ISSN 2464-4617 (print) WSCG 2016 - 24th Conference on Computer Graphics, Visualization and Computer Vision 2016
ISSN 2464-4625 (CD-ROM)

JIT-Compilation for Interactive Scientific Visualization

J. S. Mueller-Roemer
Fraunhofer IGD, TU Darmstadt
Fraunhoferstr. 5
Germany 64283, Darmstadt, Hessen
Johannes.Mueller-Roemer@igd.fraunhofer.de

C. Altenhofen
Fraunhofer IGD, TU Darmstadt
Fraunhoferstr. 5
Germany 64283, Darmstadt, Hessen
Christian.Altenhofen@igd.fraunhofer.de

ABSTRACT

Due to the proliferation of mobile devices and cloud computing, remote simulation and visualization have become increasingly important. In order to reduce bandwidth and (de)serialization costs, and to improve mobile battery life, we examine the performance and bandwidth benefits of using an optimizing query compiler for remote post-processing of interactive and in-situ simulations. We conduct a detailed analysis of streaming performance for interactive simulations. By evaluating pre-compiled expressions and only sending one calculated field instead of the raw simulation results, we reduce the amount of data transmitted over the network by up to $2/3$ for our test cases. A CPU and a GPU version of the query compiler are implemented and evaluated. The latter is used to additionally reduce PCIe bus bandwidth costs and provides an improvement of over 70% relative to the CPU implementation when using a GPU-based simulation back-end.

Keywords

Scientific Visualization, Network graphics, Mobile Computing, Compilers, LLVM, JIT

1 INTRODUCTION

In modern computer-aided engineering (CAE), compute-intensive simulations are more and more often run on remote cloud or high-performance computing (HPC) infrastructures. To avoid downloading large simulation results to a local client machine, solutions for remote visualization and remote post-processing are needed. Although the option of using a standard visualization tool via a video streaming system such as Virtual Network Computing (VNC) [Ric+98] is attractive, it is desirable to keep latencies to a minimum to increase usability [TAS06]. By transferring (partial) floating point simulation data instead, operations such as probing or changes in color mapping can be performed locally with minimal latency. Similarly, by transferring geometry or point data in 3D, smooth camera interaction becomes possible [Alt+16].

In particular, we aim to answer the following questions:

1. Can compiler technologies be used to decrease visualization latencies in a remote scientific visualization system?

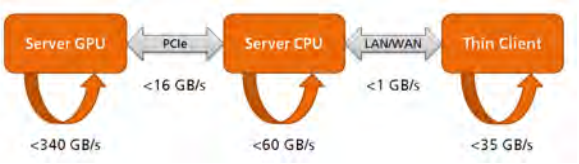

Figure 1: Network, bus and memory bandwidths relevant to streaming a GPU-based simulation. The two most limiting factors are the network bandwidth and the PCIe bus bandwidth.

2. Can GPU-based simulations running at interactive rates profit from GPU-based query compilation?

When individual result fields of a simulation are visualized, data can simply be streamed from the server running the simulation. When viewing derived values that depend on multiple fields such as the total energy density $\frac{v^2}{2} + gz + \frac{p}{\rho}$ in an Eulerian computational fluid dynamics (CFD) simulation, a different solution is required, as transferring all data would be prohibitive, especially when considering comparatively slow mobile connections (see Fig. 1) and mobile power consumption.

A simulation service could provide a fixed set of derived values. However, the derived values a user wants to visualize often depend not only on the physics domain, but also on the application domain. Therefore, compiling such a fixed set requires domain knowledge and is very likely to be incomplete and insufficient for the user to perform his or her work. For stationary simulations, a server-side interpreter for user queries is en-

tirely sufficient, as each query only has to be processed once. For interactive simulations, i.e., time-dependent simulations running at several frames per second, or the in-situ visualization of a long-running solver, however, this approach becomes costly due to the repeated interpretation overhead.

To avoid these costs, we examine the performance and bandwidth benefits of using optimizing compiler technologies for remote, in-situ post-processing and visualization of simulations running at interactive rates. The implemented query compiler has a native CPU back-end (x86 and x86-64) as well as a GPU back-end (NVIDIA PTX). The latter is used to extend the bandwidth savings to the PCIe (Peripheral Component Interconnect Express) bus in addition to the network interface, further improving performance when using GPU-based simulation algorithms. Our approach is easily extended to all platforms supported by LLVM [LA04].

2 RELATED WORK

This section describes existing methods that are related to our approach and briefly shows their benefits and drawbacks.

2.1 Compiler Technologies for Visualization

Previous applications of compilers and domain-specific languages (DSLs) to scientific visualization mostly center on volume visualization and rendering itself [Chi+12; Cho+14; Rau+14]. These systems therefore represent the entire visualization pipeline. In the streaming architecture presented in this paper, data is transformed on the server and rendered on the client. Therefore, the aforementioned systems are not directly applicable. This split corresponds to the two stages "Data Management" and "Picture Synthesis" in the system architecture used by [Duk+09]. However, they use an embedded DSL (eDSL) based on Haskell [Pey03]. As client code must be considered untrusted by the server, a general-purpose language and any eDSL based on such a language pose a great security risk. In the area of visual analytics, MapD Technologies [Map16] have recently used LLVM/NVVM [NVI16] and GPU computing with great success [MŞ15]. In contrast, we aim to bring the advantages of using compiler technologies to the field of scientific visualization, with a focus on interactively changing datasets from either in-situ or interactive simulations.

2.2 Compression

Another approach to reduce bandwidth requirements is to apply floating-point data compression. For structured data, lossy methods such as the one presented in [Lin14] achieve good results. Structured data occurs in a significant subset of simulation domains and such a method would be widely applicable. However, lossy compression before calculation of desired derived values can lead to larger errors in the compounded result. For general data, a method such as the one presented in [OB11] could be used. Their method is a lossless compression method and implemented on the GPU, making it applicable to reducing network as well as PCIe bus bandwidths and to arbitrary simulation domains. As compression is orthogonal to the method presented in this paper, any suitable compression algorithm can be chosen and combined with our approach. However, all compression methods incur an additional computation cost. A good overview of existing compression techniques for floating-point data is given in [RKB06], showing compression ratios as well as compression and decompression times.

2.3 Application Sharing

Although we present a method to reduce the amount of data transferred when the client performs part of the necessary calculations to reduce perceived latency, it is worth mentioning that transmitting the content of single applications or the entire desktop as an image or video stream is still a common way to visualize server applications on (thin) client machines across a local network or the Internet. Microsoft's Remote Desktop Protocol (RDP) [Mic16] or the platform-independent Virtual Network Computing (VNC) [Ric+98] are two popular implementations of this concept. Good results have also been achieved in the area of video streaming for games [Che+11]. However, mobile networks, especially 3G networks, can add several hundreds of milliseconds of latency [Gri13].

As shown in this section, many approaches for remote visualization exist in the context of scientific visualization and visual analytics. However, the potential of compiler technology in the field of remote visualization of interactive simulations has not been discussed yet. Especially in modern high performance computing (HPC) or cloud environments, these techniques can greatly improve usability by optimizing data transmission and increasing update rates on the clients, while minimizing server overhead and latency. Existing compression algorithms can be applied independently to decrease the required bandwidth even further. However, the resulting increase in encoding and decoding time has to be kept in mind.

3 CONCEPT AND IMPLEMENTATION

In this section, we present our prototype visualization system, which consists of:

1. an interactive simulation back-end running on the server

2. a visualization front-end running on the client
3. an application-specific streaming protocol
4. the query expression compiler

Using the streaming protocol, simulation data is transmitted at interactive rates from the server to the client. By transmitting data instead of images, many interactions, for example color map changes, become possible on the client without incurring network round trip and transmission latency. When the user wants to visualize values that are not a direct output of the simulation back-end, the query expression compiler is used to efficiently transform data on the server, reducing network bandwidth requirements. The prototype is based on a CFD simulation back-end, however, the method is directly applicable to other physical domains such as computational solid mechanics (CSM), computational aero-acoustics or computational electrodynamics. For easy reuse with other simulation back-ends, the query compiler is designed as a shared library with a simple interface.

In the following, we briefly outline the simulation back-end as well as the visualization front-end and detail the streaming protocol as well as the query compiler.

3.1 Simulation Back-End

Our query-based streaming prototype is based on an interactive, Eulerian 2D/3D-CFD code for staggered regular grids using a multigrid solver based on the one presented in [Web+15]. All computational kernels are implemented in CUDA [Nic+08]. Therefore, GPU-CPU transfers are only required for data that is sent to the client.

3.2 Visualization Front-End

Two streaming clients have been implemented:

1. A graphical client running on a desktop machine shown in Figure 2.
2. An HTML5+JavaScript client for streaming performance measurements shown in Figure 3.

The former allows user interaction such as selecting the results to show, or entering an expression combining multiple result fields. Furthermore, the color mapping can be interactively modified by manipulating the color ramp widget with the mouse. The latter was developed to determine feasibility of a web client by evaluating streaming performance including deserialization. Both can be used to stream regular 2D and 3D grids. However, visualization is limited to 2D slices in the prototype.

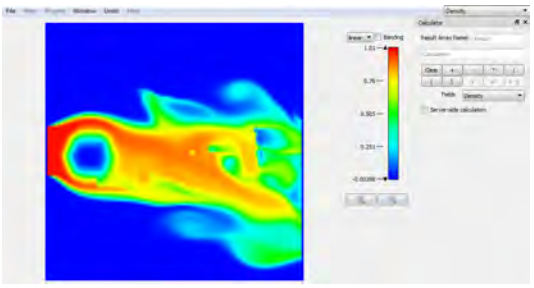

Figure 2: The graphical streaming client. The user can choose which result field to view or enter an expression combining multiple fields. Color mapping can be modified interactively by clicking and dragging the color ramp widget.

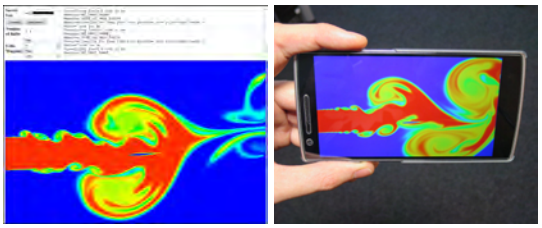

Figure 3: The web-based streaming client can be run in a web browser on desktop computers or mobile devices without installing additional software and provides a simple user interface including 2D visualization and basic logging functionality.

3.3 Streaming Protocol

The streaming protocol is based on Protocol Buffers (ProtoBuf) [Goo08] for serialization and deserialization. ProtoBuf is a platform-independent open source framework that generates serialization and deserialization code from a declarative message description, which greatly simplifies modifications to the protocol. Implementations of ProtoBuf are available for a large number of programming languages, including C++ (as used by our server) and JavaScript. To minimize overhead, the fields of physical values are marked as packed repeated fields, as shown in Listing 1. This prevents ProtoBuf from inserting type tags between each value and ensures that values are transmitted contiguously. The generated messages are transmitted using the WebSocket protocol.

Although WebSockets are based on TCP and have a greater overhead than using UDP, they have several advantages. First, WebSockets ensure that message order is preserved and that all messages are received unless the connection is lost entirely, simplifying client and server implementation. Second, an increasing number of mobile applications are provided as HTML5 web applications and WebSockets are supported by all current browsers, while TCP and UDP are not accessible from JavaScript. This ensures portability of our streaming solution to HTML5+JavaScript.

Listing 1: Main streaming message definition (ProtoBuf [Goo08]), showing the use of a packed repeated field for physical values to reduce overhead.

```
message PostGridFields
{
    required Header header = 1;

    required int32 gridSizeX = 2;
    required int32 gridSizeY = 3;

    repeated int32 posted_fields = 4 [packed=true];
    repeated float values = 5 [packed=true];

    optional Statistics statistics = 6;
    optional int32 gridSizeZ = 7;
}
```

While streaming, the client sends frame request messages whenever a simulation time step (frame) is received, causing the server to send the most current time step that has been computed since sending the previous one. This ensures that no more messages are sent than can be transferred, which would lead to buffer overruns. To prevent bandwidth from being wasted due to the latency of requesting a new frame only after the previous one has been received, two frames are requested when a new connection is established, which corresponds to double buffering. A larger number of frames could be pre-requested as well (triple buffering or more) to overcome larger transient bandwidth changes. A full evaluation over varying buffer sizes was not performed within the scope of this paper. Using the double buffering approach as described leads to an improvement in bandwidth exploitation of up to 50% compared to the naïve implementation.

Which fields are streamed to the client is determined by a query. The streaming prototype currently supports two query types:

1. Any number of result fields, e.g., `VelocityX` and `VelocityY`.
2. A query expression combining multiple fields into one.

The former is used when individual results are viewed by the user, and when post-processing is performed on the client for evaluation. The latter is forwarded to our query compiler or an interpreter that was implemented for comparison. A query expression consists of identifiers for the respective available results fields, operators or functions combining them, and parentheses for controlling operator order. The identifiers are specific to the simulation back-end and characteristic of the respective physical domain, e.g., `VelocityX`, `VelocityY` or `Pressure` for fluid simulations, or `DisplacementX`, `StressXX` or `StressXY` for structural mechanics simulations. These identifiers can be used to evaluate combinations of multiple fields such as `(VelocityX^2 + VelocityY^2) / 2 + Pressure`, which corresponds to $\frac{|\vec{v}|^2}{2} + p$, the sum of kinetic and static energy densities of a fluid with density $\rho = 1$.

3.4 Query Compiler

The query compiler prototype consists of an expression parser and an LLVM-based, optimizing back-end. Additionally, an interpreter has been implemented. The compiler is packaged as a shared library, for easy reuse on both client and server.

For many optimizations, especially vectorization, the optimizer must have knowledge if pointers to data:

1. ... may alias or not. Aliasing occurs if the same address in memory is reachable via different pointers. Aliasing prevents vectorization, as it can introduce additional dependencies between loop iterations if a pointer to data that is being read from can alias a pointer to data that is written to.
2. ... are aligned or not. Aligned data is allocated with at an adress that is a multiple of a specific power of two. This information is relevant as many vector instruction sets require loads and stores to be aligned to achieve maximum throughput.
3. ... are captured or not. A captured pointer is stored somewhere and may later on be accessed via a different call. This is mostly relevant to callers of a specific function to know if a piece of data remains accessible.
4. ... point to data that is read, written or both. This information is mostly relevant to callers who may want to reorder function calls.

Such information can be passed to LLVM via the use of function and parameter attributes. To maximize the number of optimization opportunities, the CPU back-end generates LLVM intermediate representation code (LLVM-IR) annotated with the appropriate parameter and function attributes according to the LLVM Performance Tips for Frontend Authors[1] (see Listing 2). Specifically, annotating input pointers with the `readonly` and `nocapture` attributes and the output pointer with `noalias`. However, `nocapture` and `readonly` can be inferred by the compiler and did not affect optimization. In previous LLVM versions, the use of `noalias` was necessary to ensure that vectorizing optimizations are not blocked by alias analysis. In the current LLVM

[1] `http://llvm.org/docs/Frontend/PerformanceTips.html`

top-of-tree as of March 2016 vectorized code is generated independent of the presence of the `noalias` attribute. To do so, LLVM adds runtime aliasing checks and a non-vectorized version of the code. However, this increase in code size and the additional check showed no measurable effect on time measurements in our use case. Additionally, alignment annotations (`align` n) can be used so that aligned moves are emitted instead of unaligned moves. Evaluations in a separate test environment with a result field of 4096^2 values did not result in any change in performance on either an Intel Xeon E5-2650 v2 CPU or an Intel Core i7-3770 CPU. In light of this result and as using alignment in the complete process would have required changes to the simulator's allocation strategy, alignment attributes were not used in the final evaluation.

For the GPU back-end, the LLVM NVPTX target was chosen. Alternatively, NVIDIA's proprietary NVVM-IR or OpenCL's SPIR could have been used, as both are based on LLVM-IR as well. NVVM-IR is used with libnvvm [NVI16], NVIDIA's compiler library. libnvvm supports additional proprietary optimizations, which can lead to improved performance. SPIR can be used with OpenCL to support both AMD and NVIDIA GPUs. However, both NVVM-IR and SPIR are based on older LLVM versions. Therefore, using either would mean using two different versions of LLVM for CPU and GPU code, or not having the full range of CPU optimizations, such as vectorization in the presence of potential aliasing, and instruction sets supported in current versions available. In the future, the addition of SPIR-V [Kes15], the binary intermediate representation introduced with Vulkan and OpenCL 2.1, as an additional target for LLVM [Yax15] will make targeting all platforms that support OpenCL significantly simpler.

LLVM's optimization pipeline consists of a set of passes which take LLVM-IR as input and produce transformed LLVM-IR as output, as well as a number of analysis passes. One such pass is the instruction combining pass, which replaces complex instruction sequences by simpler instructions if possible. Among these are transformations that convert calls of math library functions such as `powf` to calls of faster functions such as `sqrtf` for `powf(`x`, 0.5)` or individual floating point instructions for `powf(`x`, 2)`. However, these functions are identified by name and NVIDIA libdevice math library prefixes all names with `__nv`. To make full use of the instruction combining pass for GPU code as well, we generate code using unprefixed calls and run a subset of optimizations (primarily inlining and instruction combining) before retargeting call instructions to the prefixed versions and linking libdevice. After linking, the full set of optimization passes is run.

Listing 2: LLVM IR generated by the query compiler before optimization for an expression equivalent to a `saxpy`-operation.

```
; Function Attrs: alwaysinline nounwind readnone
define private float @kernel(float, float, float)
    #0 {
entry:
  %3 = fmul float %0, %1
  %4 = fadd float %3, %2
  ret float %4
}

; Function Attrs: nounwind
define void @map(i64, float* noalias nocapture,
    float, float* nocapture readonly, float*
    nocapture readonly) #1 {
entry:
  %5 = icmp ult i64 0, %0
  br i1 %5, label %body, label %exit

body:                          ; preds = %body, %
    entry
  %6 = phi i64 [ 0, %entry ], [ %13, %body ]
  %7 = getelementptr inbounds float, float* %3, i64
       %6
  %8 = load float, float* %7
  %9 = getelementptr inbounds float, float* %4, i64
       %6
  %10 = load float, float* %9
  %11 = call float @kernel(float %2, float %8,
      float %10)
  %12 = getelementptr inbounds float, float* %1,
      i64 %6
  store float %11, float* %12
  %13 = add nuw i64 %6, 1
  %14 = icmp ult i64 %13, %0
  br i1 %14, label %body, label %exit

exit:                          ; preds = %body, %
    entry
  ret void
}

attributes #0 = { alwaysinline nounwind readnone }
attributes #1 = { nounwind }
```

Unlike a general purpose, Turing complete programming language, the simple nature of our query expressions ensures that security is easy to maintain. A general purpose language would require sandboxing to disallow certain operations, and ensure that illegal code does not crash the entire system. Additionally, timeouts would be necessary to prevent infinite loops and/or deadlocks from affecting the server. Expressions with no explicit looping constructs and access only to mathematical functions are inherently secure. The only necessary limit is the length of the expression, as an arbi-

trarily long expression can result in an arbitrarily large amount of work.

4 RESULTS

In this section, we analyze the performance of our streaming protocol and our query compiler.

4.1 Hardware Setup

For the evaluation, the simulation server was set up on a dual Intel Xeon E5-2650v2 server (two octa-core processors running at 2.66 GHz) with two NVIDIA GRID K2 graphics cards (4 GPUs total) and 64 GiB RAM running Ubuntu Linux 13.10. The graphical client was installed on an Intel Core i7-2600 (quad-core processor running at 3.4 GHz) desktop workstation with an NVIDIA Geforce GTX 580 GPU and 16 GiB RAM running Windows 7. For the HTML5 client, tests were additionally performed on a OnePlus One smartphone with a Qualcomm Snapdragon 801 CPU (quad-core processor running at up to 2.5 GHz) and 3 GiB RAM running Cyanogen OS 12.1 (based on Android 5.11). To cover both major mobile platforms, tests were also performed on an Apple iPhone 6S with an Apple A9 CPU (dual-core processor running at up to 1.85 GHz) and 2 GiB RAM running iOS 9.2.

4.2 Network Performance and Bandwidth Limitations

Figures 4 and 5 show the system's performance in terms of data throughput and frames per second when transmitting one, two or three fields with different network bandwidths. In this particular example, these fields were `Pressure`, `VelocityX` and `VelocityY` with a size of 1024^2 floating point values each. Bandwidth limiting was realized on the server side using Linux Traffic Control `tc`. Only outgoing bandwidth is limited, but the messages sent by the client are only tens of bytes in size and should therefore not affect the results.

Increasing the available network bandwidth also increases the client's data throughput as well as the achievable frames per seconds, as more data can be transmitted across the network. At the same time, the server's throughput and frame rate drop slightly, because more time is spent serializing messages instead of calculating new results. This decrease could be compensated by implementing double buffering and performing simulation and serialization asynchronously. However, this would lead to increased memory requirements. In all cases, the server's performance is a natural upper limit for the client that cannot be exceeded. When transmitting more than one field, this limit only becomes relevant for client-server configurations in a LAN setup with more than 1 Gbit/s. For a single field, 500 Mbit/s are sufficient to reach full performance. The fixed bandwidth limit itself is never reached, as the limit is applied at the TCP level and the effective bandwidth only includes floating point data and neither other data nor WebSocket and ProtoBuf encoding overheads.

	Time [ms]		
Number of Fields	**1**	**2**	**3**
Serialization	8.81	18.9	30.0
Native (Desktop)	7.89	14.5	20.2
Chrome (Desktop)	86.5	174	254
Firefox (Desktop)	115	221	380
Chrome (Android)	435	841	1202
Safari (iOS)	233	346	516

Table 1: Serialization and deserialization times for various platforms for a varying number of fields. Even on desktop machines, deserializing a single 1024^2 field in JavaScript takes approximately 0.1 seconds.

4.3 Serialization and Deserialization Costs

Another criterion for good performance and smooth visualization is the time required to serialize the results produced on the server and to deserialize the incoming messages on the client. Table 1 shows the serialization and deserialization costs for one, two and three fields with a size of 1024^2 floating point values per field (as in Section 4.2). Each measurement represents an average over 500 simulation steps. Note, that new frames are only transmitted to the client if the processing of the previous frame is finished. For the client, we also tested different scenarios with desktop and mobile environments. As all fields are concatenated for serialization, the required time increases linearly in all cases. While serialization and deserialization take between 7 and 30 milliseconds when using ProtoBuf in a native C++ application, performance decreases significantly when switching to browser-based applications using JavaScript. Although Chrome 47.0 outperforms Firefox 42.0, deserialization times of 86 to 254 milliseconds on a desktop workstation make it challenging to reach interactive frame rates for more than one field.

On mobile devices, deserialization times of 435 or 233 milliseconds for Chrome 46.0 and Safari 601.1, respectively, make interactive frame rates effectively impossible and raise the need to investigate alternative (de)serialization methods (see Section 6).

4.4 Query Compiler

To analyze the performance of our query compiler, compile times and average evaluation times were measured for three query expressions of varying complexity involving a varying number of results fields:

1. The absolute pressure $|p|$:
   `abs(Pressure)`

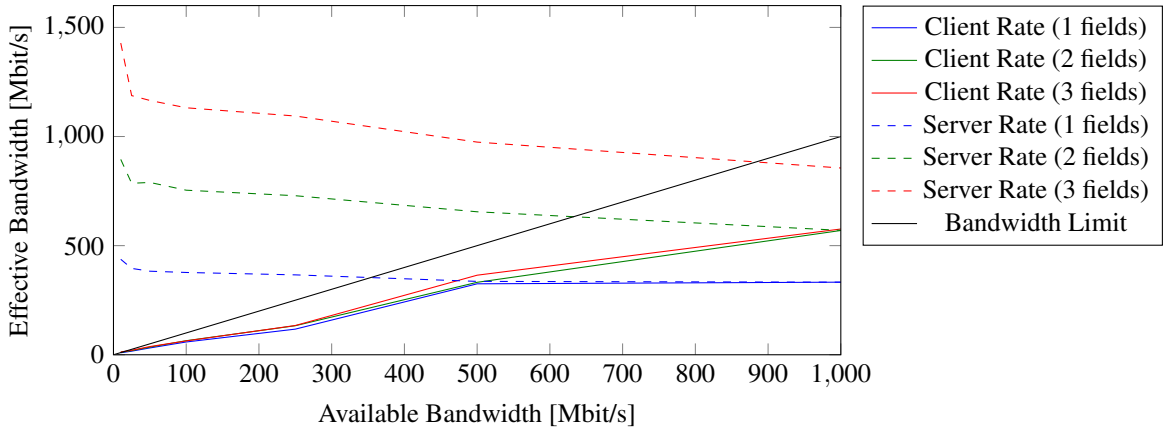

Figure 4: Effective client bandwidths when network bandwidth is limited. The rate at which the server produces data imposes an additional upper limit. This limit decreases with increasing network bandwidth as the server spends more time serializing data and is only reached for bandwidths greater than 500 Mbit/s per field.

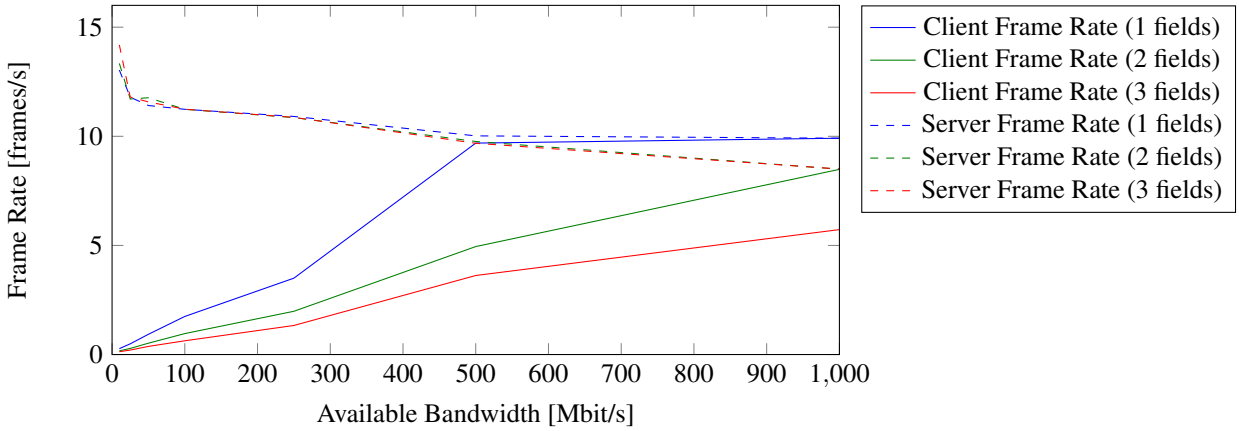

Figure 5: Client and server frame rates for several network configurations. The client frame rate increases with higher network bandwidth, as more data can be sent by the server. As in Fig. 4, the server frame rate limits the client frame rate.

2. The absolute velocity $|\vec{v}|$:
   `sqrt(VelocityX^2+VelocityY^2)`

3. The total energy density $\frac{v^2}{2} + gz + \frac{p}{\rho}$ with $g = 0$ and $\rho = 1$:
   `(VelocityX^2+VelocityY^2)/2 + Pressure`

These expressions were compiled and executed on the server described in Section 4.1. Although this set of example expressions is not exhaustive, it consists of common expressions entered by a user. The absolute value of the pressure can be of interest when the results of a compressible simulation are viewed, as the amplitude of a approximately periodic wave may be of greater interest than its absolute phase. The absolute velocity as the magnitude of a vector field is frequently required and most visualization systems include it as a built-in option. The isocontours of the total energy density are an alternative to streamlines, as according to the Bernoulli equation the total energy density must remain constant along each streamline for incompressible fluids.

All measurements in this section were performed and averaged over 80 runs of simulations on a 1024^2 grid running for 500 frames for each expression. Note that calculation is only performed for frames actually transmitted to the client and that compilation is performed once per simulation run. Therefore, the sample size for the average compilation time is 80 per expression and less than 40000 for the average calculation time.

The measured compile times are shown in Table 2. CPU compilation is completed within less than 10ms and only shows a slight increase depending on expression complexity. Although marginally slower compilation is expected due to the repetition of some optimization passes (see Sec. 3.4), GPU compilation is much slower at over 77 ms and is dominated by a constant component. Further analysis shows that 33% of that time is spent linking libdevice and 61% is spent on the final set of optimization passes. A likely reason for the signif-

	Time [ms]		
Expression	**Interp.**	**CPU**	**GPU**
Expr. 1	0.03	6.60	77.1
Expr. 2	0.04	7.22	77.1
Expr. 3	0.04	9.37	77.2

Table 2: Average compile times for the three example expressions in Sec. 4.4. The times for the interpreter only include expression parsing.

	Time [ms]		
Expression	**Interp.**	**CPU**	**GPU**
Expr. 1	14.1	8.91	4.49
Expr. 2	53.1	13.1	4.27
Expr. 3	63.1	17.1	4.38

Table 3: Average execution times for the three example expressions in Sec. 4.4.

		CPU		GPU	
Expression		**Calc.**	**Copy**	**Calc.**	**Copy**
Expr. 1	**ms**	1.99	1.04	0.10	0.98
	%	65.7	34.3	9.2	90.8
Expr. 2	**ms**	2.20	1.98	0.13	0.97
	%	52.6	47.4	11.6	88.4
Expr. 3	**ms**	1.13	3.02	0.16	0.99
	%	27.2	72.8	13.6	86.4

Table 4: Decomposition of evaluation time into calculation and GPU-CPU transfer times.

		Break-even [Frames]	
Expression		**CPU**	**GPU**
Expr. 1	**Interp.**	2 (1.27)	9 (8.02)
	CPU	—	16 (15.95)
Expr. 2	**Interp.**	1 (0.18)	2 (1.58)
	CPU	—	8 (7.91)
Expr. 3	**Interp.**	1 (0.20)	2 (1.31)
	CPU	—	6 (5.33)

Table 5: Break-even points of using CPU or GPU JIT compilation instead of an interpreter and GPU instead of CPU JIT compilation for the three example expressions in Sec. 4.4. The numbers are computed from the measurements in Tables 2 and 3 and rounded up to the nearest integer. Break-even before rounding is shown in parentheses.

icant increase in optimization time is the much larger module due to the size of libdevice.

The measured calculation times are shown in Table 3. It can be seen that the timings for the GPU version are approximately constant for all three expression, whereas for the CPU version they grow with the number of fields used in the expression. To determine the reasons for this behavior, additional measurements decomposing the total evaluation time into computation and data transfer times were performed. Table 4 shows the results of these measurements that were performed on a desktop machine equipped with an Intel Core i7-3770 CPU with 3.40 GHz and an NVIDIA GeForce GTX 580 GPU. In the case of CPU evaluation, all relevant fields have to be copied from the GPU depending on the expression used. This is reflected in the linear increase in copy times and explains the dependency seen in Table 3. For GPU evaluation, only the derived field has to be copied to system RAM. As the GPU evaluation times are dominated by the expression-independent copy component, the total evaluation time is approximately constant, as seen in Table 3 and leads to an improvement of up to 72.3% for Expr. 3.

The total time to process n simulation frames (time steps) is $t_c + nt_e$, where t_c is the compilation time, t_e is the average execution time per frame and n is the number of frames executed. Therefore the break-even between two methods a and b can be computed as $n = \left\lceil \frac{t_{c,a} - t_{c,b}}{t_{e,b} - te,a} \right\rceil$. Table 5 summarizes the different break-even points of using just-in-time (JIT) compilation instead of an interpreter. In all but the first case, the cost of compilation for CPU is amortized within the first frame, as the sum of compilation and execution time for one frame is less than the execution time for the interpreter. Due to the large compilation overhead, the break-even point of using the GPU instead of the CPU occurs significantly later. The break-even point of using the GPU instead of the CPU is reached after less than 10 frames for Expressions 2 and 3. As compilation time is independent of field size and execution time depends linearly on it, the break-even point will be reached even more quickly for larger simulation domains.

5 CONCLUSION

Using the query compiler introduced in Section 3.4, only one result field has to be sent to the client. This ensures that a high visualization frame rate can be achieved with bandwidths as low as 500 Mbit/s (see Sec. 4.2), allowing the user to view more current data. Additional latency and computation costs due to deserialization are avoided as well, making HTML5 clients feasible on desktop workstations (see Sec. 4.3). By using an optimizing compiler, server CPU and GPU times are reduced by a factor of up to 14 compared to the naïve approach of using an interpreter (see Sec. 4.4). As computation time and required bandwidth directly affect visualization latency, research question 1 *"Can compiler technologies be used to decrease visualization latencies in a remote scientific visualization system?"* can be answered positively.

The second research question *"Can GPU-based simulations running at interactive rates profit from GPU-based query compilation?"* can be confirmed as well. By computing derived expressions directly on the GPU, a significant amount of time can be saved. By only copying a single field independent of the number of fields used in the expression, the amount of data transferred over the PCIe bus can be reduced, as shown in

Sec. 4.4. Additionally, computation speed is increased by a factor of up to 20 compared to the CPU.

In summary, we have performed a detailed analysis of streaming performance and shown that optimizing compiler technologies such as LLVM can be used to significantly improve performance and reduce bandwidth costs for streaming visualization of interactive simulations. By additionally moving data transformation work to the GPU, the costs of PCIe bus transfers can be minimized as well for GPU-based simulation back-ends.

Compared to MapD (see Sec. 2.1) we have taken a similar approach of leveraging compiler technologies for visualization, but applied it to interactive scientific visualization instead of visual analytics. The range of available options is currently significantly smaller, but further enhancements are outlined in the following section.

Compared to application sharing (see Sec. 2.3) our approach of pre-transforming simulation data on the server before transmitting it to the client for final visualization has both benefits and drawbacks. Many interactions relevant during exploration of simulation results, including color map changes and panning/zooming in 2D or camera position in 3D, can now be performed without any network round trip latency using our approach. Application sharing always incurs at least one network round trip for all user interactions. However, the time to first image is potentially higher, as floating point simulation data is frequently larger than the resulting image compressed using a video codec. This also decreases the number of frames per second that can be transmitted given a limited bandwidth. This drawback can be offset by applying compression methods as well (see Sec. 2.2). Furthermore, the portion of simulation data that is transmitted could be limited to the visible part and resolution, however this limits panning/zooming or can create temporary holes in the visualization that are fixed as soon as an updated frame is received.

6 FUTURE WORK

Several potential extensions could be implemented to improve performance further or increase flexibility. Compression algorithms including those presented in [OB11] or [Lin14] can be added to further reduce bandwidth requirements at the cost of additional processing on both client and server. Queries could be extended to support subfields, i.e., named boundaries or subdomains, for instance an inlet in a CFD simulation or a specific component in a CSM simulation. Especially in combination with reductions, for example averages or maximums of fields, such subfield queries could become useful. However, parallel reductions as required by the GPU back-end require reimplementation of many scalar optimizations such as common subexpression elimination, as parallelism can not be expressed directly in LLVM-IR. Expressing parallelism in LLVM is a topic of ongoing research (see, e.g., [Kha+15]). Furthermore, calculations involving matrices and tensors would be useful for several physical domains, including CSM. Fields could also be annotated with physical units to detect mistakes due to adding fields with mismatched units.

Considering the bad JavaScript performance, alternative serialization formats promising lower deserialization costs such as Cap'n Proto [San16] or FlatBuffers [Goo16], or JavaScript's native JSON (JavaScript object notation) format could be investigated. However, these typically come at an increased bandwidth cost.

ACKNOWLEDGEMENTS

This work has been supported in part by the EU project CloudFlow (FP7-2013-NMP-ICT-FoF-609100).

REFERENCES

[Alt+16] Christian Altenhofen et al. "Rixels: Towards Secure Interactive 3D Graphics in Engineering Clouds". In: *Transactions on Internet Research (TIR)* 12.1 (Jan. 2016), pp. 31–38. ISSN: 1820-4503.

[Che+11] Kuan-Ta Chen et al. "Measuring the Latency of Cloud Gaming Systems". In: *Proceedings of the 19th ACM International Conference on Multimedia*. MM '11. Scottsdale, Arizona, USA: ACM, 2011, pp. 1269–1272. ISBN: 978-1-4503-0616-4. DOI: 10.1145/2072298.2071991.

[Chi+12] Charisee Chiw et al. "Diderot: A Parallel DSL for Image Analysis and Visualization". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '12. Beijing, China: ACM, 2012, pp. 111–120. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254079.

[Cho+14] Hyungsuk Choi et al. "Vivaldi: A Domain-Specific Language for Volume Processing and Visualization on Distributed Heterogeneous Systems". In: *Visualization and Computer Graphics, IEEE Transactions on* 20.12 (Dec. 2014), pp. 2407–2416. ISSN: 1077-2626. DOI: 10.1109/TVCG.2014.2346322.

[Duk+09] D.J. Duke et al. "Huge Data But Small Programs: Visualization Design via Multiple Embedded DSLs". English. In: *Practical Aspects of Declarative Languages*. Ed. by Andy Gill and Terrance Swift. Vol. 5418. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 31–45. ISBN: 978-3-540-92994-9. DOI: 10.1007/978-3-540-92995-6_3.

[Goo08] Google. *Protocol Buffers (protobuf)*. Website, retrieved 2016-03-14. 2008. URL: https://github.com/google/protobuf.

[Goo16] Google. *Flatbuffers*. Website, retrieved 2016-03-14. 2016. URL: https://github.com/google/flatbuffers.

[Gri13] Ilya Grigorik. *High Performance Browser Networking*. O'Reilly Media, 2013. ISBN: 978-1-4493-4476-4.

[Kes15] John Kessenich, ed. *SPIR-V Specification*. Version 1.00, Rev. 2. Khronos Group, Nov. 2015. URL: https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.pdf.

[Kha+15] Dounia Khaldi et al. "LLVM Parallel Intermediate Representation: Design and Evaluation using OpenSHMEM Communications". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. ACM, Nov. 2015, 2:1–2:8. DOI: 10.1145/2833157.2833158.

[LA04] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. Mar. 2004, pp. 75–88. DOI: 10.1109/CGO.2004.1281665.

[Lin14] P. Lindstrom. "Fixed-Rate Compressed Floating-Point Arrays". In: *Visualization and Computer Graphics, IEEE Transactions on* 20.12 (Dec. 2014), pp. 2674–2683. ISSN: 1077-2626. DOI: 10.1109/TVCG.2014.2346458.

[Map16] MapD Technologies, Inc. *MapD*. Website, retrieved 2016-03-14. 2016. URL: http://www.mapd.com/.

[Mic16] Microsoft. *Remote Desktop Protocol*. Website, retrieved 2016-03-14. 2016. URL: https://msdn.microsoft.com/en-us/library/aa383015(VS.85).aspx.

[MŞ15] Todd Mostak and Alex Şuhan. *MapD: Massive Throughput Database Queries with LLVM on GPUs*. Website, retrieved 2016-03-14. June 2015. URL: https://devblogs.nvidia.com/parallelforall/mapd-massive-throughput-database-queries-llvm-gpus.

[Nic+08] John Nickolls et al. "Scalable Parallel Programming with CUDA". In: *ACM Queue* 6.2 (Mar. 2008), pp. 40–53. ISSN: 1542-7730. DOI: 10.1145/1365490.1365500.

[NVI16] NVIDIA. *CUDA LLVM Compiler*. Website, retrieved 2016-03-14. 2016. URL: https://developer.nvidia.com/cuda-llvm-compiler.

[OB11] Molly A. O'Neil and Martin Burtscher. "Floating-point Data Compression at 75 Gb/s on a GPU". In: *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. GPGPU-4. Newport Beach, California, USA: ACM, 2011, 7:1–7:7. ISBN: 978-1-4503-0569-3. DOI: 10.1145/1964179.1964189.

[Pey03] Simon Peyton Jones, ed. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. ISBN: 978-0-521-82614-3.

[Rau+14] P. Rautek et al. "ViSlang: A System for Interpreted Domain-Specific Languages for Scientific Visualization". In: *Visualization and Computer Graphics, IEEE Transactions on* 20.12 (Dec. 2014), pp. 2388–2396. ISSN: 1077-2626. DOI: 10.1109/TVCG.2014.2346318.

[Ric+98] Tristan Richardson et al. "Virtual network computing". In: *IEEE Internet Computing* 2.1 (Jan. 1998), pp. 33–38. DOI: 10.1109/4236.656066.

[RKB06] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. "Fast lossless compression of scientific floating-point data". In: *Proceedings of the Data Compression Conference*. DCC '06. 2006, pp. 133–142. DOI: 10.1109/DCC.2006.35.

[San16] Sandstorm.io. *Cap'n Proto*. Website, retrieved 2016-03-14. 2016. URL: https://capnproto.org/.

[TAS06] N. Tolia, D.G. Andersen, and M. Satyanarayanan. "Quantifying interactive user experience on thin clients". In: *Computer* 39.3 (Mar. 2006), pp. 46–52. ISSN: 0018-9162. DOI: 10.1109/MC.2006.101.

[Web+15] Daniel Weber et al. "A Cut-Cell Geometric Multigrid Poisson Solver for Fluid Simulation". In: *Computer Graphics Forum* 34.2 (May 2015), pp. 481–491. ISSN: 0167-7055. DOI: 10.1111/cgf.12577.

[Yax15] Liu Yaxun. *[RFC] Proposal for Adding SPIRV Target*. Website, retrieved 2016-03-14. June 2015. URL: http://lists.llvm.org/pipermail/llvm-dev/2015-June/086848.html.