BWW*12]. Typically, only coherent rays, which result in (almost) identical traversals of the underlying bounding volume hierarchy, lead to an actual speed-up. Quickly diverging rays, e.g., secondary rays, can typically be handled more efficiently using an optimized, single-ray traversal. Analogously, diverging particle trajectories, e.g., due to different lengths, may reduce or even neglect the performance benefit of SIMD-parallel processing.

Our main goal is to increase the performance of CPU-based particle tracing on a single CPU core. Our optimized SIMD tracer kernel complements work in the large-data arena in that it can be integrated into a distributed memory parallel particle tracer in order to increase per-node integration performance.

In summary, we make the following two contributions: First, we present a SIMD-parallel formulation for streamline computation which relies on the auto-vectorization features of an optimizing compiler. Second, we develop two optimization approaches which target performance issues in the initial version. Our analysis shows that optimized SIMD computations have a significant impact on the performance, outperforming the scalar version of the same code by a fac-

© The Eurographics Association 2015.

DOI: 10.2312/pgv.20151154

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

tor of up to 5.6×. We conclude with a discussion of the presented approaches, their benefits and shortcomings and outline possible areas of future work.

2. Related Work

Integral curves serve to directly and intuitively depict vector-valued data [SKH*05] and they are a basic building block of a wide range of more sophisticated vector field visualization algorithms, e.g., line integral convolution [CL93], integral surface generation [Hul92, GKT*08], and the computation of the Finite Time Lyapunov Exponent (FTLE) [Hal01, GGTH07, SP07]. For a comprehensive overview, we refer to [MLP*10]. Here, we focus on parallel integral line computation.

Tracing large amounts of particle traces by independently computing individual traces is an embarrassingly parallel problem. Hence, it is well-suited for GPGPU implementations [KKKW05, SBK06, BSK*07, HSW11]. The effect of data reduction by mesh decimation has been investigated in [BRKE*11]. Ferstl et al. use the GPU to interactively compute separating structures [FBTW10]. All these approaches target highly interactive use cases. At the same time, they are inherently limited in the data-set size, either because of the relatively small GPU memory or due to the size of the host workstation's RAM.

In the realm of large-data particle tracing, parallel execution on distributed memory machines is mandatory. Pugmire et al. investigate the scalability of the two standard parallelization approaches: *parallelize-over-seeds* and *parallelize-over-blocks* [PCG*09]. They identify balancing issues and propose a hybrid master-slave approach to solve these. Nouanesengsy et al. perform workload-aware partitioning of the underlying vector field [NLS11]. In [NLL*12], a parallel pathline computation is introduced, which enables large-scale FTLE computations.

Hybrid approaches integrate mixed hardware resources. Camp et al. combine a distributed memory parallel particle tracer with local, thread-level parallelism [CGC*11], whereas GPUs are used as accelerator devices in [CKP*13].

The algorithm proposed in this paper focuses on SIMD-parallel execution on the CPU and hence on low-level memory layout optimization. Our approach complements distributed memory parallel approaches such as [PCG*09, NLS11] or the hybrid approach presented in [CGC*11] by optimizing the low-level advection kernel for individual cores. It differs from GPU-based approaches in that it entirely runs on the CPU and aims to achieve peak performance there. In particular, our algorithm has direct access to the entire host memory without the need to transfer memory back and forth between host and device memory. In order to be able to measure peak CPU performance without confounding factors, we rely on in-core data in this paper. Thus, we do not include a discussion of I/O, which has been shown to be a significant bottleneck. Approaches that help to alleviate this issue are discussed, e.g., in [CCC*11, JEHG14].

SIMD-parallel execution has extensively been studied in the area of ray-casting for global illumination [BWW*12, WSBW01, WBB08, WWB*14]. These approaches are based on the observation that neighboring rays typically intersect the same scene objects. Hence, coherent rays are merged into a *packet* and the tracing computation is executed in a SIMD-parallel fashion which results in highly localized data accesses [WSBW01]. This principle has successfully been applied to the ray-casting of implicit functions [KHH*07] and to direct volume rendering of data given by a set of radial basis functions [KTW*11]. However, performance will quickly degrade if rays diverge from each other, as is the case, e.g., for secondary rays. Vectorized single ray traversal may then be more efficient [WBB08, BWW*12].

3. Experimental Setup

We will discuss our findings inline with the technical exhibition, because the optimizations discussed in the next section build on each other. Therefore, we introduce our experimental setup in this section.

3.1. Test Data & Parameter Settings

In order to cover a number of different application settings, we chose three different flow fields. All three are from turbulent fluid mechanics research and represent real-world particle tracing problems. They share the following characteristics: all three result from Direct Numerical Simulations (DNS) of turbulent flows; they are given on a Cartesian grid; only the vector field is loaded, neglecting all other available data fields. The data is given as single precision floating point numbers and in each scenario we seed 10,000 streamlines.

Turbulent Channel Flow The data set is a fully developed, wall-bounded turbulent flow with a resolution of $512 \times 512 \times 385$ grid points which amounts to 1.13 GB of vector data. The vector field has a dominant velocity component along the x-axis, resulting mostly straight-line traces. Seeds are placed on a uniform lattice which is aligned with the channel entry.

Turbulent Shear Flow This data set shows a turbulent shear flow given on a grid of $1,024 \times 768 \times 768$ which amounts to 6.75 GB of raw data. In this case, particles are seeded in a plane inside the turbulent region, as shown by Gampert et al. [GBH*14]. The complex flow structure in this region, which is evident from Fig. 1 (center) results in relatively long, curled traces.

Homogeneous Isotropic Turbulence This data set of an homogeneous isotropic forced turbulent flow with zero-mean velocity field has a resolution of $1,440^3$ grid points

© The Eurographics Association 2015.

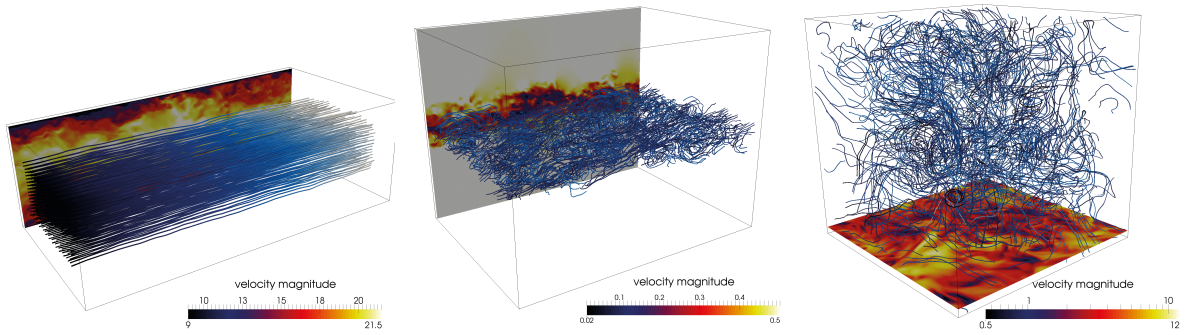

Figure 1: Overview of the three different data sets and their seed set distribution: channel flow (left), turbulent shear flow (center), and isotropic turbulence (right). Velocity magnitude is color coded to the slices in the background whereas streamlines show integration time (dark to light blue).

resulting in 33.37 GB of vector data. In this scenario, seeds are randomly dispersed throughout the domain. Therefore, this benchmark creates a number of quasi-random data accesses for a large number of particle traces.

We use the following integration settings. Traces are computed for at most 1,000 steps unless they pre-maturely leave the domain. Integration uses a fixed-step, fourth-order Runge-Kutta (RK4) scheme. Preliminary experiments showed that performance heavily depends on the integration step length. For shorter steps, data accesses become more localized since particles remain within the same grid cell for several iterations; hence, performance increased due to cache hits. Therefore, we decided to present performance with respect to a number of integration step settings: for the minimum step length the code will perform 20 integrations per cell at maximum velocity, whereas the maximum step length would allow the code to pass two cells within a single integration. We argue that this choice broadly covers the range of practical settings.

3.2. Hardware Configuration

Most modern CPUs feature instruction set extensions that target vector math operations. One instance is the Intel® Xeon® E5-2600v3 family processor. The Xeon processor's design strives to balance single-threaded performance and multi-threaded performance through a moderate number (up to 18) out-of-order cores. In addition to the instructions of the IA-32 and Intel64 instruction sets [Cor14b], the Xeon processor's SIMD units can execute Intel® Advanced Vector Extensions 2 (AVX2, includes Intel® SSE instructions) with 256-bit vector registers. Our benchmark system is equipped with two Intel Xeon E5-2699v3 processors, each of which features 18 cores and a total of 64 GB of RAM at 2133 MHz. All benchmarks are compiled with the Intel® Composer XE 15.0.1 20141023.

The Xeon processor's front-end uses a pre-decoding stage

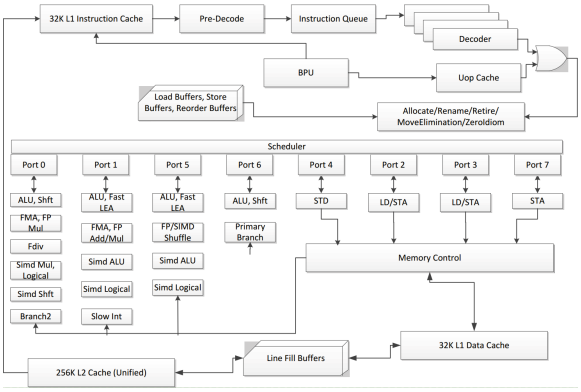

Figure 2: Design of the Xeon core with eight execution pipelines (from [Cor14a]).

that feeds instructions into the back-end (see Fig. 2). The back-end's execution stages have a total of eight pipeline ports: two load/store address ports, one store port, one store address port, and four ports for a mix of scalar and SIMD instructions [Cor14a]. While a scheduling stage performs an out-of-order allocation of the instructions to pipeline ports, the retirement stage at the end of the execution materializes the effects of instructions in correct program order. The pipeline can execute two fused-multiply add instructions or an arbitrary mix of addition and multiplication operations at the same time. For maximum throughput, the instruction stream must equally utilize the different ports by providing a mix of arithmetic, shuffle, and load/store instructions. Unbalanced instruction streams saturate one port, which then becomes the bottleneck and causes the other ports to effectively stall, waiting for overloaded pipelines to finish.

The memory sub-system uses several levels of cache starting with private, inclusive L1 (32 KB) and L2 (256 KB) data caches. The last level cache is a distributed cache structure which can be shared between all cores of the processor pack-

© The Eurographics Association 2015.

Algorithm 1 Standard particle tracer

```
for all seed points s do
  currentPt := s
  while trace not terminated do
    record currentPt
    currentPt :=integrate(v, currentPt)
    check termination for currentPt
  end while
end for
```

Algorithm 2 SIMD-parallel packet particle tracer

```
while unfinishedSeedSet ≠ ∅ do
  currentPacket :=
      groupSeedsIntoPacket(unfinishedSeedSet)
  while HasActiveSeeds(currentPacket) do
    record positions from currentPacket
    currentPacket :=integrate(v, currentPacket)
    check termination for currentPacket
  end while
end while
```

age. The Xeon processor uses several prefetchers to load data from the main memory into the L1 and L2 caches ahead of time.

4. SIMD-Parallel Particle Tracing

Algorithm 1 shows the pseudo-code of a standard particle tracer which advects seeds s through a vector field $\mathbf{v}$. There are two options to make use of SIMD-parallel execution. First, we could use SIMD extensions in order to speed up the computation of a single trace, e.g., by an SIMD implementation of the low-level math computations involved in integration and interpolation. Second, we could exploit the observation that particle advection follows the same computational pattern for all particles, albeit on different data and for a potentially different number of integration iterations.

As stated above, contemporary CPU architectures feature SIMD registers of 256 bits or more. Therefore, basic vector math operations on single-precision floats for 4D points (x,y,z,t) will not fill these registers, which jeopardizes some of the performance potential. Thus, we follow the example of SIMD parallel packet ray tracing: we group multiple particles into a *packet* and perform advection for the entire packet in a SIMD-parallel fashion. This includes the stages of point location, velocity field interpolation, and integration. Algorithm 2 shows the resulting packet-parallel tracer code.

4.1. Facilitating Auto-vectorization

In order to implement an efficient, yet maintainable SIMD parallel particle tracing algorithm, we first investigate the use of *auto-vectorization* by the compiler. In this way, the algorithm can be implemented in a high-level programming language (here: C++) without the need to fall back to hardware-specific features such as intrinsics or assembly programming. This approach's main advantage is that is portable across different architectures. Given a suitable compiler, code can be optimized for a number of target architectures, regardless of specifics such as hardware vendor or size of SIMD registers. Its main disadvantage is that ultimate performance critically depends on the compiler's sophistication.

In order to reveal the inherent parallelism of SIMD

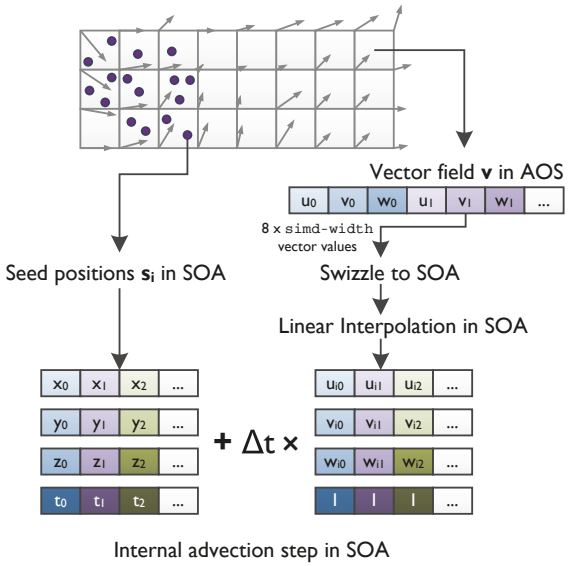

Figure 3: Schematic view of the hybrid data layout assuming a simple Euler integration step.

computations to the compiler, we formulate the integration and interpolation computations in terms of fixed-sized tuples. Each such tuple contains data for `simd-width` many entries per component. All computations use a *structure of arrays* (SOA) layout, i.e., tuples take the form $(x_0,x_1,x_2,\ldots,y_0,y_1,y_2,\ldots,z_0,z_1,z_2,\ldots,t_0,t_1,t_2,\ldots)$ as shown in the lower part of Fig. 3. Computations on these tuples are implemented using `for`-loops whose iteration count is compile-time fixed. These loops form the basic unit for auto-vectorization: whenever deemed save, the compiler automatically translates them into SIMD code.

The SOA layout is flexible with respect to vector register size. For example, a 128 bit-wide SIMD architecture processes four single precision floats in parallel, whereas 512 bit-wide registers pack 16 such entries. Thus, the SOA layout allows us to adjust the size of the packets—the number of particles being processed in parallel—to the target architecture. This can be generically expressed with C++ templates.

© The Eurographics Association 2015.

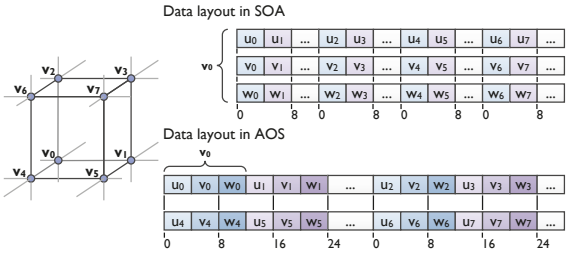

Figure 4: AOS and SOA data layouts for the raw vector data given on a structured grid.

While SOA storage is the default choice for SIMD algorithms, it turns out that this does not straightforwardly apply to particle tracing with its data-dependent, quasi-random data access pattern. The vector field **v** has to be interpolated at each particle position. For any two particles, these positions result in data accesses that typically are not adjacent to each other in memory.

Fig. 4 contrasts the two layouts for raw data storage. For a single, trilinear interpolation, the algorithm has to load the eight vector values at the current cell's vertices. For an SOA layout, these loads will result in requests for at most 24 cache lines, one per point per vector field component. Assuming that consecutive entries along the x-axis collocate in the same cache line, the demand is reduced to 12 potential cache misses. However, out of every line, the interpolation will only consume two float values, i.e., eight bytes, while typical cache lines contain on the order of 64 bytes.

In the AOS layout, the data is packed more tightly. Hence, we need to access only eight cache lines, one per point. Similarly to the AOS case, we can assume that the two data entries in x-direction share the same cache line. As a result, only four cache lines are touched by an interpolation resulting in the consumption of 24 bytes out of every line fetched. Due to this increase, and based on the assumption that particles will rarely be close enough to each other in space that the respective data accesses target the same cache lines, we decided to use an SOA layout for our raw data.

This hybrid layout—i.e., SOA for computation and AOS for data storage—necessitates an on-the-fly format conversion during data access. Fig. 3 gives a schematic view of this process. The interpolation routine first determines the current cells for each particle position in the current packet. It then fetches the corresponding data values in AOS and performs a *swizzle* operation to form eight SOA data tuples each of which holds `simd-width` many vector values for one of the cell's vertices. This data is then used for linear interpolation and subsequent integration in SOA. Note that although Fig. 3 sketches the process for an explicit Euler scheme, our code relies on fourth-order Runge-Kutta integration.

Fig. 5 summarizes our measurements for our test cases (cf. Sec. 3.1). The following particle tracer configurations are shown: tracing with auto-vectorization disabled (*no-vec*), scalar tracing using a packet size of one trace but with auto-vectorization enabled (*SIMD1*), vectorized tracing with a packet size of four (*SIMD4*), vectorized tracing with eight traces per packet (*SIMD8*), optimized interpolation (*AVX2*, see Sec. 4.2), and tracing with re-packing (*packed-AVX2*, see Sec. 4.3).

The results for the first four configurations show that packet tracing with auto-vectorization does not outperform the *SIMD1* setting. We find that auto-vectorization yields only a small speedup far from the theoretically possible 8×. Specifically, we see an average vectorization speedup of 1.13×, 1.34×, and 1.35× for `SIMD1`, `SIMD4`, and `SIMD8`, respectively. Despite the fact that wider a `simd-width` benefits more from vectorization, we see an overall slowdown compared to the the single-trace configuration. This is due to the divergence in the packed particle traces: some traces finish earlier than others, causing redundant computations which are later masked out. This effect is illustrated in Fig. 6, which closely resembles the performance degradation of Fig. 5. Overall, these measurements show that the compiler was not able to take advantage of the vectorization-friendly data layout.

An analysis of the compiler-generated assembly code reveals that the compiler has not been able to vectorize the data transformation from AOS to SOA. Instead, it heavily relies on scalar load and store operations, which reduce the number of useful memory operations in-flight and thus degrade effective memory bandwidth. An IACA [Cor12] analysis of the auto-vectorized code shows that the data ports (i.e., ports 2 and 3 in Fig. 2) are clogged by the instruction stream.

4.2. Optimized Data Access

To address these issues, we decided to optimize the interpolation routine using AVX2 intrinsics. This approach, however, sacrifices portability for performance. The main idea behind the intrinsic implementation is to fully utilize vectorized loads while retaining the hybrid memory layout. Moreover, our AVX2 version directly exploits the fact that the values of two neighboring velocity vectors along the *x*-dimension are successively stored in main memory. Thus, they can be loaded with a single 256 bit-wide load instruction. This is a distinct advantage of the AOS data layout of the vector field: six out of the eight bytes provided by a single load operation are used for computation.

We provide the full source code of our trilinear interpolation scheme in Algorithm 4 on page 10. The SOA layout of the packed traces results in perfect vectorization, i.e., there is not a single scalar instruction.

The interpolation function (cf. Algorithm 4) starts by calculating the offset of the current probe position with respect to the data grid's origin. Since we are using a regular

© The Eurographics Association 2015.

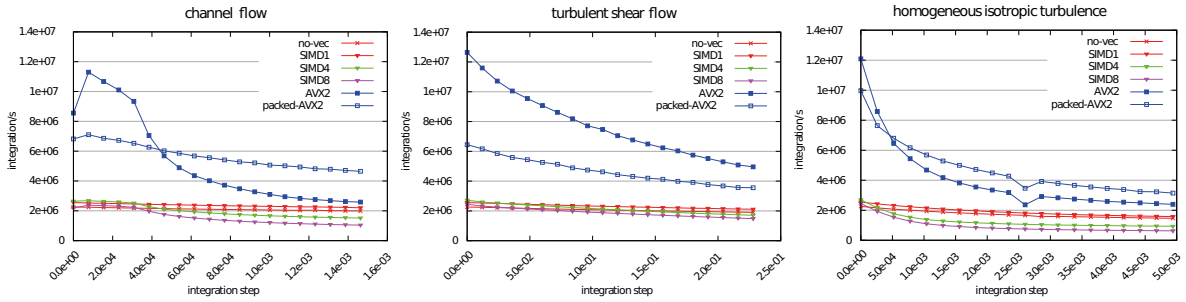

Figure 5: Benchmark results for the three different test cases.

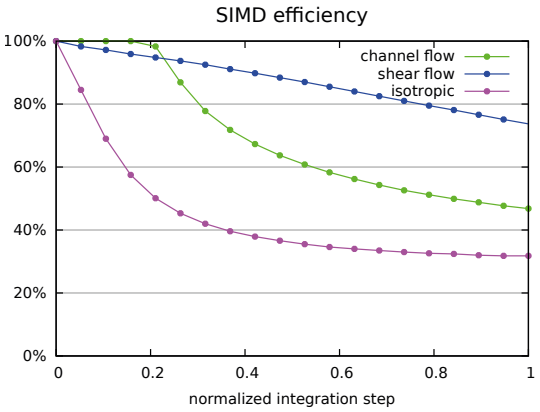

Figure 6: SIMD efficiency (average percentage of active traces) over integration step (normalized by the maximum step for the respective configuration).

grid, we can multiply this offset by the reciprocal grid spacing (lines 11–13) and then convert these values to integers in order to obtain the grid coordinates (lines 16–21). Once these are available for each probe (lines 39–40), we load-and-transpose the velocities v_0^i and v_1^i for each probe $i \in \{0,1,...,7\}$ of the current packet (lines 56, 63, 71, and 79). These operations are performed with only eight vectorized loads compared to 48 scalar loads of the compiler-generated version. Please note that we had to force the compiler to inline these function calls (using `__forceinline`) which resulted in an overall speedup of roughly 15% over the non-inlined version. Because the transpose is fully inlined and intermediate results can be kept in registers, the remainder of the trilinear interpolation (lines 57-93) is trivially vectorized.

Another optimization unique to our AVX2 implementation is that we provide a low-level interface to for the RK4 integrator such that unnecessary loads and stores can be avoided and most of the computation is kept in registers. This optimization yields an additional speedup of roughly 20%.

The results for the AVX2-enhanced interpolation are also shown in Fig. 5. The optimized version provides a performance increase of up to 5.6× for the turbulent shear flow at small integration steps. This confirms that the non-vectorized AOS to SOA transformation of the compiler-generated code has indeed been a major bottleneck. Compared to the AVX2 version, an AVX1 version, which has to simulate some of the AVX2 integer operations, performs approximately 7.6% slower. The AVX-optimized interpolation routines are both much more sensitive to the integration step. Shorter integration steps lead to more interpolations residing inside the same cell which in turn results in better cache performance.

4.3. Re-Packing Particle Traces

The second bottleneck which we have identified in Sec. 4.1 is the degradation of performance due to different trace length and degrading SIMD efficiency (cf. Fig. 6). In order to mitigate this issue, we propose a re-packing scheme that periodically interrupts the advection process, detects terminated traces, and re-packs traces in a way that particles within each packet are spatially close to each other. Algorithm 3 illustrates this process.

Fig. 5 (packed-AVX2) shows the integration performance of the re-packing tracer using the AVX2-optimized interpolation, a uniform binning grid of 32^3 meta cells, and a re-packing every 100 integrations. For the turbulent shear flow, the integration rate for the re-packing tracer is consistently lower than the AVX2-only tracer. For the other two scenarios, however, it eventually passes the AVX2 version.

A closer look at the integration scenarios reveals that most of the traces in the shear flow do not terminate before reaching the threshold of $1,000$ integration steps. Thus, there is no significant difference in trace lengths and consequently SIMD efficiency remains uniformly high. Therefore, the overhead of repeatedly re-organizing particles into new packets is not offset by a performance benefit due to more

© The Eurographics Association 2015.

Algorithm 3 Particle tracer with re-packing.

```
for all seed points s do
  insert s into meta grid
end for
numActiveTraces = numSeeds
while numActiveTraces > 0 do
  for all bins b ∈ meta grid do
    while b ≠ ∅ do
      currentPacket :=
          getPacketSeedsFromBin(b)
      advectNiterations(N,currentPacket)
      terminates =
          checkTermination(currentPacket)
      hashEndPoints(currentPacket)
      numActiveTraces− = terminates
    end while
  end for
end while
```

homogeneous packet runtimes. In contrast, traces regularly terminate before reaching their maximum length for longer integration steps in the other two scenarios. Consequently, we do see a positive effect of re-packing here. A comparison of the relative performance of the AVX2 scheme and the repacking scheme with Fig. 6 shows that the performance benefits are clearly related to SIMD efficiency. The data suggests that the break even point for the re-packing scheme is slightly below 80%.

5. Discussion

Our results show that it is not straightforward to harness the computational power of SIMD parallel execution for particle tracing computations. Among the challenges that need to be addressed are data layout, efficient data access, and non-uniform trace lengths.

The hybrid memory layout is beneficial from a memory access point of view. Yet, it prevents the compiler to auto-vectorize low-level data accesses. Thus, performance of the portable, auto-vectorized code is not better than that of the optimized single-trace version. A manually optimized version of the data AOS-to-SOA data transformation and the subsequent interpolation, however, shows that packet streamline tracing outperforms the scalar code by a factor of up to 5.6×.

Our low-level analysis revealed the following interesting points. First, the auto-vectorized code did not suffer so much from a lack of bandwidth as it did from an excessive number of scalar load operations that clogged up the CPU's corresponding ports 2 and 3 (cf. Fig. 2). Re-arranging the computation greatly reduced pressure on these ports. While this issue seemed to be due to memory bandwidth limitations at first glance, it turned out to be a low-level bottleneck of the underlying CPU architecture.

Second, we observed little to no memory re-use or caching effects for the n-wide packet particle tracer. This is in stark contrast to packet ray-tracing, where ray coherence is key to good performance. In future work, we would like to investigate several options of how to increase memory locality.

There are several aspects that we have not addressed in this paper. First, our algorithm currently relies on a fixed-step, fourth order Runge-Kutta scheme. We deliberately decided not to implement an adaptive step-size scheme. Step-size control critically depends on the integration of each particle. It is non-trivial to perform efficiently in an SIMD-parallel fashion. Nonetheless, we plan to integrate and assess an adaptive integration scheme in future work.

Second, we have not investigated the relationship between SIMD parallel packet tracing and thread-level parallelism. Since all cores on a single CPU share the same memory controllers and the L3 cache, we expect that there will be significant, potentially adverse effects. Moreover, when we move to larger, multi-socket systems, NUMA effects will likely become an issue.

Third, our algorithm currently only works on Cartesian grids. Handling unstructured meshes poses several challenges, chief among them being efficient point location. Tree-based schemes as the CellTree introduced in [GJ10] provide a good solution to this problem. However, it remains to be seen how that solution can be transformed to cater for SIMD queries which comprise multiple particle positions at once. Here, work on the SIMD-parallel traversal of BVHs may provide valuable insights.

Finally, we would like to note that although we hand-coded all routines for the experiments described in this paper, this is hardly a feasible strategy for large-scale, production environments. One way to mitigate this issue is the development of and contribution to highly optimized, visualization APIs which feature basic data structures and algorithms, as recently demonstrated, e.g., in [MAGM11, LSA12, MAPS12]. These APIs abstract much of the low-level optimizations and thus enable the development of portable, yet high-performing visualization codes.

6. Conclusion

In this paper, we have presented a packet streamline tracer which exploits low-level SIMD parallelism. Our optimized solution outperforms the scalar version of our code by a factor of up to 5.6×. A low-level analysis has revealed two performance bottlenecks, which have been addressed in this paper. A manually optimized data access scheme helps to increase interpolation performance, whereas regularly re-packing traces helps to maintain good SIMD efficiency.

The optimizations presented in this paper should provide immediate benefit for other visualization methods that rely

© The Eurographics Association 2015.

on the computation of large amounts of particle traces, e.g., dissipation elements [WP06]. Our future work will focus on the extension of our approach to multi-socket, multi-core machines in order to speed up such large-scale computations.

Acknowledgements

This work has been supported by the DFG under grant KU 1132/10-1 and received funding from the German excellence initiative.

Disclaimer

In the interest of full disclosure, the authors would like to note that the measurements presented in this paper have been performed on hardware provided by Intel Corporation. Intel and Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Other brands and names are the property of their respective owners.

Performance tests are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. System configuration: Intel "Grizzly Pass" server two Intel Xeon E5-2699v3 processors with 18 cores at 2.3 GHz (64 GB DDR4 with 2133 MHz), CentOS Linux release 7.0.1406 (kernel version 3.16.0-rc2+) and Intel S2600WTT, and Intel Composer XE 15.0.1 20141023.

References

[BRKE*11] BUSSLER M., RICK T., KELLE-EMDEN A., HENTSCHEL B., KUHLEN T.: Interactive Particle Tracing in Time-Varying Tetrahedral Grids. In *Proc. of the EG Symp. on Parallel Graphics and Visualization* (2011), pp. 71–80. doi:10.2312/EGPGV/EGPGV11/071-080. 2

[BSK*07] BÜRGER K., SCHNEIDER J., KONDRATIEVA P., KRÜGER J., WESTERMANN R.: Interactive Visual Exploration of Unsteady 3D Flows. In *Proc. of EG/IEEE VGTC Symposium on Visualization* (2007), pp. 251–258. 2

[BWW*12] BENTHIN C., WALD I., WOOP S., ERNST M., MARK W.: Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture. *IEEE Trans. Vis. Comp. Graph. 18*, 9 (2012), 1438–1448. doi:10.1109/TVCG.2011.277. 1, 2

[CCC*11] CAMP D., CHILDS H., CHOURASIA A., GARTH C., JOY K. I.: Evaluating the Benefits of an Extended Memory Hierarchy for Parallel Streamline Algorithms. In *Proc. of the Large Data Analytics and Visualization* (2011), pp. 57–64. doi:10.1109/LDAV.2011.6092318. 2

[CGC*11] CAMP D., GARTH C., CHILDS H., PUGMIRE D., JOY K. I.: Streamline Integration Using MPI-Hybrid Parallelism on a Large Multicore Architecture. *IEEE Trans. Vis. Comp. Graph. 17*, 11 (2011), 1702–1713. doi:10.1109/TVCG.2010.259. 2

[CKP*13] CAMP D., KRISHNAN H., PUGMIRE D., GARTH C., JOHNSON I., BETHEL E. W., JOY K. I., CHILDS H.: GPU Acceleration of Particle Advection Workloads in a Parallel, Distributed Memory Setting. In *Proc. of the EG Symp. on Parallel Graphics and Visualization* (2013), pp. 1–8. 2

[CL93] CABRAL B., LEEDOM L.: Imaging Vector Fields Using Line Integral Convolution. In *Proc. of Siggraph '93* (1993), pp. 263–270. 2

[Cor12] CORPORATION I.: Intel® Architecture Code Analyzer, 2012. http://software.intel.com/en-us/articles/intel-architecture-code-analyzer. 5

[Cor14a] CORPORATION I.: Intel® 64 and IA-32 Architectures Optimization Reference Manual, 2014. Document number 248966-030. 3

[Cor14b] CORPORATION I.: Intel® 64 and IA-32 Architectures Software Developer's Manual, 2014. Document number 325462-052US. 3

[FBTW10] FERSTL F., BURGER K., THEISEL H., WESTERMANN R.: Interactive Separating Streak Surfaces. *IEEE Trans. Vis. Comp. Graph. 16*, 6 (2010), 1569–1577. doi:10.1109/TVCG.2010.169. 2

[GBH*14] GAMPERT M., BOSCHUNG J., HENNIG F., GAUDING M., PETERS N.: The Vorticity Versus the Scalar Criterion for the Detection of the Turbulent/Non-Turbulent Interface. *Journal of Fluid Mechanics 750* (2014), 578–596. 2

[GGTH07] GARTH C., GERHARDT F., TRICOCHE X., HAGEN H.: Efficient Computation and Visualization of Coherent Structures in Fluid Flow Applications. *IEEE Trans. Vis. Comp. Graph. 13*, 6 (2007), 1464–1471. doi:10.1109/TVCG.2007.70551. 2

[GJ10] GARTH C., JOY K. I.: Fast, Memory-Efficient Cell Location in Unstructured Grids for Visualization. *IEEE Trans. Vis. Comp. Graph. 16*, 6 (2010), 1541–1550. doi:10.1109/TVCG.2010.156. 7

[GKT*08] GARTH C., KRISHNAN H., TRICOCHE X., BOBACH T., JOY K. I.: Generation of Accurate Integral Surfaces in Time-Dependent Vector Fields. *IEEE Trans. Vis. Comp. Graph. 14*, 6 (2008), 1404–1411. doi:10.1109/TVCG.2008.133. 2

[Hal01] HALLER G.: Distinguished Material Surfaces and Coherent Structures in Three-Dimensional Fluid Flows. *Physica D: Nonlinear Phenomena 149*, 4 (2001), 248–277. doi:10.1016/S0167-2789(00)00199-8. 2

[HSW11] HLAWATSCH M., SADLO F., WEISKOPF D.: Hierarchical Line Integration. *IEEE Trans. Vis. Comp. Graph. 17*, 8 (2011), 1148–1163. doi:10.1109/TVCG.2010.227. 2

[Hul92] HULTQUIST J. P.: Constructing Stream Surfaces in Steady 3D Vector Fields. In *Proc. of IEEE Visualization* (1992), p. 171Ű178. URL: http://dl.acm.org/citation.cfm?id=949685.949718. 2

[JEHG14] JIANG M., ESSEN B. V., HARRISON C., GOKHALE M.: Multi-threaded streamline tracing for data-intensive architectures. In *Proc. of the Large Data Analytics and Visualization* (2014), pp. 11–18. doi:10.1109/LDAV.2014.7013199. 2

[KHH*07] KNOLL A., HIJAZI Y., HANSEN C., WALD I., HAGEN H.: Interactive Ray Tracing of Arbitrary Implicits with SIMD Interval Arithmetic. In *Proc. of the IEEE Symposium on*

© The Eurographics Association 2015.

Interactive Ray Tracing (2007), pp. 11–18. doi:10.1109/RT.2007.4342585. 2

[KKKW05] KRÜGER J., KIPFER P., KONDRATIEVA P., WESTERMANN R.: A Particle System for Interactive Visualization of 3D Flows. *IEEE Trans. Vis. Comp. Graph. 11*, 6 (2005), 744–756. doi:10.1109/TVCG.2005.87. 2

[KTW*11] KNOLL A., THELEN S., WALD I., HANSEN C., HAGEN H., PAPKA M.: Full-Resolution Interactive CPU Volume Rendering with Coherent BVH Traversal. In *Proc. of the IEEE Pacific Visualization Symposium* (2011), pp. 3–10. doi:10.1109/PACIFICVIS.2011.5742355. 2

[LSA12] LO L.-T., SEWELL C., AHRENS J.: PISTON: A Portable Cross-Platform Framework for Data-Parallel Visualization Operators. In *Proc. of the EG Symp. on Parallel Graphics and Visualization* (2012). doi:10.2312/EGPGV/EGPGV12/011-020. 7

[MAGM11] MORELAND K., AYACHIT U., GEVECI B., MA K.-L.: Dax Toolkit: A proposed framework for data analysis and visualization at Extreme Scale. In *Proc. of the Large Data Analytics and Visualization* (2011), pp. 97–104. doi:10.1109/LDAV.2011.6092323. 7

[MAPS12] MEREDITH J. S., AHERN S., PUGMIRE D., SISNEROS R.: EAVL: The Extreme-scale Analysis and Visualization Library. In *Proc. of the EG Symp. on Parallel Graphics and Visualization* (2012). doi:10.2312/EGPGV/EGPGV12/021-030. 7

[MLP*10] MCLOUGHLIN T., LARAMEE R. S., PEIKERT R., POST F. H., CHEN M.: Over Two Decades of Integration-Based, Geometric Flow Visualization. *Computer Graphics Forum 29*, 6 (2010), 1807–1829. 2

[NLL*12] NOUANESENGSY B., LEE T.-Y., LU K., SHEN H.-W., PETERKA T.: Parallel Particle Advection and FTLE Computation for Time-Varying Flow Fields. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2012), pp. 1–11. doi:10.1109/SC.2012.93. 2

[NLS11] NOUANESENGSY B., LEE T.-Y., SHEN H.-W.: Load-Balanced Parallel Streamline Generation on Large Scale Vector Fields. *IEEE Trans. Vis. Comp. Graph. 17*, 12 (2011), 1785–1794. doi:10.1109/TVCG.2011.219. 2

[PCG*09] PUGMIRE D., CHILDS H., GARTH C., AHERN S., WEBER G.: Scalable Computation of Streamlines on Very Large Datasets. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2009). 2

[SBK06] SCHIRSKI M., BISCHOF C., KUHLEN T.: Interactive Particle Tracing on Tetrahedral Grids Using the GPU. In *Proceedings of Vision, Modeling, and Visualization (VMV) 2006* (2006), pp. 153–160. 2

[SKH*05] SCHIRSKI M., KUHLEN T., HOPP M., ADOMEIT P., PISCHINGER S., BISCHOF C.: Virtual Tubelets - Efficiently Visualizing Large Amounts of Particle Trajectories. *Computers & Graphics 29*, 1 (2005), 17–27. 2

[SP07] SADLO F., PEIKERT R.: Efficient Visualization of Lagrangian Coherent Structures by Filtered AMR Ridge Extraction. *IEEE Trans. Vis. Comp. Graph. 13*, 6 (2007), 1456–1463. doi:10.1109/TVCG.2007.70554. 2

[WBB08] WALD I., BENTHIN C., BOULOS S.: Getting Rid of Packets - Efficient SIMD Single-Ray Traversal Using Multi-Branching BVHs. In *Proc. of the IEEE Symposium on Interactive Ray Tracing* (2008), pp. 49–57. doi:10.1109/RT.2008.4634620. 1, 2

[WP06] WANG L., PETERS N.: The Length-Scale Distribution Function of the Distance Between Extremal Points in Passive Scalar Turbulence. *J. Fluid Mech. 554* (2006), 457–475. doi:10.1017/S0022112006009128. 8

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3 (2001), 153–165. doi:10.1111/1467-8659.00508. 1, 2

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Trans. Graph. 33*, 4 (2014), 143:1–143:8. doi:10.1145/2601097.2601199. 2

© The Eurographics Association 2015.

Algorithm 4: AVX2 Interpolate()

```
__m256 Interpolate_avx_sp(const __m256 pos[3], __m256 *result)
{
   //create mask
   __m256 mask = _mm256_cmp_ps(pos[0], _BoxMin[0], 13);
   mask = _mm256_and_ps(mask, _mm256_cmp_ps(pos[1], _BoxMin[1], 13));
   mask = _mm256_and_ps(mask, _mm256_cmp_ps(pos[2], _BoxMin[2], 13));
   mask = _mm256_and_ps(mask, _mm256_cmp_ps(pos[0], _BoxMax[0], 2));
   mask = _mm256_and_ps(mask, _mm256_cmp_ps(pos[1], _BoxMax[1], 2));
   mask = _mm256_and_ps(mask, _mm256_cmp_ps(pos[2], _BoxMax[2], 2));

   __m256 gridCoordsFraction_x = _mm256_mul_ps(pos[0], _1DivSpacing[0]);
   __m256 gridCoordsFraction_y = _mm256_mul_ps(pos[1], _1DivSpacing[1]);
   __m256 gridCoordsFraction_z = _mm256_mul_ps(pos[2], _1DivSpacing[2]);

   //in case a trace becomes inactive, we load the first grid point
   __m256 gridCoords_x = _mm256_and_ps(mask, _mm256_floor_ps(gridCoordsFraction_x));
   __m256 gridCoords_y = _mm256_and_ps(mask, _mm256_floor_ps(gridCoordsFraction_y));
   __m256 gridCoords_z = _mm256_and_ps(mask, _mm256_floor_ps(gridCoordsFraction_z));
   __m256i gridCoords_idx = _mm256_cvtps_epi32(gridCoords_x);
   __m256i gridCoords_idy = _mm256_cvtps_epi32(gridCoords_y);
   __m256i gridCoords_idz = _mm256_cvtps_epi32(gridCoords_z);

   /*
   * Each trace has to interpolate its velocity from the following coordinates:
   *       v6 ------- v7
   *      / |        / |
   *     /  |       /  |
   *    /   |      /   |    Please note: Each trace might belong to a different box.
   *   v2 ------- v3   |                 Thus, the samples v0, v1,..., v7 might be
   *   |    |     |    |                 different for each trace.
   *   |   v4 ------- v5
   * y |   /       |  /
   *   |  /        | / z
   *   | /         |/
   *   v0 ------- v1
   *         x
   */
   //compute offset v0 for each trace
   __m256i v0 = _mm256_add_epi32(gridCoords_idx, _mm256_mullo_epi32(gridCoords_idy, _grid_resolution[0]));
   v0 = _mm256_add_epi32(v0, _mm256_mullo_epi32(gridCoords_idz, _grid_resolution_xy));

   int pointIds[8] __attribute__((aligned(32)));
   __m256 samples[8] __attribute__((aligned(32)));

   /**
    * Linear interpolation of the form: v01 = v0 + wx * (v1 - v0)
    */
   //compute weights used for interpolation
   __m256 wx = _mm256_sub_ps(gridCoordsFraction_x, gridCoords_x);
   __m256 wy = _mm256_sub_ps(gridCoordsFraction_y, gridCoords_y);
   __m256 wz = _mm256_sub_ps(gridCoordsFraction_z, gridCoords_z);

   __m256i front_bottom_left = v0;
   _mm256_store_si256((__m256i *)pointIds, _mm256_mullo_epi32(front_bottom_left, _dim));
   load_and_transpose8x6(_vectorBasePointer, pointIds, samples);
   // interpolate between v0 and v1 -> v01
   __m256 v01_x = _mm256_add_ps(samples[0], _mm256_mul_ps(wx, _mm256_sub_ps(samples[3], samples[0])));
   __m256 v01_y = _mm256_add_ps(samples[1], _mm256_mul_ps(wx, _mm256_sub_ps(samples[4], samples[1])));
   __m256 v01_z = _mm256_add_ps(samples[2], _mm256_mul_ps(wx, _mm256_sub_ps(samples[5], samples[2])));

   __m256i front_top_left = _mm256_add_epi32(v0, _grid_resolution[0]);
   _mm256_store_si256((__m256i *)pointIds, _mm256_mullo_epi32(front_top_left, _dim));
   load_and_transpose8x6(_vectorBasePointer, pointIds, samples);
   // interpolate between v2 and v3 -> v23
   __m256 v23_x = _mm256_add_ps(samples[0], _mm256_mul_ps(wx, _mm256_sub_ps(samples[3], samples[0])));
   __m256 v23_y = _mm256_add_ps(samples[1], _mm256_mul_ps(wx, _mm256_sub_ps(samples[4], samples[1])));
   __m256 v23_z = _mm256_add_ps(samples[2], _mm256_mul_ps(wx, _mm256_sub_ps(samples[5], samples[2])));

   __m256i back_bottom_left = _mm256_add_epi32(front_bottom_left, _grid_resolution_xy);
   _mm256_store_si256((__m256i *)pointIds, _mm256_mullo_epi32(back_bottom_left, _dim));
   load_and_transpose8x6(_vectorBasePointer, pointIds, samples);
   // interpolate between v4 and v5 -> v45
   __m256 v45_x = _mm256_add_ps(samples[0], _mm256_mul_ps(wx, _mm256_sub_ps(samples[3], samples[0])));
   __m256 v45_y = _mm256_add_ps(samples[1], _mm256_mul_ps(wx, _mm256_sub_ps(samples[4], samples[1])));
   __m256 v45_z = _mm256_add_ps(samples[2], _mm256_mul_ps(wx, _mm256_sub_ps(samples[5], samples[2])));

   __m256i back_top_left = _mm256_add_epi32(front_top_left, _grid_resolution_xy);
   _mm256_store_si256((__m256i *)pointIds, _mm256_mullo_epi32(back_top_left, _dim));
   load_and_transpose8x6(_vectorBasePointer, pointIds, samples);
   // interpolate between v6 and v7 -> v67
   __m256 s67_x = _mm256_add_ps(samples[0], _mm256_mul_ps(wx, _mm256_sub_ps(samples[3], samples[0])));
   __m256 s67_y = _mm256_add_ps(samples[1], _mm256_mul_ps(wx, _mm256_sub_ps(samples[4], samples[1])));
   __m256 s67_z = _mm256_add_ps(samples[2], _mm256_mul_ps(wx, _mm256_sub_ps(samples[5], samples[2])));

   // interpolate between v01 and v23 -> v0123
   // (omitted) ...
   // interpolate between v45 and v67 -> v4567
   // (omitted) ...

   // interpolate between v0123 and v4567 -> result
   result[0] = _mm256_add_ps(v0123_x, _mm256_mul_ps(wz, _mm256_sub_ps(v4567_x, v0123_x)));
   result[1] = _mm256_add_ps(v0123_y, _mm256_mul_ps(wz, _mm256_sub_ps(v4567_y, v0123_y)));
   result[2] = _mm256_add_ps(v0123_z, _mm256_mul_ps(wz, _mm256_sub_ps(v4567_z, v0123_z)));
}
```

© The Eurographics Association 2015.