Prediction of Distributed Volume Visualization Performance to Support Render Hardware Acquisition

G.Tkachev, S.Frey, C.Müller, V.Bruder, T.Ertl

University of Stuttgart

Abstract

We present our data-driven, neural network-based approach to predicting the performance of a distributed GPU volume renderer for supporting cluster equipment acquisition. On the basis of timing measurements from a single cluster as well as from individual GPUs, we are able to predict the performance gain of upgrading an existing cluster with additional or faster GPUs, or even purchasing of a new cluster with a comparable network configuration. To achieve this, we employ neural networks to capture complex performance characteristics. However, merely relying on them for the prediction would require the collection of training data on multiple clusters with different hardware, which is impractical in most cases. Therefore, we propose a two-level approach to prediction, distinguishing between node and cluster level. On the node level, we generate performance histograms on individual nodes to capture local rendering performance. These performance histograms are then used to emulate the performance of different rendering hardware for cluster-level measurement runs. Crucially, this variety allows the neural network to capture the compositing performance of a cluster separately from the rendering performance on individual nodes. Therefore, we just need a performance histogram of the GPU of interest to generate a prediction. We demonstrate the utility of our approach using different cluster configurations as well as a range of image and volume resolutions.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing algorithms I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics

1. Introduction

Volume rendering is a powerful method for visualization of three-dimensional data obtained through measurements or simulation. It is therefore one of the most important techniques in scientific visualization. Volume rendering is not only computationally expensive, but also an embarrassingly parallel problem. Therefore, it is often solved on (multi-layered) parallel architectures such as GPU clusters. In such scenarios, many different factors influence rendering performance, which makes the prediction of the overall rendering performance a challenging task. However, performance prediction has a variety of useful applications. First, it supports decisions made during systems design: a performance model can be quickly used to estimate how hypothetical changes to hardware or software will affect the performance of an application, without implementing them. Second, it can be used for recognizing the need and driving the optimization process of a newly created application. The implementation of parallel applications is a difficult, error-prone task, and an existing model may help find issues in application performance. Finally, a performance model is useful during equipment procurement, allowing potential performance gains to be estimated before purchasing expensive hardware, helping to achieve an optimal performance-to-price ratio.

While there is a significant amount of research on performance prediction, visual computing applications have a set of special, challenging properties. Volume rendering, on which we focus in this work, is an inherently interactive technique, with user input having a large impact on the load balance and, ultimately, performance. Approaches assuming stable and repetitive performance (like in many HPC codes) are inapplicable to this case, warranting specialized performance models. Furthermore, visual computing applications use highly parallel discrete graphics hardware, complicating the overall prediction problem with an additional node-local level of parallelism and transfer bottlenecks, both in each GPU and between GPUs of a node. These properties make predictions significantly more complex. While machine learning-based approaches are widely used to capture the properties of complex systems, a general model of cluster performance should ideally be trained using data from a wide range of possible variations. In our target application scenario dealing with render hardware upgrades for GPU clusters, this means including measurements from reasonable combinations of different sizes of clusters with varying node hardware (CPUs, etc.), different number and types of GPUs, and different network interconnects. However, interchanging node hardware of the whole cluster to perform each

set of measurements is impractical in most cases as it would require an enormous amount of equipment and effort.

To address this, we propose a two-level approach to predicting parallel volume rendering performance based on artificial neural networks (ANNs). Neural networks are a flexible data-driven tool that can be used without manually putting application and hardware-specific knowledge into the model. For training, we propose generation of performance histograms, which allow for acquiring more training data from a single cluster. These can be used to emulate different node hardware by simply stalling the nodes during local rendering for a predefined amount of time. This way, we basically decouple local rendering (which is practically limited by GPU performance) and compositing (which is dominated by the network speed). By changing local render time, we can measure its effect on the overall cluster performance, and collect the data needed to train a neural network capable of predicting it.

In this paper, after a discussion of related work in Sec. 2, we discuss what we consider to be the main contributions of our paper (i) A two-level approach to predicting the performance of an upgraded cluster for distributed volume rendering (Sec. 3), (ii) the use of performance histograms to emulate local rendering performance (Sec. 4) and (iii) of neural networks to capture cluster-level performance on the basis of these performance histograms (Sec. 5). We (iv) demonstrate that our approach adequately predicts the performance of an upgraded cluster (Sec. 6), and discuss its properties, current limitations and future work (Sec. 7).

2. Related work

Volume visualization. An overview of the current state of the art in GPU techniques for interactive large-scale volume visualization was given by Beyer et al. [BHP15]. Popularly, parallel rendering is classified into three classes: sort-first, sort-middle and sort-last [MCEF94]. In our work, we use latter, i. e. we parallelize over volume data (in object space) by having each GPU create a local image from its own data. In a second step, local results are composited into yield the final image. The two phases differ in that the first one can be done completely independently, while compositing is done collectively and begins only after finishing all rendering tasks.

Rendering performance of large scale systems has been the subject of several studies in the past. Petarka et al. [PYR*08] implemented, tested and analyzed performance of parallel volume rendering on an IBM Blue Gene/P, while Howison et al. [HBC12] investigated the benefits of a hybrid parallel approach to volume raycasting in their work. Focusing on small to medium scale GPU visualization clusters (similar to the one used in this work), Müller et al. [MSE06] as well as Fogal et al. [FCS*10] (among others) presented respective performance characteristics for volume rendering.

Performance prediction. Assessment, modeling and prediction of application performance in distributed environments is an active field of research in high-performance computing. Different approaches for performance modeling have been proposed in recent years. Those include performance skeletons [SSX08], regression [BRL*08], micro benchmarks [EB16] and neural networks [SIM*07]. We use neural networks as a component in our prediction

model. Ipek et al. [IdSSM05] also predicted performance of a large scale parallel application applying a multilayer neural network. They trained their model with data from executions on the target platform, to capture full system complexity. Lee et al. [LBdS*07] extended the neural network approach with additional statistical techniques for preliminary data analysis and added a comparison to a piece-wise polynomial regression approach in their work. They both applied their models on well known HPC benchmarks, namely SMG2000, a semicoarsening multigrid solver [BFJ00], and High-Performance LINPACK [Pet04]. However, those benchmarks only consider CPUs and have significantly different characteristics in comparison to distributed volume rendering. Unfortunately, CPU-focused techniques are typically inadequate for modeling GPU performance. In contrast, we not only consider GPUs, but also crucially have a different focus on predicting performance for hardware upgrades.

GPU performance models for guiding application optimization have been proposed by Baghsorkhi et al. [BDP*10]. They developed a compiler-based approach for analyzing GPU kernel code and modeling its performance. Zhang and Owens [ZO11] took a different approach but with a similar goal. They utilize micro-benchmarks to accurately measure various aspects of GPU performance, using the results to construct a performance model. Artificial neural networks for performance and power prediction of GPGPU applications were applied by Wu et al. [WGL*15]. They formed a collection of representative scaling behaviors by employing k-means clustering. Using neural network classifiers, they mapped those scaling behaviors to performance counter values. However, their performance estimation model was designed to predict how applications scale as a GPU configuration changes, i.e. their main objective as well as the scope differs from ours (in that they focus on single GPU performance).

In particular for parallel volume rendering, Rizzio et al. [RHI*14] constructed an analytical model for predicting of the scaling behavior on GPU clusters, separately considering different rendering phases. Larsen et al. [LHK*16] developed a performance model for rasterization, ray tracing and volume rendering algorithms in the context of in-situ visualization. They constructed an analytical model for the performance of every application being executed on a single machine, and used statistical methods to determine constants. Then the authors extended the model to parallel execution by introducing a similar model for image compositing performance. Their model is "semi-empirical" [HGKS11], i. e. a combination of empirical measurements (e. g., execution times) and an analytical performance model. In contrast, we take a solely empirical approach, training a neural network and intentionally abstracting away hardware and application-specific details from the model. This has the benefit of implicitly capturing the interplay between application and hardware, without the need for manually adapting the model to a given scenario. Also, we can faster adapt to significant hardware changes via (automatic) training rather than re-modeling. Most importantly, our objective differs in that we focus on hardware procurement as our main use case rather than the question of feasibility in the context of in-situ rendering.

3. Overview

Objective. The main objective of the approach presented here is to predict the total render time T_c of a frame, based on the size of the
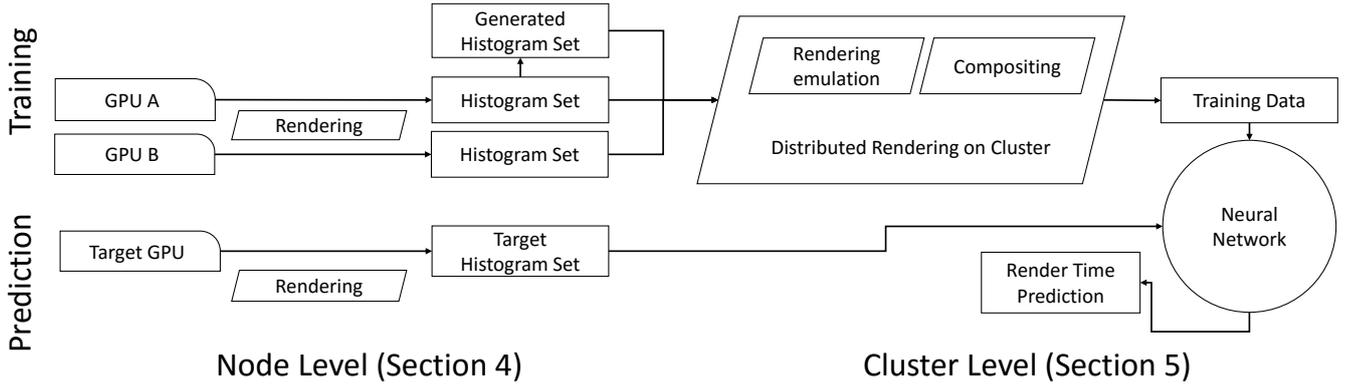
Figure 1: *An overview of our prediction approach. First, we measure local rendering performance of our training hardware, obtaining two sets of histograms (top-left). We generate an additional set of histograms by scaling our measured data. Then, we use the histograms to perform rendering emulation on the cluster (top-right), collecting the resulting cluster frame time for training our neural network. We can then predict cluster performance by measuring a histogram set on the target GPU (bottom-left), and using it as input to the model (bottom-right).*

resulting image (I), the data set and its size (D), view parameters (V), the node hardware (H) and the cluster size (C). More formally, we are looking for a model that achieves the following mapping:

$$(I, D, V, H, C) \rightarrow T_c. \tag{1}$$

In particular, our focus lies on predicting the performance impact of changes to the node hardware H.

Parallel volume rendering. To develop our prediction technique, we implemented a parallel volume renderer. It can be classified as a sort-last renderer [MCEF94], using object-space partitioning to parallelize the computation. Using this technique, the rendering of each frame basically consists of two major phases: local rendering and inter-node compositing. During the local rendering phase, each node independently renders its own partition, performing raycasting on GPUs. During the second phase, inter-node compositing combines the individual images into the final rendering using the 2-3 swap compositing scheme [YWM08]. In 2-3 swap, the compositing is performed in steps, during which nodes exchange and compose data in small groups of up to four nodes. Initially, each node has a full image with only its own volume partition rendered on it. As data is exchanged each step, the image that the node holds shrinks, while its contents become more complete, i. e. contribution of other volume partitions is taken into account. And in the end, each processor is left with a small complete chunk of the final image.

Modeling via neural networks. A distributed renderer running on a GPU cluster is a complex system exhibiting parallelism at multiple levels and featuring intricacies like network congestion. In this work, we chose artificial neural networks for capturing this complexity. An artificial neural network is a network of interconnected neurons, in which a signal flows from the input neurons through the whole network to the output neurons. In the course of this, the neurons transform the sum of the weighted incoming signals to an output signal using a nonlinear activation function. While the network is being trained, input data is "fed" to the network, gradually adjusting weights of the connections to improve the network's prediction accuracy (Sec. 5 for a more detailed discussion).

Our two-level prediction approach. To obtain acceptable results from a neural network, it needs to be trained with a lot of training data on a wide range of hardware (we evaluate the benefits of additional training data in Sec. 6.3). The more general the model should be, the more hardware, data sets and different output resolutions we need for collecting the training data to prevent the network from becoming biased towards one of these factors. However, the number of GPU clusters available for training is limited, and exchanging all GPUs of a cluster for extensive testing is unfeasible in practice.

To overcome this, we reformulate our model by splitting it into two levels similar to the two phases of an object-space distributed renderer. First, we render an image of a subset of the data on each node (local rendering). Second, a compositing phase follows, which exchanges data over the network and assembles the final image on the CPU. In our setup, the GPU only affects the local rendering phase. This allows us to abstract this part of the hardware and application parameters as a *local render time* T_n, which specifies how long it takes a node of the cluster to finish its local rendering. With this, the model from Eq. 1 can be split into two models:

$$(I, D, V, H) \rightarrow T_n \tag{2}$$
$$(I, C, T_n) \rightarrow T_c \tag{3}$$

The first model (Eq. 2) represents the local rendering phase, where image size (I), data set (D), view parameters (V) and node hardware (H) define the local render time (T_n). The second model (Eq. 3) covers the compositing phase, mapping image size (I), cluster size (C) and local render time (T_n) to the final cluster frame time (T_c). An advantage of this reformulation is that Eq. 3 does not rely on information about the hardware used for rendering, in contrast to the original model (Eq. 1). We therefore can emulate different rendering hardware on a single cluster by simply stalling the nodes according to T_n. This allows us to gather more performance measurements from the cluster by emulating different local GPU rendering times without actually installing different graphics cards in the nodes (see Sec. 4). The data gathered this way can then be used for training the neural network to predict Eq. 3 (see Sec. 5). The model we eventually obtain is only tied to the hardware used for compositing and the network hardware and topology. This means that our model can

make meaningful predictions on the basis of local render time measured on a single node equipped with target hardware. A graphical overview of our approach is presented in Fig. 1.

4. Emulation of local rendering performance

A key technique for acquiring sufficient training data for the cluster performance model is the emulation of local render times T_n (Eq. 2). This section covers our approach to suitably representing local rendering performance via so-called *performance histograms*.

Motivation and objective. The main issue when taking a machine learning-based approach to performance prediction for clusters is that typically only one (or very few) specific hardware configurations are available for obtaining training data. Therefore, the training data is conceptually restricted to this one configuration, and allows prediction only in terms of scaling with the number of nodes. To circumvent this, we propose to remove local rendering from the model to only capture how communication (hardware and network topology) affects the final time, training the model to predict the cluster performance already given local performance. We represent this local performance using so-called performance histograms, which can either be measured from arbitrary hardware available on individual machines, or even be generated artificially. This allows us to emulate different local rendering performance on a single cluster and collect more training data. Additional training data helps the cluster model (discussed in Sec. 5) to better learn the dependency between local rendering performance and overall render time, and prevents the model from implicitly adapting to particular node performance.

Perf. histograms representing local render time. An *emulation run* works similar to real rendering, but during the local rendering phase, we simply stall the nodes without performing any actual computation. Once all the nodes have finished waiting, the internode compositing phase proceeds as normal. The nodes exchange and compose random data, but the amount of compositing and communication remains unchanged, allowing us to measure how a given local render time affects the overall cluster performance (Eq. 3). By varying the local render time, we can simulate rendering with different combinations of hardware and application parameters, producing more training data from a single cluster. Note, that during compositing we always consider the full image, and do not make any optimizations based on footprints.

The local render time T_n obviously does not represent the time it takes to render the whole volume on a single node, but rather the time it takes for a single node of a cluster to render its partition. In our implementation, we use a static, uniform partitioning of the volume into bricks, i.e. each partition has an approximately equal number of voxels. However, render times still can differ significantly between bricks due to perspective projection and early ray termination. This results in dynamic load imbalance with different nodes becoming the bottleneck under different camera orientations.

Thus, instead of using a static, predefined local render time for each node, we use a representation of node performance that can capture dynamic load imbalance. To do so, we use a distribution of local render time, which describes a probability of a node taking a certain amount of time to perform local rendering during a frame. This distribution can be expressed as a histogram, which is randomly
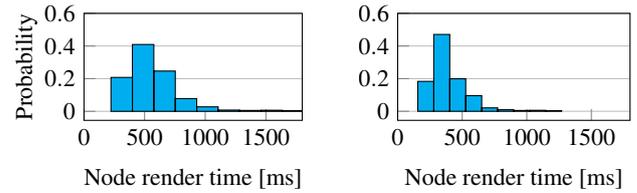

Figure 2: *Performance histograms created from measurements (left), and artificially generated (right). They represent local rendering performance of a node of a 24-node cluster for image size 6144^2 and volume size 1024^2.*

sampled to decide how long a node needs to stall when emulating local rendering. This way, we can replicate the varying character of local render time, while still maintaining the same average performance, resulting in a reasonably accurate emulation of volume rendering performance (Fig. 5b).

Obtaining performance histograms. As outlined in Sec. 3, our goal is to use rendering emulation to collect more training data from a single cluster. To perform more emulation runs, we require more performance histograms that describe the node performance to be emulated. The histograms can be obtained either by measuring them on various node hardware, or by generating them artificially, representing some hypothetical hardware. We explore viability of both approaches, measuring some histograms on existing hardware and generating additional histograms to get more training data.

To measure the histograms on existing node hardware, we execute a run of the volume renderer for each combination of image size, volume size and cluster size parameters (this influences the size of the volume partitions). We record the local render time for each node and frame and sort the measurements into a histogram. By measuring a histogram for each combination of the parameters, we obtain a set of histograms that captures performance of the tested node hardware. Additionally, the histogram approach allows us to uniformly represent both single and dual GPU configurations. Two GPUs act together to render the node's partition, and from the standpoint of local render time are viewed as a single faster rendering device. Thus, we can measure two sets of histograms: one in single GPU mode, and another in dual GPU mode.

Then, we generate modified histograms that are similar to those measured during actual volume rendering to produce a larger variation of training data (Fig. 2). This noticeably improves the results of our cluster model (Sec. 6.3). To generate artificial histograms, we take the values from a histogram previously measured on hardware under the same rendering parameters and perturb it slightly using a uniform distribution. Without perturbation, all generated histograms would be the same, making the network biased towards their particular shape. To define the domain of the histogram, i.e. the minimal and the maximal local render time, we take the values from the measured histogram and scale it by a constant, effectively imitating slower/faster hardware with similar scaling behavior. This way, we generate two additional sets of histograms. Note, that while the primary purpose of modifying histograms is to generate a larger variation of training data, the concept could also be applied to basically simulate arbitrary combinations of hardware and volume
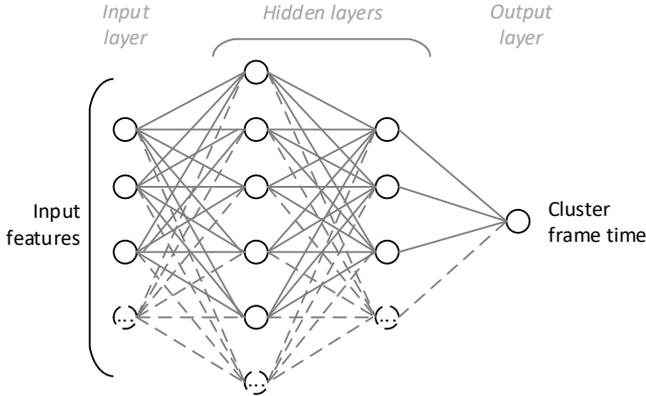
Figure 3: *Our neural network for cluster performance prediction. It includes two hidden layers of 16 and eight neurons with ReLU activation function and L2-regularization.*

(a) Chameleon   **(b)** Learning curve

Figure 4: *(a) Test data rendering. (b) Learning curve showing improving prediction accuracy with larger training sets.*

rendering characteristics. However, investigating this more closely is beyond the scope of this paper and remains for future work.

5. Cluster performance model

In this section, we present our cluster performance model. For this, we train a neural network using data acquired through our rendering emulation technique (Sec. 4).

Model input and output. For our model we train a neural network that has the following input data:

Image resolution (I). Although image size is an application parameter that affects local rendering and is implicitly captured in a performance histogram, it also defines the amount of data exchanged over the network, and thus, it is useful for predicting the cluster frame time.

Cluster size (C). The number of nodes affects both the amount of data exchanged and the communication pattern.

Performance histogram (T_n). The performance histogram is fed into the network using two sets of features. The average, minimum and maximum local render time features define the domain of the histogram, which is a rough estimate of node performance. Ten bin features represent the distribution of local render time, which encodes the load imbalance of the rendering application (Sec. 4). By choosing to use ten histogram bins, we aim to maximize the resolution of the histogram to provide more data for the network. However, a further increase of the bin number causes the histograms to have occasional gaps, introducing undesirable noise into the input data.

Cluster frame time (T_c). As output, the model provides a prediction for the cluster frame render time. Specifically, we predict the average frame time recorded over the camera path. The choice of the target variable is made in line with our emulation technique (Sec. 4): our histogram-based emulation is designed to match the average performance (Sec. 6), so by training the model to predict the average performance of the emulation, we transitively predict the average performance of actual volume rendering.
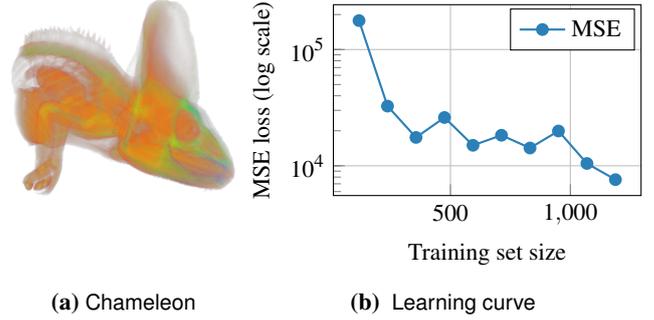
Neural network architecture. Our architecture is a series of fully interconnected layers of neurons (Fig. 3). Each neuron of the network can be viewed as a unit that outputs a nonlinear weighted sum of the incoming signals. It sums the incoming signals according to weights of the connections, and then applies the activation function to the result [Bis95]. The activation function is a nonlinear function that enables ANNs to model complex nonlinear dependencies. For our model, we chose the rectified linear unit (ReLU) as activation function, that allows for faster network training than the also widely used *tanh* function [GBB11]:

$$g_{ReLU}(x) = ln(1 + e^x). \tag{4}$$

In our neural network, the first layer is the input, with one neuron for every input feature (Fig. 3). The last layer is the output and consists of a single neuron, since the render time is our sole output variable. The rest of the layers are the so-called hidden layers. Generally, the number of neurons can be varied to control the complexity of the network. Too little complexity typically means that the network is too simple to represent important relations, while too much complexity induces the risk of overfitting (where the network becomes so flexible, that it can closely fit every training data point instead of capturing a general trend). Because a good choice is very hard to make a priori, we determined the adequate structure for our network experimentally as follows. We started with a simple single-layer network, and increased the number of neurons and layers while the accuracy of prediction was improving. Using this procedure resulted in a network with two hidden layers of 16 and eight neurons, since further increase in the network's complexity yielded no improvement in accuracy of prediction.

Neural network training. For training the network, we use both measured and generated histograms (Sec. 6). Each point of the training data maps a combination of image size, cluster size and a corresponding histogram to the resulting cluster frame time (Eq. 3).

First, the network is evaluated on the input data, making a left-to-right pass through the network, called forward propagation. Next, the back propagation algorithm [RHW85] is used to compute all the gradients of the network, i. e. how much the loss changes when altering each weight of the network. The gradients can be used to update the weights, changing them in the direction of decreasing loss, effectively performing gradient descent. The loss function quantifies the accuracy of the network by means of how much the

network's output (y_o) deviates from the target output (y_t). In this work, we employ the *mean squared error (MSE)* loss (Eq. 5).

$$MSE = \frac{1}{n} \sum_{i=1}^{n} \left(y_t^{(i)} - y_o \right)^2 \qquad (5)$$

To address the aforementioned overfitting issues, a regularization term is introduced to the loss function, penalizing large network weights. A regularization parameter λ is used to control the amount of regularization. For our model, we perform L2-regularization over all the weights w_{ij} in the network (Eq. 6), automatically choosing the value of λ during training. Specifically, we train a network for the values of $\lambda \in \{10^{-3}, 10^{-2}, \ldots, 10^5\}$, and evaluate it on the validation data, choosing λ resulting in a smallest MSE loss:

$$E_r = MSE + \frac{\lambda}{2} \sum_{i,j} w_{ij}^2 \qquad (6)$$

6. Results

In this section, we evaluate the prediction accuracy of our model. We begin by describing the training data for the neural network. Then we assess the quality of our prediction in a cluster upgrade scenario, and also on a different cluster with a similar network configuration.

6.1. Implementation

Our renderer can utilize both one and two GPU(s) per cluster node. In a case of multiple GPUs, the raycasting results are locally composed among the GPUs. The raycasting uses front-to-back compositing with early ray termination and is implemented in CUDA. Once all nodes have finished local rendering, the intermediate results are exchanged among the nodes and composited into the final image using the 2-3 swap compositing scheme [YWM08]. We used asynchronous operations provided by the MSMPI implementation for our inter-node communication. The volume is partitioned using a *k*-d tree, which implicitly provides ordering for compositing. We statically assign a partition to each node during initialization, without performing runtime load balancing. If a node has multiple GPUs, the partition is split further, assigning a sub-partition to each GPU.

With four nodes on a 1024^3 volume with 6144^2 image size the renderer achieved an average frame time of 3545 ms, with 2084 ms spend in local rendering phase and 1461 ms in compositing. As we increased the cluster size, the frame time started to decrease, with compositing having a larger impact on performance. For example, with 32 nodes the average frame time was reduced to 2081 ms, where local rendering and compositing took 394 ms and 1687 ms respectively. Overall, both local rendering and compositing had a significant impact on the cluster performance, which showed the necessity of a two-level approach to cluster performance modeling.

For our neural network implementation, we used the Keras framework [Cho15], which in term relies on Theano [The16].

6.2. Collecting training data

We acquired the training data for our model on a 33-node GPU cluster. Each node was equipped with two Intel Xeon E5620 CPUs, 24 GB RAM, two NVIDIA GeForce GTX 480 GPUs and DDR

InfiniBand. The InfiniBand network had full bisectional bandwidth. For evaluation, we used a render run consisting of 72 frames while orbiting the camera twice around the volume on the XZ-plane. We recorded the local render time for each volume partition and the overall cluster frame time. We considered every combination of the following parameters (a total of 594 configurations):

- Image size $\in \{1024^2, 2048^2, 3072^2, 4096^2, 5120^2, 6144^2\}$
- Volume size $\in \{256^3, 512^3, 1024^3\}$ (scaled version of the Chameleon, Fig. 4a, with the original size being 1024^3 voxels)
- Cluster size $\in \{1, 2, \ldots 33\}$

We measured histograms for both single and dual GPU mode for the 594 configurations mentioned above. Furthermore, from the sets of measured histograms, we artificially generated two additional sets. We did this by perturbing the histogram bins with a uniform distribution and scaling the domain of the histograms by a factor of 0.7. The rationale behind this factor is that it significantly changes the local rendering performance, while the result still remains within the same order of magnitude as measurements. This produces variation in the training data that helps prediction (Sec. 6.3).

Finally, we used measured and generated histograms to perform rendering emulation on the cluster, obtaining the data for training the network (see Fig. 1 for an overview):

- S_{m1}: Measured-histogram data set, single GPU mode
- S_{m2}: Measured-histogram data set, dual GPU mode
- S_{g1}: Generated-histogram data set, derived from S_{m1}
- S_{g2}: Generated-histogram data set, derived from S_{m2}

6.3. Emulated and actual render timings

An important aspect of our approach is that we used rendering emulation to obtain training data for our model (as discussed in Sec. 4), making it possible to train a full cluster model while measuring only on a single cluster. The benefit of additional training data can be seen in Fig. 4b where we depict the learning curve, plotting MSE loss on the test data against the training set size. The initial steep drop corresponds to the drastic improvement made after having almost no training data. (Note, that to show more details a logarithmic scale is used on the y-axis.) As more training data is added, one can see an improvement in accuracy on the test data set (the model becomes better at generalizing to previously unseen data). This shows that the additional training data acquired through usage of performance histograms and rendering emulation improves the prediction accuracy of our model and makes the approach practical.

We investigated the accuracy of our distribution-based rendering emulation technique by comparing its performance to that of an actual rendering application. For this, we performed a set of normal rendering runs, recording not only the cluster frame time, but also the performance histograms. (Fig. 5b). We can see that the performance emulated with the histograms closely matches the performance of an actual render run.

To demonstrate the importance of using a distribution of values for our emulation, we further compared it against a simpler vector-based technique, which assigns a predefined local render time to each node. For this, local render times were averaged for each node over all frames to acquire a local render time vector.
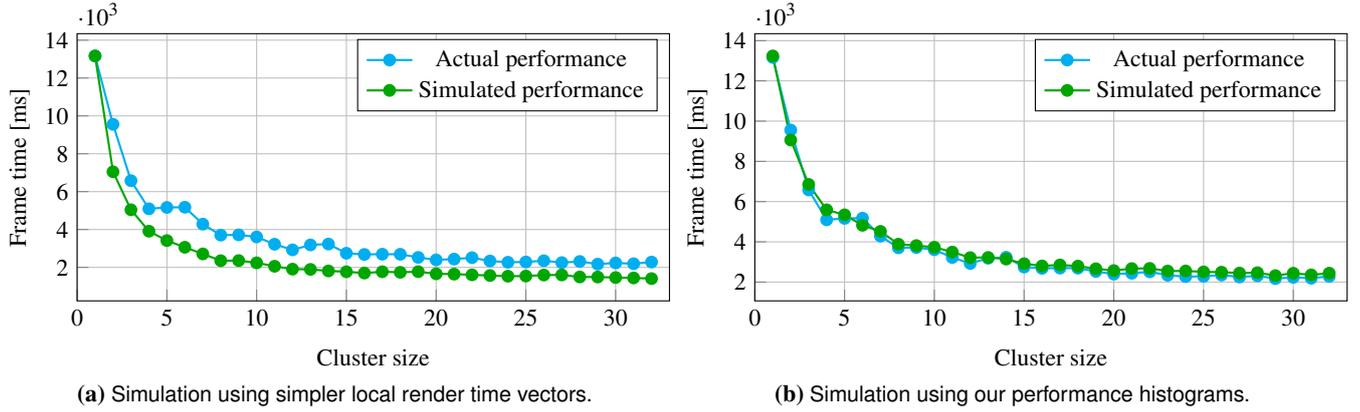
(a) Simulation using simpler local render time vectors.

(b) Simulation using our performance histograms.

Figure 5: *Comparison of actual and simulated performance for 6144^2 image size and 1024^3 volume size. 5a: Simulation using local render time vectors, with each node stalling for the average amount of time taken by this node during an actual rendering run. 5b: Simulation using performance histograms, with each node sampling the distribution during runtime to determine the amount of time it should be stalling.*
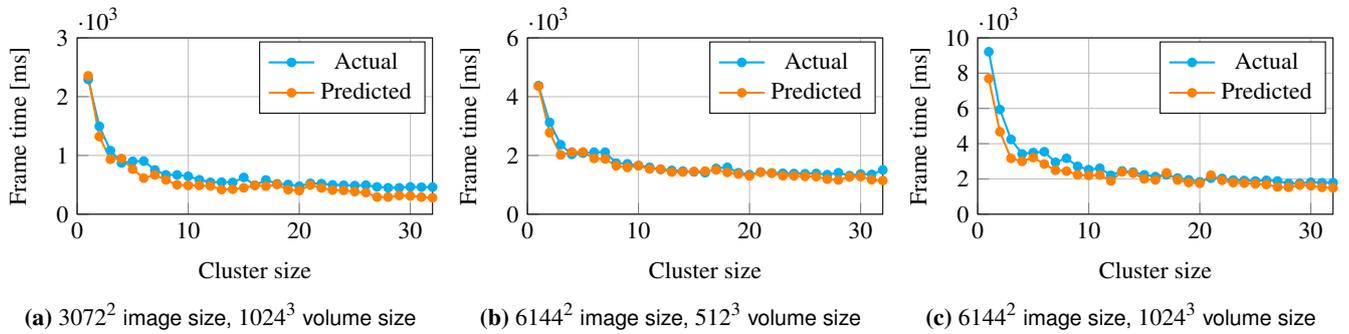

(a) 3072^2 image size, 1024^3 volume size

(b) 6144^2 image size, 512^3 volume size

(c) 6144^2 image size, 1024^3 volume size

Figure 6: *Evaluation of our model in the cluster upgrade scenario for different image and volume resolutions. The model is trained using data obtained in single GPU mode, and used to predict performance of the cluster with two GPUs per node. All three subplots represent different 'slices' of the same data, for different fixed values of image and volume size.*

Fig. 5a illustrates that the vector-based approach allows imitating a general performance trend, but has a large deviation from the actual performance. The vectors capture the static load imbalance, i. e. nodes having different computational load overall (e. g., due to some parts of the volume being more opaque than others), but it does not capture the dynamic load imbalance, which refers to the effect of nodes having different computational load every frame, with different nodes becoming the fastest/slowest under different camera orientations. However, our histogram-based approach covers both types of load imbalance. Therefore, it has a significantly smaller performance deviation (Fig. 5b), making it suitable for obtaining better training data.

6.4. Predicting performance of an upgraded cluster

We evaluated our approach in a cluster GPU upgrade scenario. Specifically, we investigated benefits of upgrading single GPU nodes to dual GPU nodes (same model). We trained our model using two data sets that were obtained from measurements in single GPU mode: a measured-histogram data set S_{m1} and a generated-histogram data set S_{g1}. No data collected in dual GPU mode was used for training. Next, to make a prediction, we collected a set of histograms on the target hardware, i. e. on a node equipped with two GPUs. This way,

we compared the prediction of the model to the actual rendering performance, and not just the emulation performance. To test our prediction, we executed the volume renderer on the whole cluster running in dual GPU mode, without any rendering emulation, which enables us to compare the prediction to the actual rendering performance and not just the emulation performance. The results are presented in Fig. 6, with model achieving an MSE loss of $3.826 \cdot 10^4$ and an R^2 score of 0.95. It can be seen that the model works well both for a smaller and larger number of nodes. For instance, in Fig. 7b we see how the prediction matches both the steep descent in the beginning of the graph, and the "tail" of the graph, where the communication overhead prevents further performance gain. Furthermore, the model also reproduces the smaller details of the scaling curve, predicting which cluster sizes are more favorable: in Fig.6c both the prediction and the actual performance have a local minimum at even cluster sizes.

6.5. Predicting performance across different clusters

While the main focus of our approach is on predicting the outcome of upgrading node rendering hardware, it can also be used to predict the performance of a different cluster having a similar network configuration. For this application case, the test was performed
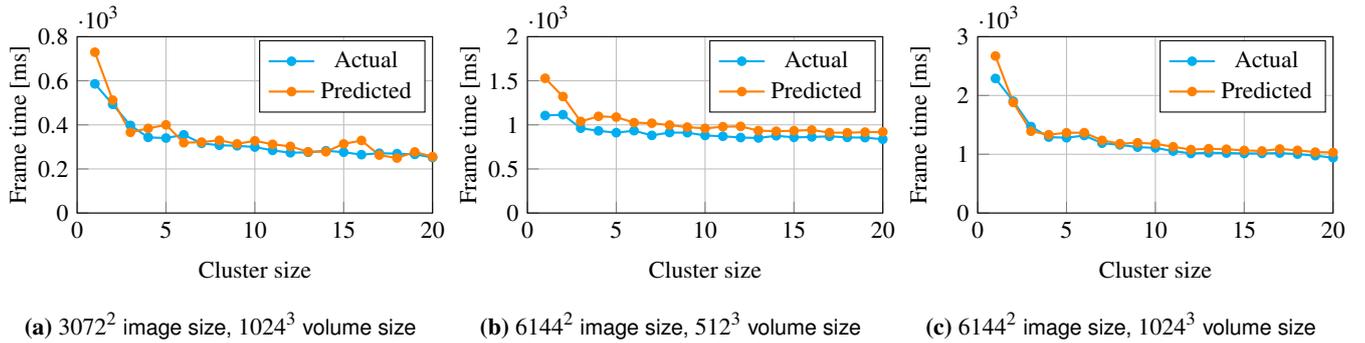
(a) 3072^2 image size, 1024^3 volume size **(b)** 6144^2 image size, 512^3 volume size **(c)** 6144^2 image size, 1024^3 volume size

Figure 7: *Evaluation of our model on a different cluster. The model is trained using data collected on one cluster, and used to predict performance of a different cluster with similar network configuration.*

with data from a different cluster. Therefore, we trained our model with measured data sets S_{m1} and S_{m2}, using data acquired both in single and dual GPU mode. The generated data set S_{g2} was used to automatically choose a value of the regularization parameter λ during training (Sec. 5). For testing, we performed measurements on a 20-node cluster with two Intel Xeon E5-2640 v3 CPUs, 128 GB RAM, an NVIDIA Quadro M6000 and FDR InfiniBand (note that this is a faster network interconnect than in the cluster used for measuring the training data). As mentioned before, no rendering emulation was used for the test data, so we compared the prediction of the model to the actual rendering performance, and not just the emulation performance.

The model achieves an MSE loss of $7.603 \cdot 10^3$ and an R^2 score of 0.93. It exhibits some jittering when predicting performance on smaller image and volume sizes, producing an initial spike in Fig. 7a and 7b. In this case the exact shape of the curve changes between different executions of the neural network training (due to random initialization of the network's weights), but the overall trend remains the same: on a small number of nodes, the model predicts lower performance compared to measured results. An explanation to this deviation is that the GPUs of the training cluster are significantly older and slower than the ones of the test cluster, and, unlike them, benefit from parallelization even for smaller image and volume sizes. Thus, the network has observed a steady decline in frame time, as we add more nodes to the cluster. However, the test cluster has a different scalability trend, which results in a large deviation from the model's prediction for a small number of nodes.

In all three cases we can see that the model's prediction has some bias as the model consistently predicted slower performance than what was actually achieved. We attribute this deviation to the different network interconnects, as the lower network bandwidth of the training cluster is implicitly captured in the model. Hence, when the model is used to predict performance of a cluster with a faster network, it consistently underestimates the performance. This implies, as expected, that our approach is not suitable for predicting performance of an arbitrarily different cluster. While changes in rendering hardware are covered by our histogram approach, the communication/compositing is learned by the model from the data of the training cluster, and changes in that respect are not accounted for in any way. However, our results show that we can still yield reasonable results when applying this approach to clusters with similar network configuration (cf. Sec. 7).

Layers	Neurons	Lambda	Test score	Val. score
1	64	1	10207.09	5949.29
1	64	10	12801.47	6200.41
1	128	0.01	8178.46	6233.40
2	16+8	1	10241.14	6289.66
		...		
2	128+64	0.01	11632.76	14039.96
1	8	1	30517.56	15841.59
3	64+32+16	0.1	17013.04	17065.43
		...		
3	8+4+2	0.01	106556.08	302147.11
3	8+4+2	1	111366.93	303968.76
3	8+4+2	0.1	138458.65	341182.30

Table 1: *Comparison of NN architectures. Performance of 75 networks ordered by the validation score, best (top) to worst (bottom).*

6.6. Evaluation of the neural network architecture

To further validate our empirically chosen neural network architecture (see Sec. 5) we trained 75 different networks, varying the number of layers (1, 2, 3), the number of neurons (8, 16 … 128) and the regularization parameter ($10^{-2}, 10^{-1} \ldots 10^1$). In Table 1 we present the results for best-, medium- and worst-performing networks, chosen based on their validation dataset score (average of 10 runs). We can see, that the best-performing networks show similar results, however our '16+8' architecture has significantly fewer neurons and hence, lower training time. Among the networks with medium performance, we can find both complex architectures (e.g. 128+64 neurons) with low regularization, and simple architectures (e.g. 8 neurons) with stronger regularization. The former suffer from overfitting the data, while the latter are too simple to capture some of the details of the training data. Finally, the worst-performing networks are deeper networks with fewer neurons (e.g. 8+4+2 neurons) that attempt to reduce the data to a very compact representation (just 2 neurons), which is insufficient to meaningfully represent the training data, leading to poor accuracy.

7. Discussion, limitations and future work

In our approach, we used performance histograms as a representation of node performance. However, these depend on the exact rendering scenario used for their measurement (camera path). Choosing a different camera path for performing the rendering may have an

effect on the local rendering performance (Sec. 4). Although our model conveniently abstracts away such details of local rendering, using a model trained under a certain scenario to predict performance of a significantly different scenario may potentially worsen the prediction accuracy. In this case the neural network might not have observed this significantly different performance histogram, and might give a less accurate prediction. We performed some testing of these conditions, predicting rendering performance for a camera orbiting in a different plane and with a different transfer function, observing no significant drop in prediction accuracy. However, a more thorough investigation is left for future work.

Furthermore, our histogram construction technique exerts a limitation on the complexity of the camera trajectory used for performance measurement. In particular, we construct the histograms by measuring the performance of each node during each frame of a rendering run and sorting all the measurements into bins. Thereby, we aim to capture both, the dynamic load imbalance (i. e., nodes having different local render time during different frames) and the variation in the overall computational load caused by the volume being faster to render under certain camera orientations. This results in mixing together two different distributions: one characterizes load imbalance among the nodes, the other the overall computational load, which varies between the frames. For example, if we measured a histogram in a scenario where camera distance to the volume is changed significantly, the resulting histogram could be interpreted by our rendering emulation as "some nodes being slow, some being fast", instead of "all nodes being slow during certain frames, and all nodes being fast during other frames". Eventually, this would lead to a different emulated rendering performance, making our method less accurate. This limitation could be addressed by using a distribution of distributions, representing local render time distribution during each frame separately. For emulation, one would first choose a distribution for the current frame, and then use it to determine the local render time for each node. However, this extension has two major challenges that require further investigation: sorting distributions into a histogram, and finding a vectorial representation suitable for training neural networks.

Furthermore, our statistical approach would benefit from a larger amount of systematically gathered empirical data of volume rendering performance. This way, one could learn about how render times are distributed in general and use this information to build more representative emulations.

For the sake of simplicity, we did not implement any advanced optimization techniques (empty-space skipping, interleaved rendering and composition, etc.) in our volume renderer. However our approach is in principle capable of handling these methods, and we would like to investigate its prediction accuracy in future work.

Our performance prediction approach is best suited for supporting equipment procurement, allowing cluster performance to be estimated by only performing measurements on one of its nodes, without purchasing the whole set of hardware. However, the model implicitly captures network hardware and topology of the training cluster and therefore has some limitation in general applicability and re-usability. Optimally, it needs to be trained on a cluster with similar network conditions. This presents no problem in the case of node hardware upgrade, but can become cumbersome when building a completely new GPU cluster. Possible future extensions are therefore the emulation of the composition phase performance and a suitable representation of communication patterns. This could be used to abstract away cluster-specific details, similar to how we use performance histograms to abstract away node hardware.

One could argue that an analytical model for estimating local rendering performance, e. g. in the form of a cost-per-sample calculation, could be a better approach. However, our usage of histogram poses the advantage of being possibly a much more universal approach, in terms of variations in the volume rendering technique or even applicability to other applications. Furthermore, (GPU) hardware algorithms such as caching, swizzling, 3D memory etc. can be the cause of significant performance deviations [BFE16].

8. Conclusion

In this paper, we presented an approach to predicting the performance of a distributed GPU volume renderer for supporting equipment acquisition scenarios. We proposed a two-level approach to prediction, distinguishing between node and cluster level. This allowed us to capture complex performance characteristics via a neural network, without the need to collect training data on multiple clusters with different hardware. By only using measurements from a single cluster and individual GPUs, our approach is capable of predicting the performance gain of either upgrading GPUs of an existing cluster, equipping one node with multiple GPUs, or purchasing a new cluster with a comparable network configuration. For individual GPUs (i. e., the node level), we generated performance histograms, which essentially capture local rendering performance. These performance histograms were then used to emulate the rendering equipment of a whole cluster for cluster-level measurement runs. Using these measurements we trained a neural network to predict the cluster performance based on local rendering performance.

With our approach, we were able to obtain accurate predictions for our main application scenario, the upgrade of GPUs in an existing cluster. Furthermore, when using it in a scenario of a faster, but similar network configuration, we still achieved an adequate accuracy with a consistent error reflecting the difference in network performance. For such application cases, we intend to extend our model of cluster performance to become network-agnostic. We also want to gather a wider variety of measurements for various hardware to improve the model on the local rendering level as well.

Acknowledgments

References

[BDP*10] BAGHSORKHI S. S., DELAHAYE M., PATEL S. J., GROPP W. D., HWU W.-M. W.: An adaptive performance modeling tool for gpu architectures. In *ACM Sigplan Notices* (2010), vol. 45, pp. 105–114. 2

[BFE16] BRUDER V., FREY S., ERTL T.: Real-time performance prediction and tuning for interactive volume raycasting. In *Proc. SIGGRAPH Asia 2016 Symposium on Visualization* (2016),

pp. 7:1–7:8. URL: http://doi.acm.org/10.1145/3002151.3002156, doi:10.1145/3002151.3002156. 9

[BFJ00] Brown P. N., Falgout R. D., Jones J. E.: Semicoarsening multigrid on distributed memory machines. *SIAM Journal on Scientific Computing 21*, 5 (2000), 1823–1834. 2

[BHP15] Beyer J., Hadwiger M., Pfister H.: State-of-the-art in gpu-based large-scale volume visualization. *Computer Graphics Forum* (2015). URL: http://onlinelibrary.wiley.com/doi/10.1111/cgf.12605/abstract. 2

[Bis95] Bishop C. M.: *Neural networks for pattern recognition*. Oxford University Press, 1995. 5

[BRL*08] Barnes B. J., Rountree B., Lowenthal D. K., Reeves J., de Supinski B., Schulz M.: A regression-based approach to scalability prediction. In *Proc. ACM/IEEE Supercomputing* (2008), pp. 368–377. URL: http://doi.acm.org/10.1145/1375527.1375580, doi:10.1145/1375527.1375580. 2

[Cho15] Chollet F.: Keras. https://github.com/fchollet/keras, 2015. 6

[EB16] Escobar R., Boppana R. V.: Performance prediction of parallel applications based on small-scale executions. In *Proc. High Performance Computing* (2016), pp. 362–371. doi:10.1109/HiPC.2016.049. 2

[FCS*10] Fogal T., Childs H., Shankar S., Krüger J., Bergeron R. D., Hatcher P.: Large data visualization on distributed memory multi-gpu clusters. In *Proceedings of the Conference on High Performance Graphics* (2010), Eurographics Association, pp. 57–66. 2

[GBB11] Glorot X., Bordes A., Bengio Y.: Deep sparse rectifier neural networks. In *Aistats* (2011), vol. 15, p. 275. 5

[HBC12] Howison M., Bethel E. W., Childs H.: Hybrid parallelism for volume rendering on large-, multi-, and many-core systems. *IEEE Trans. Vis. Comput. Graphics 18*, 1 (2012), 17–29. doi:10.1109/TVCG.2011.24. 2

[HGKS11] Hoefler T., Gropp W., Kramer W., Snir M.: Performance modeling for systematic performance tuning. In *State of the Practice Reports* (2011), SC '11, pp. 6:1–6:12. URL: http://doi.acm.org/10.1145/2063348.2063356, doi:10.1145/2063348.2063356. 2

[IdSSM05] Ipek E., de Supinski B. R., Schulz M., McKee S. A.: *An Approach to Performance Prediction for Parallel Applications*. Springer, Berlin, 2005, pp. 196–205. URL: http://dx.doi.org/10.1007/11549468_24, doi:10.1007/11549468_24. 2

[LBdS*07] Lee B. C., Brooks D. M., de Supinski B. R., Schulz M., Singh K., McKee S. A.: Methods of inference and learning for performance modeling of parallel applications. In *Proc. SIGPLAN Principles and Practice of Parallel Programming* (2007), pp. 249–258. URL: http://doi.acm.org/10.1145/1229428.1229479, doi:10.1145/1229428.1229479. 2

[LHK*16] Larsen M., Harrison C., Kress J., Pugmire D., Meredith J. S., Childs H.: Performance modeling of in situ rendering. In *Proc. High Performance Computing, Networking, Storage and Analysis* (2016), SC '16, pp. 24:1–24:12. URL: http://dl.acm.org/citation.cfm?id=3014904.3014936. 2

[MCEF94] Molnar S., Cox M., Ellsworth D., Fuchs H.: A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl. Mag. 14*, 4 (1994), 23–32. doi:10.1109/38.291528. 2, 3

[MSE06] Müller C., Strengert M., Ertl T.: Optimized volume raycasting for graphics-hardware-based cluster systems. In *Proc. EGPGV* (2006), pp. 59–66. 2

[Pet04] Petitet A.: HPL – a portable implementation of the high-performance linpack benchmark for distributed-memory computers. *http://www.netlib.org/benchmark/hpl/* (2004). 2

[PYR*08] Peterka T., Yu H., Ross R. B., Ma K.-L., et al.: Parallel volume rendering on the ibm blue gene/p. In *Proc. EGPGV* (2008), pp. 73–80. 2

[RHI*14] Rizzi S., Hereld M., Insley J. A., Papka M. E., Uram T. D., Vishwanath V.: Performance modeling of vl3 volume rendering on gpu-based clusters. In *Proc. EGPGV* (2014), pp. 65–72. URL: http://dx.doi.org/10.2312/pgv.20141086, doi:10.2312/pgv.20141086. 2

[RHW85] Rumelhart D. E., Hinton G. E., Williams R. J.: *Learning internal representations by error propagation*. Tech. rep., DTIC Document, 1985. 5

[SIM*07] Singh K., Ipek E., McKee S. A., de Supinski B. R., Schulz M., Caruana R.: Predicting parallel application performance via machine learning approaches. *Concurrency and Computation 19*, 17 (2007), 2219–2235. URL: http://dx.doi.org/10.1002/cpe.1171, doi:10.1002/cpe.1171. 2

[SSX08] Sodhi S., Subhlok J., Xu Q.: Performance prediction with skeletons. *Cluster Computing 11*, 2 (2008), 151–165. URL: http://dx.doi.org/10.1007/s10586-007-0039-2, doi:10.1007/s10586-007-0039-2. 2

[The16] Theano Development Team: Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints abs/1605.02688* (May 2016). URL: http://arxiv.org/abs/1605.02688. 6

[WGL*15] Wu G., Greathouse J. L., Lyashevsky A., Jayasena N., Chiou D.: Gpgpu performance and power estimation using machine learning. In *Proc. HPCA* (2015), pp. 564–576. doi:10.1109/HPCA.2015.7056063. 2

[YWM08] Yu H., Wang C., Ma K.-L.: Massively parallel volume rendering using 2–3 swap image compositing. In *Proc. High Performance Computing, Networking, Storage and Analysis* (2008), pp. 1–11. 3, 6

[ZO11] Zhang Y., Owens J. D.: A quantitative performance analysis model for gpu architectures. In *Proc. HPCA* (2011), pp. 382–393. 2