Eurographics Symposium on Parallel Graphics and Visualization (2021)
M. Hadwiger, M. Larsen, F. Sadlo (Editors)

Performance Tradeoffs in Shared-memory Platform Portable Implementations of a Stencil Kernel

E. Wes Bethel[1,2], Colleen Heinemann[1,3], and Talita Perciano[1]

[1]Lawrence Berkeley National Laboratory, Berkeley CA, USA
[2]San Francisco State University, San Francisco CA, USA
[3]University of Illinois, Urbana-Champaign IL, USA

Abstract

Building on a significant amount of current research that examines the idea of platform-portable parallel code across different types of processor families, this work focuses on two sets of related questions. First, using a performance analysis methodology that leverages multiple metrics including hardware performance counters and elapsed time on both CPU and GPU platforms, we examine the performance differences that arise when using two common platform portable parallel programming approaches, namely OpenMP and VTK-m, for a stencil-based computation, which serves as a proxy for many different types of computations in visualization and analytics. Second, we explore the performance differences that result when using coarser- and finer-grained parallelism approaches that are afforded by both OpenMP and VTK-m.

CCS Concepts
• ***Computing methodologies*** $\rightarrow$ ***Parallel programming languages;*** • ***Theory of computation*** $\rightarrow$ ***Shared memory algorithms;***

1. Introduction

As computational platforms become increasingly heterogeneous, consisting of multi-core CPUs and accelerators like many-core GPUs, an area of much recent research focuses on the idea of software tools and programming environments to achieve platform portable parallelism across different types processors. While there are multiple design and implementation approaches for platform-portable code, the performance tradeoffs between these approaches is not always clear, nor are the performance tradeoffs within a single approach where one may have multiple design strategies.

For example, OpenMP recently added support for GPU-offload of OpenMP-parallel codes [Li20], and projects like VTK-m [M*19] provide support for backends that run on multiple devices, including multi-core CPUs and many-core GPUs. In both of these examples, there are numerous ways to implement a particular algorithm that produces the correct answer and that runs in parallel.

In this work, we examine the performance that results when exploring multiple design pathways in multiple platform-portable, shared-memory parallel programming frameworks, namely OpenMP and VTK-m. The design trade-offs focus primarily on coarser- versus finer-grained parallelism, and the study results reveal findings about cost of overhead associated with platform portability. We use a methodology for evaluating performance that goes well beyond runtime alone to examine hardware performance counters on CPU and GPU platforms. This study is timely given the increasing emphasis on platform portability in heterogeneous computing environments.

The primary contributions and novelty of this work are: (1) a performance comparison of a key computational kernel that is widely used in analysis and visualization implemented using two different methods for achieving platform portable parallel code, namely OpenMP and VTK-m; (2) examination of performance differences that result when using coarse- and fine-grained parallelism design options in both frameworks including the potential cost of overhead associated with platform portability; (3) a description of two different VTK-m implementations of a stencil-based convolution kernel.

2. Design and Implementation: Shared-memory Parallel Stencl-based Convolution

2.1. Overview

In a stencil based computation, each point of a multidimensional grid is updated with contributions from its neighbors. This form of computation lies at the heart of many different types of scientific computations, such as solving partial differential equations on a regular, structured grid (c.f., [RRMc*97]). We include this computational pattern because it is common in many types of computing applications, such as numerical simulation, image analysis/computer vision, convolutional neural networks, as well as visualiza-

DOI: 10.2312/pgv.20211043

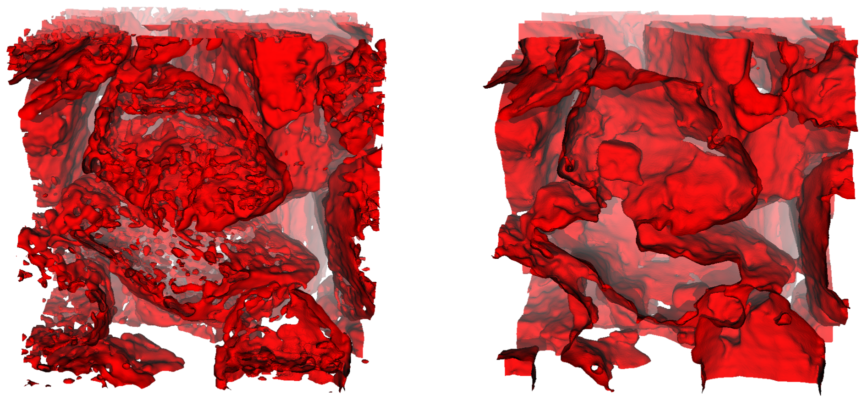

Figure 1: *The original data (left), obtained via computed tomography at the Advanced Light Source, is a 3D image of a sandstone sample, and appears "grainy" or "noisy" when visualized using an isocontour. Application of the 3D version of the Gaussian smoothing kernel with a large stencil size, 19^3, produces an isocontour that is much smoother (right), and closer to the results we would obtain using high quality image segmentation methods (c.f., [PHC*20].) Such methods are an important part of the scientific data analysis workflow.*

tion methods like isocontouring that perform computations using stencils of mesh points.

In our study, we focus on a particular type of stencil-based computation, namely convolution, using Gaussian weights, which is often used for image smoothing and noise reduction. This form of computation is a structured memory access code, where memory is accessed in a regular and predictable fashion.

In this computation, each destination pixel $d(i)$ is a sum of nearby pixels averaged using a weighting scheme that gives more weight to pixels closer to i, and less pixels further away (Eq. 1).

$$d(i) = \sum g(i,\bar{i}) \tag{1}$$

where the Gaussian weights are given by

$$g(i,\bar{i}) = e^{-\frac{1}{2}\left(\frac{\delta(i,\bar{i})}{\sigma_d}\right)^2} \tag{2}$$

In Eq. 2, $\delta(i,\bar{i})$ is the distance between pixels i and $\bar{i}$. σ is a parameter that defines whether the weights are more tightly focused around the source pixel (somewhat less smoothing), or if the weights are more diffuse, and give more weight to pixels further away (somewhat more smoothing). Computation of the smoothed pixel value consists of performing a sum of products of the source pixel weights with the filter weights. An example for 2D image convolution appears in Listing 1, and this computation trivially generalizes to N dimensions.

We focus on this form of implementation in this paper because it is well understood and has been well studied elsewhere and in different contexts (c.f. [RYQ11]). The study of alternative formulations, such as the separable convolution [Cho17], which accomplish the same type of task but with fewer arithmetic operations, would make for interesting future work. Our focus here is on the potential overhead of programming in different environments and using different coarse- vs. fine-parallelism tradeoffs with a well understood and well recognized algorithm.

An application that uses the `smoothPixel` method in Listing 1 will iterate over the pixels/voxels in a source image/volume, and invoke this method at each pixel/voxel. This type of application is straightforward to parallelize in that the sense that the computation at each $d(i)$ is independent of the computation at all other locations; these computations may be performed independently and in parallel.

2.2. Parallelization with OpenMP

Listing 2 shows two such parallelizations using OpenMP: one is more coarse-grained, while the other is more fine-grained. In the coarse-grained parallelization, the loop parallelization occurs over scanlines: eath OpenMP thread will be assigned one scanline's worth of pixels to process. OpenMP does assignments using one of several different strategies, such as round-robin, etc. depending on the setting of a runtime environment variable (c.f. [CMD*00]). This type of parallelization is relatively coarse-grained in the sense that each thread is assigned a significant amount of work.

In the fine-grained approach, shown in the lower part of Listing 2, the loop parallelization occurs over the entire collection of pixels. In contrast to the coarse-grained parallel example, this fine-grained approach has one pixel assigned to each thread.

2.3. Parallelization with VTK-m

One potential VTK-m implementation of this stencil operation is to use the same computational kernel shown in Listing 1, but then let VTK-m manage how this computation is invoked. To do so, we define a *Field Map worklet* that has input and output parameters that are "arrays"; these are essentially `std::vector` objects. Then, after populating the input array with the source image, we invoke the *dispatcher* that invokes the `ImageConvolutionWorklet`, as shown in Listing 3, which is an abbreviated summary of the worklet. Then, VTK-m will invoke that worklet once per array item using one of several different potential device backends, depending upon user build configuration options and runtime choices (see the VTK-m User's Manual for more details [M*19]). In the case of our application, this worklet is invoked once per input pixel.

In this case, the per-pixel invocation of the Field Map worklet is like the fine-grained OpenMP parallelism approach shown in Listing 2. We refer to this design as the VTK-m-*FM* method, where *FM* refers to the use of the *Field Map Worklet*, and which uses an explicit indexing computation to access specific locations in the input and output image data arrays. Note that in principle it is possible in VTK-m to implement a coarser-grained parallelism design, but doing so would go "against the grain" of VTK-m's design principles.

There are other potential implementations of this stencil that we could pursue in VTK-m, implementations that could potentially take greater advantage of VTK-m's data parallel primitives (DPPs). These DPPs include operations like Reduce, Sort, CopyIf, ScanExclusiveByKey, and so forth. Some interesting future work would entail exploring recasting the stencil computation in terms of using VTK-m's DPP Device Algorithms, as has been done with other types of computations, such as probabilistic graphical modeling optimization [LPH*18].

For this study, we explore a second VTK-m implementation, namely one that uses a `Point Neighborhood Worklet`, which we refer to as the VTK-m-*PN* algorithm. Like the VTK-m-*FM* algorithm, VTK-m invokes the worklet once per input grid location. One significant difference is that the VTK-m-*PN* algorithm may access field values of nearby points within a neighborhood of a given size, as opposed to having access to the entire mesh.

In other words, the VTK-m-*FM* method needs to do its own indexing: it is handed a 1D input index, and then needs to use this input to produce an index into a multidimensional array. In contrast, methods that use the `Point Neighborhood Worklet` will use an `inputData.Get()` method to access data, rather than using an index computation. Due to space limitations, we show only an abbreviate listing of the VTK-m-*PN* code in Listing 4. In both cases, VTK-m is performing extra work to manage worklet execution, work is the overhead associated with its platform-portable parallel capabilities.

3. Results

Our primary research objective is to identify performance differences of alternative implementations of a convolution kernel using two different platform-portable parallel coding environments on two different platforms: GPU and GPU, and to use hardware performance counters to help understand the nature of the performance differences.

3.1. Methodology

Computational Platforms. All tests were conducted on Cori at NERSC on specialized GPU nodes. Each node consists of two 2.40 GHz Intel Xeon Gold 6148 (Skylake) processors with 384 GB of DDR4 memory, and 8 NVIDIA Tesla V100 (Volta) GPUs, each with 16 GB HBM2 memory [Nat21]. Our OpenMP-CPU tests were run on these Skylake processors, and our Cuda and OpenMP-offload tests were run on the Volta GPUs.

Software Environment. For the VTK-m configurations, we make use of VTK-m v1.5.0 [Mor21]. For the OpenMP on Skylake tests, we used Intel's `icc` compiler version 19.1.2.254, and used the following flags to enable vectorization: `-O3 -march=skylake -mtune=skylake -DNDEBUG -funroll-loops`. For the VTK-m tests with a Cuda backend, we used `gcc/g++` 7.5.0 and Cuda 11.1. For the OpenMP-offload tests with the Cuda backend, we used `clang` 12.0.0.

Dataset and Algorithmic Parameters. We are using a scientific dataset that was obtained by the Lawrence Berkeley National Laboratory Advanced Light Source X-ray beamline 8.3.2 [D*15]. This dataset contains cross-sections of a geological sample and conveys information regarding the x-ray attenuation and density of the scanned material as a gray scale value. The original data consists of a stack of 500 images at resolution 1290×1305. The images in Fig. 1 are created with a subset of this data, and the performance tests use an augmented version of a single 2D slice that is approximately $5K^2$ in size.

Performance Measures and Tools. For this study, we are leveraging the LIKWID software infrastructure v5.0.0 [THW10, TG20]. LIKWID is a set of lightweight, command-line tools that are useful for obtaining measurements of hardware performance counters on Linux platforms in user space. On the GPU, we use `nvprof`, which is part of the Cuda 11.2 distribution.

Testing Procedures. On the CPU platform, we measure and compare elapsed runtime in seconds, vectorization level, number of instructions executed, L3 and L2 cache miss rate, and Cycles Per Instruction (CPI), which is a derived metric computed as the quotient of CPU_CLK_UNHALTED_CORE / INSTR_RETIRED_ANY to give an estimate of the number of clock cycles per instruction (CPI) (c.f. [PH14]). On the GPU platform, we measure and compare elapsed runtime in milliseconds, a count of instructions computed as the average per warp, and global_hit_rate, the L1 cache hit rate for global memory loads.

3.2. Discussion of Results

When looking at the CPU performance data in Tab. 1, we see a comparison of coarse- and fine-grained parallelism, along with a comparison to a serial C++ run. Here, we see a C++ code with OMP loop parallelism, both coarse- and fine-grained, along with two different VTK-m implementations. For this problem, we see similar measures of runtime, instruction count, vectorization level, and memory utilization for the C++/OpenMP coarse and fine, and the VTK-m-*FM* implementation. Looking at the number of instructions executed and compared to the serial version, both coarse and fine OpenMP implementations and the VTK-m-*FM* implementation have similar levels of overhead, about 15%, 18%, and 22%, respectively, that is attributable to the parallel runtime environment associated with each parallel coding approach. The VTK-m-*PN* implementation executes about 20 times as many instructions, a significant amount of the overhead associated with VTK-m's *Point Neighborhood* worklet in this case.

In the GPU performance data in Tab. 2, we see both the coarse-

and fine-grained OpenMP-offload methods have nearly identical levels of performance in terms of runtime, number of instructions executed, and global_hit_rate, which measures the L1 Cache hit rate. Both these versions are are approximately 20 times faster than the serial CPU version. The question of whether or not faster runtime is possible through additional OpenMP optimization, or via different environments like OpenACC would make for interesting future work.

In contrast, the VTK-m-*FM* method is executing significantly more instructions, which is reflected in a much larger runtime. Since this code uses essentially the exact same code for performing the stencil computation as the C++/OpenMP code, but is invoked from within a VTK-m worklet, the increased number of instructions is attributable to the cost of VTK-m overhead for mapping the worklet onto warp threads. The VTK-m-*PN* method appears to use even more overhead for simplifying access to field data by the *Point Neighborhood* worklet code, as access to the input field goes through a VTK-m `getData()` function rather than array indexing. This function-brokered data access mechanism is why the vectorization level of the VTK-m-*PN* method on the CPU is 0%.

4. Related Work

In response to the end of Dennard scaling, system and processor architectures have evolved to use deepening memory hierarchies combined with increasing node-level parallelism [EBA*11]. Over the past decade or so, numerous programming models and environments — such as OpenMP [CMD*00], OpenCL [GHK*11], Kokkos [ETS14], and OpenACC [Ope21], to name a few — have emerged where the objective is to provide for platform portability as well as efficient shared-memory parallelism.

In the visualization community, the VTK-m library [MSU*16] follows a similar technology trajectory, where user code can be executed with one of several different device- or platform-level backends, such as TBB [Rei07], OpenMP [Li20], or CUDA [Nvi18]. VTK-m is positioned to be a follow-on to the VTK library [SML98], which has served as a stalwart in the visualization community for more than two decades but is limited to predominantly serial use due to a combination of factors, including close entanglement of data structures and execution models and use of static variables that maintain state and are hence not thread-safe.

Given the community interest in VTK-m, our work here seeks to provide more insight into the performance differences that result when implementing algorithms using OpenMP and VTK-m, with an eye towards understanding the performance differences that result in different algorithmic design patterns and on different platforms.

There have been several recent works that look at comparing the performance of traditional methods and those implemented in VTK-m. These include raytracing (Larsen, et al. 2015 [LMNC15]), particle advection (Pugmire, et al. 2018 [PYK*18]), and graph analytics (Lessley, et al., 2017 [LPM*17]), to name a few. One thing these have in common is use of runtime, or its derivative, as the lone metric. The work here differs in that it includes the use of hardware performance counters, considers coarse- and fine-grained parallelism on both CPU and GPU platforms, and makes use of OpenMP device offload for implementing a GPU-based version of a staple analysis kernel, a stencil-based computation.

	Serial	Coarse	Fine	VTK-m-*FM*	VTK-m-*PN*
	C++	OMP	OMP	OMP	OMP
Runtime (sec)	124.28	16.46	16.34	16.18	526.50
Inst. exec. 10^9	468.52	540.57	553.67	571.88	10459.88
FLOPS Scalar 10^9	30.78	35.26	35.35	35.26	639.82
FLOPS Vector 10^9	193.07	221.23	221.76	221.23	0.04
Vectorization %	86.24	86.25	86.25	86.25	0.00
CPI	0.6	0.71	0.69	0.68	1.3
L3 miss %	0.95	0.1	0.29	0.31	0.25
L2 miss %	0.01	0.01	0.01	0.01	0.02

Table 1: *Results of runs on the Intel Skylake CPU comparing multiple coarse- and fine-grained parallel implementations. All four sets of OpenMP runs are performed at 8-way concurrency.*

	Coarse	Fine	VTK-m-*FM*	VTK-m-*PN*
	OMP/CUDA	OMP/CUDA	CUDA	CUDA
Runtime (ms)	5.54	5.56	393.41	734.72
Inst. exec. 10^5	124.79	124.79	47244	183000
global_hit_rate%	44.32	44.32	97.72	99.09

Table 2: *Results of runs on the NVIDIA Volta GPU comparing coarse- and fine-grained parallel implementations.*

There is a long history of performance analysis of stencil-based codes, such as [RYQ11] and many others, including separable convolution approaches [Cho17]. Our focus here is on comparing performance of this staple algorithm in two specific platform-portable parallel coding environments, OpenMP and VTK-m that are of significant interest to the HPC visualization community.

5. Conclusion and Future Work

This study sheds light on the performance characteristics of a key computational motif, a stencil-based computation, in light of the trade-offs that occur in multiple software environments that implement platform portability in conditions where we vary coarse- vs fine-grained parallelism. Due to growing interest in platform portable approaches for heterogeneous computing, combined with significant development effort being applied towards both OpenMP and VTK-m for such uses, this study's results are timely.

The study's findings reveal there is a measurable cost of overhead associated with VTK-m's platform portability in this particular problem using these particular formulations. On the CPU, this cost is clearly visible for the VTK-m-*FM* implementation. On the GPU, these costs are clearly visible for both VTK-m-*FM* and VTK-m-*PN* implementations. The fact there is an overhead cost associated with platform portability is not unexpected, but the magnitude of the costs, depending on platform and VTK-m worklet, is somewhat surprising and opens the door for further investigations both in terms of the underlying mechanism VTK-m uses to decompose a large problem into smaller worklet-sized chunks as well as to application to other computational motifs and platform-portable coding environments.

References

[Cho17] CHOLLET F.: Xception: Deep learning with depthwise separable convolutions. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), pp. 1800–1807. doi:10.1109/CVPR.2017.195. 2, 4

[CMD*00] CHANDRA R., MENON R., DAGUM L., KOHR D., MAYDAN D., MCDONALD J.: *Parallel Programming in OpenMP*. Elsevier Science, 2000. URL: https://books.google.com/books?id=pv-jXf7oGtAC. 2, 4

[D*15] DONATELLI J., ET AL.: Camera: The center for advanced mathematics for energy research applications. *Synchrotron Radiation News 28*, 2 (2015), 4–9. 3

[EBA*11] ESMAEILZADEH H., BLEM E., AMANT R. S., SANKARALINGAM K., BURGER D.: Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)* (June 2011), pp. 365–376. 4

[ETS14] EDWARDS H. C., TROTT C. R., SUNDERLAND D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing 74*, 12 (2014), 3202 – 3216. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing. URL: http://www.sciencedirect.com/science/article/pii/S0743731514001257, doi:https://doi.org/10.1016/j.jpdc.2014.07.003. 4

[GHK*11] GASTER B., HOWES L., KAELI D. R., MISTRY P., SCHAA D.: *Heterogeneous Computing with OpenCL*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011. 4

[Li20] LI K.: OpenMP Accelerator Support for GPUs, 2020. OpenMP ARB website, /urlhttps://www.openmp.org/updates/openmp-accelerator-support-gpus/, last accessed March 2020. 1, 4

[LMNC15] LARSEN M., MEREDITH J., NAVRATIL P., CHILDS H.: Ray tracing within a data parallel framework. In *2015 IEEE Pacific Visualization Symposium, PacificVis 2015 - Proceedings* (United States, 7 2015), Takahashi S., Liu S., Scheuermann G., (Eds.), IEEE Pacific Visualization Symposium, IEEE Computer Society, pp. 279–286. doi:10.1109/PACIFICVIS.2015.7156388. 4

[LPH*18] LESSLEY B., PERCIANO T., HEINEMANN C., CAMP D., , CHILDS H., BETHEL E. W.: DPP-PMRF: Rethinking Optimization for a Probabilistic Graphical Model Using Data-Parallel Primitives. In *8th IEEE Symposium on Large Data Analysis and Visualization (LDAV)* (Berlin, Germany, Oct. 2018). 3

[LPM*17] LESSLEY B., PERCIANO T., MATHAI M., CHILDS H., BETHEL E. W.: Maximal Clique Enumeration with Data Parallel Primitives. In *7th IEEE Symposium on Large Data Analysis and Visualization (LDAV)* (Phoenix, AZ, USA, Oct. 2017). doi:10.1109/LDAV.2017.8231847. 4

[M*19] MORELAND K., ET AL.: The vtk-m user's guide, vtk-m version 1.5, Oct. 2019. http://m.vtk.org/images/c/c9/VTKmUsersGuide-1-5.pdf. 1, 3

[Mor21] MORELAND K.: VTK-m Releases, 2021. Online at https://m.vtk.org/index.php/VTK-m_Releases#VTK-m_Version_1.5.0, last access Mar 2021. 3

[MSU*16] MORELAND K., SEWELL C., USHER W., LO L., MEREDITH J., PUGMIRE D., KRESS J., SCHROOTS H., MA K.-L., CHILDS H., LARSEN M., CHEN C.-M., MAYNARD R., GEVECI B.: VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications (CG&A) 36*, 3 (May/June 2016), 48–58. 4

[Nat21] NATIONAL ENERGY RESEARCH SCIENTIFIC COMPUTING CENTER: Cori GPU Nodes Hardware Information, 2021. Online at https://docs-dev.nersc.gov/cgpu/hardware/, last accessed March 2021. 3

[Nvi18] NVIDIA CORPORATION: CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, June 2018. 4

[Ope21] OpenACC Website, Apr. 2021. Online at https://www.openacc.org/, last accessed Apr 2021. 4

[PH14] PATTERSON D. A., HENNESSY J. L.: *Computer Organization and Design - The Hardware / Software Interface (5th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2014. 3

[PHC*20] PERCIANO T., HEINEMANN C., CAMP D., LESSLEY B., BETHEL E. W.: Shared-memory parallel probabilistic graphical modeling optimization: Comparison of threads, openmp, and data-parallel primitives. In *ISC 2020, Springer LNCS* (June 2020), ISC 2020. (To appear). 2

[PYK*18] PUGMIRE D., YENPURE A., KIM M., KRESS J., MAYNARD R., CHILDS H., HENTSCHEL B.: Performance-Portable Particle Advection with VTK-m. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)* (Brno, Czech Republic, June 2018), pp. 45–55. 4

[Rei07] REINDERS J.: *Intel Threading Building Blocks*, first ed. O'Reilly & Associates, Inc., USA, 2007. 4

[RRMc*97] ROTH G., ROTH G., MELLOR-CRUMMEY J., MELLOR-CRUMMEY J., KENNEDY K., KENNEDY K., BRICKNER R. G., BRICKNER R. G.: Compiling stencils in high performance fortran. In *In Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM* (1997), ACM Press, pp. 1–20. 1

[RYQ11] RAHMAN S. M. F., YI Q., QASEM A.: Understanding stencil code performance on multicore architectures. In *Proceedings of the 8th ACM International Conference on Computing Frontiers* (New York, NY, USA, 2011), CF '11, Association for Computing Machinery. URL: https://doi.org/10.1145/2016604.2016641, doi:10.1145/2016604.2016641. 2, 4

[SML98] SCHROEDER W., MARTIN K. M., LORENSEN W. E.: *The Visualization Toolkit (2nd Ed.): An Object-Oriented Approach to 3D Graphics*. Prentice-Hall, Inc., USA, 1998. 4

[TG20] THOMAS GRUBER E. A.: likwid-perfctr: Measuring applications' interaction with the hardware using the hardware performance counters, Jan. 2020. Web location: https://github.com/RRZE-HPC/likwid/wiki/likwid-perfctr, last accessed October 2019. 3

[THW10] TREIBIG J., HAGER G., WELLEIN G.: Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures* (San Diego CA, 2010). 3