A Compute Unified System Architecture for Graphics Clusters Incorporating Data-Locality

Christoph Müller, Steffen Frey, Magnus Strengert, Carsten Dachsbacher, Thomas Ertl

Abstract—We present a development environment for distributed GPU computing targeted for multi-GPU systems as well as graphics clusters. Our system is based on CUDA and logically extends its parallel programming model for graphics processors to higher levels of parallelism, namely the PCI bus and network interconnects. While the extended API mimics the full function set of current graphics hardware – including the concept of global memory – on all distribution layers, the underlying communication mechanisms are handled transparently for the application developer. To allow for high scalability, in particular for network interconnected environments, we introduce an automatic GPU-accelerated scheduling mechanism that is aware of data locality. This way, the overall amount of transmitted data can be heavily reduced, which leads to better GPU utilization and faster execution. We evaluate the performance and scalability of our system for bus and especially network level parallelism on typical multi-GPU systems and graphics clusters.

Index Terms—GPU computing, Graphics Clusters, Parallel Programming

1 Introduction

Each new generation of GPUs provides flexible programmability and computational power which exceeds previous generations. At the beginning of this remarkable process, graphics hardware expanded its real-time shading capabilities from simple Gouraud shading and texture mapping to rendering with multiple texture stages and combiners. While graphics hardware became more powerful, it also became intricate and difficult to use. As a consequence, shading languages – which have already been successfully used in non-real time applications – have been brought to mainstream graphics hardware. One of the first projects with this objective was the Stanford Real-Time Programmable Shading Project [19]. Nowadays, graphics hardware is capable of executing very costly and complex algorithms formerly only practical with CPUs. Processing non-graphics tasks on GPUs further spurred the development of programming models which are detached from the traditional rendering pipeline policy. Various interfaces for high-performance, data-parallel computations exist, among others NVIDIA's CUDA [17], AMD's CTM [18], Brook [4] and Sh [11] and their spin-offs *PeakStream* and *RapidMind*. All expose the intrinsic parallelism of GPUs to the user and provide means to perform general-purpose computations. This research area received increasing attention lately and the dedicated webpage at `www.gpgpu.org` gives an impression of the broad range of applications.

Efficiently programming GPUs imposes specific rules on the programmer: Algorithms need to be formulated in a way such that parallel execution is possible. Although this applies to all parallel languages, from now on we will focus on CUDA, which serves as a basis for our extended system. While the pure computational power of contemporary GPUs exceeds 380 GFlops in peak performance, the bottlenecks are the limited amount of available memory (1.5 GB for NVIDIA QuadroFX GPUs and 2.0 GB for AMD Firestream 9170 stream processors) and memory bandwidth: Challenging problems with data sets which do not fit into memory at once require the computation to be split and executed sequentially or they might introduce a significant communication overhead stressing the bus bandwidth. Fortunately, the internal scheduling and sequencing process is hidden from the programmer; however, it is necessary to handle multiple GPUs manually by creating threads for each GPU and by explicitly taking care of shared data transfer.

Our work addresses higher levels of parallelism and computations with large data sets: Our extended programming language, CUDASA, behaves similarly to a single-GPU CUDA system, but is able to distribute computations to multiple GPUs attached to the local system or to machines linked via network interconnects. Data-intensive computations, which would require sequential execution on a single GPU, can easily be parallelized to multiple GPUs and further accelerated by distributing the workload across cluster nodes. This distribution is realized with distributed shared memory provided by the CUDASA architecture. For this, a part of each node's main memory is dedicated for the shared memory range. Unfortunately, frequent accesses to this memory layer can cause significant communication overhead, which may even exceed the actual computation time on the computation nodes [23]. Therefore, we further extended the CUDASA framework and focused on a new scheduling mechanism that aims for data locality when paral-

- *All authors are with the Visualisierungsinstitut der Universität Stuttgart, Germany.*
E-mail: christoph.mueller@vis.uni-stuttgart.de

Manuscript received 25th June 2008.

lelizing the work over the network. Below the cluster level, our unified approach seamlessly integrates and exploits the intrinsic parallelism of GPUs – which is also reflected in CUDA and similar languages – and is thus able to provide a consistent development interface for a great variety of target configurations including inhomogeneous systems with single or multiple GPUs and bus or network interconnects.

Extending CUDA means that existing specifications and possibly restrictions transfer to our system. Nevertheless, we think that this makes it easier for programmers to equip their existing CUDA programs with multi-GPU support or to deploy their experience with CUDA to develop such programs.

The remainder of this article is organized as follows. The next section gives an overview about related work and GPU programming languages in particular. We recapitulate the details of the CUDASA programming model and the compile process in section 3. Providing parallelism across buses and networks is described in section 4 and 5. Section 5.1 introduces the new scheduling mechanism and illustrates the division of labor between the programmer and the system to make the scheduler data locality-aware. Finally, we analyze our system with different synthetic and real-world test cases on multi-GPU systems and a graphic cluster and provide detailed comparisons to the original CUDASA system.

2 Related Work

As indicated in the previous section, various options exist for performing general-purpose computations on GPUs (GPGPU). Several languages and interfaces have been especially designed for treating the GPU as a stream processor, and most of them build upon C/C++ and extend it with specific instructions and data types.

The high-level language programming of GPUs has been introduced with Sh [11] and C for Graphics [7] and later led to API-specific shader languages such as GLSL and HLSL. The increasing computational resources and flexibility of GPUs sparked the interest in GPGPU and specialized programming environments – besides traditional rendering APIs – have been developed: Brook [4] extends C with simple data-parallel constructs to make use of the GPU as a streaming coprocessor. Glift [10] and Scan Primitives [21] focus on convenient data structures and facilitate the implementation of various algorithms on GPUs. Moerschell and Owens [14] presented an implementation of distributed shared memory for multi-GPU systems, while Scout [12] takes one step further and provides modules for scientific visualization techniques. The probably most commonly used high-level GPGPU language is NVIDIA's CUDA [17], which serves as a basis for our work. It is in line with the aforementioned languages and extends C/C++ with parallel stream processing concepts. CTM [18] breaks ranks and provides low-level assembler access to AMD/ATI GPUs for hand-tuned high-performance computations.

Basically all of the aforementioned languages can be used to distribute computations across multiple GPUs, but – and this is an important motivation for our work – only if this is explicitly implemented and "hardwired" in the host application. None of them provides *language concepts* for parallelism on higher levels such as across multiple GPUs or even across nodes within a network cluster. Most related to the CUDASA project is the Zippy framework [5], which provides means to program GPU clusters using a distributed shared memory model and a CUDA-like computation kernel concept supporting several high-level shading languages as basic building blocks. However, Zippy exposes its functionality in the form of an API and a library rather than a programming language.

In the CUDASA project, we focus on the higher level parallelism and extend CUDA to enable multi-GPU and cluster computing with the goal of increased performance. Another use of parallel computations is to introduce redundancy for reliable computations, which has been investigated by Sheaffer et al. [22]. Both directions benefit from ROCKS clusters and CUDA Roll [16]: A live boot system which easily and quickly sets up network clusters with CUDA support.

The Cell Broadband Engine of Sony, Toshiba and IBM has a hierarchy of transient memory similar to modern GPUs: Each of the Synergistic Processing Elements is equipped with local memory, which is comparable with the on-board memory of a GPU, while the Power Processor element works on system memory like a conventional CPU. Sequoia [6] is a language extension to C++ that provides means of programming such a system by explicitly specifying the work on each memory hierarchy level. The programmer implements methods for a *task* object that control the workload distribution on higher levels and perform the actual computations on the lowest level.

Languages for stream processing on GPUs profit from experiences from parallel programming with CPUs and network clusters. This is a mature research area beyond the scope of this work. However, we want to mention STAPL [1], which provides containers and algorithms compatible with the C++ STL but supporting parallelization. They enable STL algorithms to run in parallel with minimal code changes. A parallel programming language worth mentioning is the Jade language extension to C [20], which requires the programmer to explicitly specify memory access patterns for code segments. Jade uses this information to determine possible concurrency on a per code segment base and generates deterministic parallel code. Although an automatic detection of parallelism is a desirable feature, we require the programmer to explicitly specify it due to our commitment to CUDA. To the interested reader, we recommend Bal et al.'s comprehensive overview [3] of parallel languages and their comparison of parallel programming paradigms [2] as further reading.

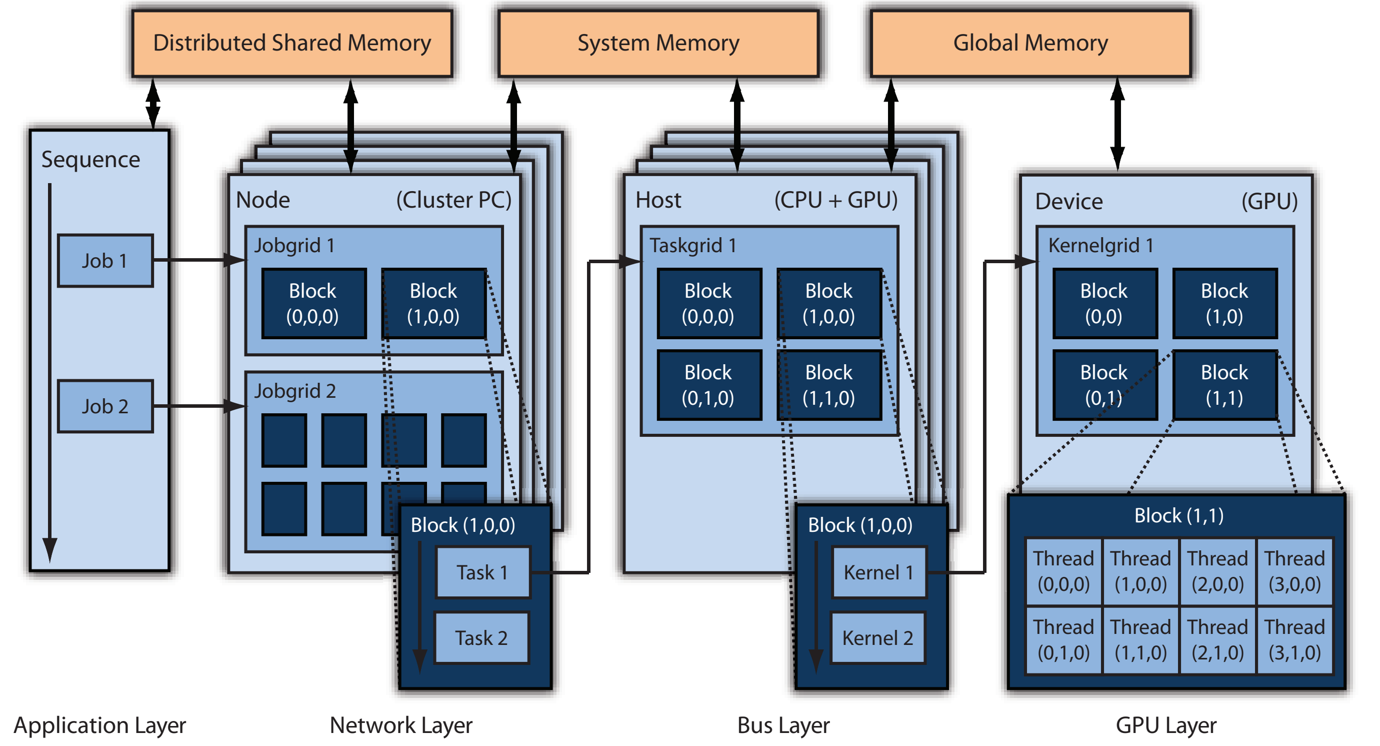

Fig. 1. Schematic overview of all four abstraction layers of the CUDASA programming environment. The topmost layer is placed left, with decreasing level of abstraction from left to right.

3 System Overview

In this section, we recapitulate the CUDASA programming environment and its programming model. Both are tightly coupled to the schematic overview in Figure 1 and we recommend referring to it while following the description.

3.1 Programming Environment

The CUDASA programming environment consists of four abstraction layers as depicted in Figure 1 from left (top layer) to right (bottom layer). Each of the three lower levels addresses one specific kind of parallelism: The lowest utilizes the highly parallel architecture of a single graphics processor, while the next higher level builds upon the parallelism of multiple GPUs within a single system. The third layer adds support for distributing program execution in cluster environments and enables parallelism scaling with the number of participating cluster nodes. Finally, the topmost layer represents the sequential portion of an application, which issues function calls executed exploiting the parallelism of the underlying abstraction levels. Each layer exposes its functionality to the next higher level via specific user-defined functions, which are declared using the extended set of function type qualifiers implemented in CUDASA. These functions are called using a consistent interface across all layers whereas each call includes the specification of an execution environment, i.e. the grid sizes, of the next lower level.

GPU Layer: The lowest layer (see Figure 1, right) simply represents the unmodified CUDA interface for programming GPUs. Existing CUDA code does not require any modifications to be used with our system – quite the contrary, it serves as a building block for higher levels of parallelism.

Bus Layer: The second layer (Figure 1) abstracts from multiple GPUs within a single system, for example SLI, Crossfire, Quad-SLI configurations, or single-box setups based on the QuadroPlex platform. A CPU thread together with a GPU forms a basic execution unit (BEU) called *host* on the bus layer. The programmability of these BEUs is exposed to the programmer through *task functions*, which are the pendants to *kernel functions* of the GPU layer. We follow the execution model of CUDA and define that a single call to a *task* consists of a grid of distinctive blocks. A scheduler distributes the pending workloads to participating *hosts* and also handles inhomogeneous system configurations, e.g. systems with two different GPUs or different number of physical PCIe lanes to the GPUs. The scheduling process works transparently to the user and the desirable consequence is that the application design is completely independent of the underlying hardware. For example, a once compiled CUDASA program is able to fully exploit the power of a QuadSLI system by executing four kernel blocks in parallel, while it processes blocks sequentially on a single-GPU system.

While the main focus of CUDASA is to provide easy access to multiple GPUs, the *bus layer* is also able to delegate tasks to CPU cores. This enables us to use CPU cores (in parallel if available) for tasks of a CUDASA program which cannot be executed on GPUs or for which the user wants the execution to happen on CPUs. Tasks, both with and without GPU support, can be used together in arbitrary combinations. We can also use CPU cores to emulate a system with multiple GPUs using the built-in device emulation provided by CUDA. The user specifies the operation mode (CPU only or CPU+GPU) of each task at compile time and optionally defines a maximum number of parallel devices to be used.

Network Layer: The third layer adds support for clusters of multiple interconnected computers. Its design is very similar to the underlying *bus layer*: A single computer, called *node*, acts as the BEU of the network layer and all nodes process blocks of the *jobgrid* (issued through a *job function*) in parallel. Again, the scheduling mechanism takes care of distributing the workload in both, homogeneous and inhomogeneous environments.

The difference to all underlying layers is that the network layer has to provide its own implementation of a distributed shared memory model in software. The distributed memory provides means to transfer data between blocks of a *jobgrid* and successive *jobs*. It can be considered as the pendant to the *global memory* in CUDA, which is used to transfer data between blocks and kernels. However, in contrast to GPUs, this memory does not exist as an "onboard component", but each node makes a part of its system memory available to the distributed shared memory pool. That is, the global address space that the shared memory exposes to the programmer is scattered across all nodes. Consequently, memory accesses have varying costs depending on whether the requested data is already resident on the requesting node. Therefore, the CUDASA scheduler, which originally assigned *jobs* arbitrarily to nodes [23], has been extended to do a data locality-aware scheduling if the programmer can declare the memory access patterns of the application at compile time.

Of course, the network layer can be omitted during program development as well as at compile time, whenever the program execution is targeted for a single-PC configuration only.

Application Layer: The topmost layer describes a sequential process, which issues calls to *job functions*. It also takes care of the (de-)allocation of distributed shared memory, which holds input and output data and is processed by the *nodes*. The distributed shared memory enables the processing of computations which exceed the available system memory of a single *node*. It is also the means of communication between the *sequence* and the *jobs* as well as between multiple *jobs*. A single *job* in turn can communicate with the *tasks* it spawns simply via system memory and each *task* communicates with the GPU by acting as a normal CUDA *host* and transferring data via *global memory*.

3.2 Programming Model

In this section, we describe the three main components of our extensions to the CUDA programming environment: A runtime library, a minimal set of extensions to the CUDA language itself, and the self-contained CUDASA compiler.

Runtime Library: The runtime library provides the basic functionality of *job* and *task* scheduling, distributed shared memory management, and common interface functions, such as atomic functions and synchronization mechanisms for all new abstraction layers. We implemented two versions, one with network layer support for cluster environments and one without, for single node execution.

Language Extensions: Our extensions to the original CUDA language solely introduce additional programmability for the higher levels of parallelism, while the syntax and semantics of the GPU layer remain unchanged. Hence, existing CUDA code does not require any manual modifications and can be used with CUDASA directly. For each new layer (bus, network, and application layer), CUDASA defines a set of function type qualifiers to specify a function's target BEU and its corresponding scope visibility. This is in line with the existing CUDA qualifiers `__device__` and `__global__` for functions executed on a graphics device and `__host__` functions acting as the front-end for CUDA device functions. Table 1 lists the CUDA and CUDASA keywords. As indicated there, each layer introduces specific built-in variables holding block indices and dimensions (Table 1, second column from right), each accessible to functions of the corresponding and the underlying layers. In order to allow the CUDASA network scheduler to work data locality-aware, the programmer must have the possibility to provide the required information about distributed shared memory usage for every *job function* invocation using some kind of source code annotation. These annotations are realized using three new keywords (Table 1, right column):

- `__descriptor__` is used to mark a function as code that determines the memory ranges used by a *job*.
- `__using__` resembles the standard C++ keyword and connects the aforementioned function with a pointer parameter of a *job function*.
- `__nomap__` is a keyword to control whether the compiler should generate code which automatically maps the memory.

Sec. 5.1 describes how these keywords are integrated into the CUDASA language. It is worth mentioning that the use of the keywords for declaring data locality is completely optional. That is, applications in CUDASA as described in [23] are transparently handled by the extended language without any changes. However, often the programmer is aware of the distributed shared memory access patterns of the *jobs* before the start of the *jobgrid* execution. The newly introduced scheduler

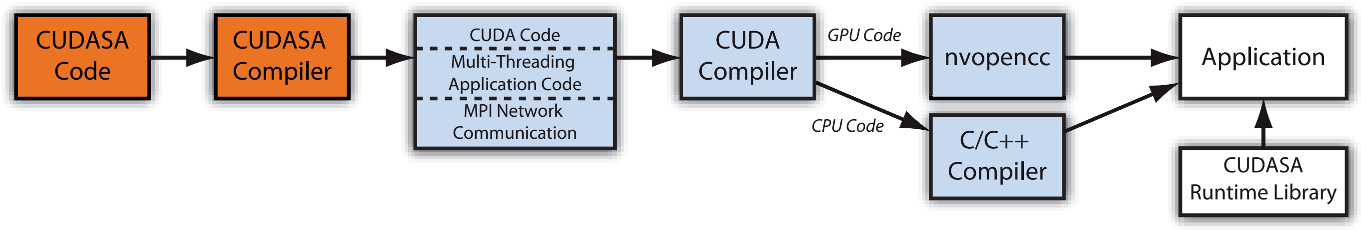

Fig. 2. The CUDASA compiler processes a program and outputs standard CUDA code, but also generates multi-threading and network code providing higher levels of parallelism. The CUDA compiler separates GPU and CPU code and hands it over to the corresponding compilers (nvopencc and any standard C/C++ compiler). The generated application executable, together with the CUDASA runtime library, is able to distribute workload across clusters and GPUs.

mechanism uses this information to improve scheduling in terms of remote network communication costs and consequently to improve the overall execution time.

Finally, CUDASA needs a way to link the abstraction layers and define function calls to the respective next-lower layer. Again, we follow CUDA and generalize its concepts to higher levels of abstraction: A CUDA function call requires the *host* level to specify an *execution configuration*, which includes the requested grid and block sizes for the parallel execution on the GPU. In an analogous manner, functions of each layer are allowed to call the exposed functions (see Table 1) of the next underlying layer. In order to maintain a consistent interface, we use the CUDA-specific parenthesized parameter list (denoted with <<< ... >>>) for the specification of the *execution configuration*.

Obviously, we limited our extensions to the CUDA language to a minimal set of new keywords. However, they provide powerful control over all levels of additional parallelism and enable the tackling of much more complex computations while keeping the additional programming and learning overhead for the user very low. Specifically, programming CUDASA *job* and *task functions* is very similar to CUDA *kernel functions* with respect to distributing the workload. All communication-related tasks are completely hidden from the user and are covered by the CUDASA runtime library and the compiler described next.

CUDASA Compiler: The last component of the CUDASA programming environment is the self-contained compiler, which processes CUDASA programs and outputs code which is then compiled with the standard CUDA tools (Figure 2). Although regular expressions can handle the new set of keywords, we cannot use them for the translation of CUDASA code to the underlying parallelization mechanisms. This requires detailed knowledge of variable types and function scopes and can only be obtained from a full grammatical analysis. The code translation process is described in detail in section 4 for the bus layer and in section 5 for the network layer.

CUDA itself exposes the C subset of C++ to the programmer, while some language-specific elements rely on C++ functionality, e.g. templated texture classes. CUDASA needs to act as a pre-compiler to CUDA including the ability to parse the header files of CUDA. Consequently, the CUDASA compiler needs to cope with the full C++ standard to translate the new extensions into plain CUDA code. We opted for building our compiler using Elkhound [13], a powerful parser generator capable of handling C++ grammar, and Elsa, a C/C++ parser based on Elkhound. We extended the compiler to support all CUDA-specific extensions to the C language as well as our extensions described in the previous paragraphs. The compiler takes CUDASA code as input and outputs code which is strictly based on CUDA syntax without any additional extensions. This means that the additional functionality exposed by CUDASA is translated into plain C code, which refers to functions of the CUDASA runtime library.

4 Bus Parallelism

The goal of bus parallelism is to scale processing power and the total available memory with the number of GPUs within a single system. For this, a *task* needs to be executed in parallel on multiple graphics devices, i.e. blocks of a *taskgrid* are assigned to different GPUs. CUDA demands a one to one ratio of processes or CPU threads to GPUs by design. Thus, each BEU of the bus layer has to be executed as a detached thread. Practically speaking, a *host* corresponds to a single CPU thread with a specific GPU device assigned to it.

Calling a *task* triggers the execution of the *host* threads and initializes the scheduling of the *taskgrid* blocks. A queue of all blocks waiting for execution is held in system memory. Idle *host* threads process pending blocks until the queue is empty, i.e. the execution of the complete *taskgrid* is finished. Mutex locking ensures a synchronized access to the block queue (this requires multiple operations in a single critical section), provides the necessary thread-safety, and also avoids a repeated processing of blocks on multiple BEUs. The block-threads are organized using a thread pool in order to keep the overhead for calling a *task* at a minimum. This is particularly important to avoid the costly initialization of CUDA for every function call.

Abstraction	Exposed	Internal	Built-ins	Additional Keywords
application layer		**`__sequence__`**		
network layer	**`__job__`**	**`__node__`**	**`jobIdx`**, **`jobDim`**	**`__descriptor__`**, **`__using__`**, **`__nomap__`**
bus layer	**`__task__`**	`__host__`	**`taskIdx`**, **`taskDim`**	
GPU layer	`__global__`	`__device__`	`gridDim`, `blockIdx` `blockDim`, `threadIdx`	

TABLE 1
The extended set of function type qualifiers of CUDASA. New keywords are printed bold-face. Internal functions are only callable from functions within the same layer, while exposed functions are accessible from the next higher abstraction layer. Built-ins are automatically propagated to all underlying layers.

A polling mechanism achieves load balancing on the block level across *hosts* as the actual execution time for each block implicitly controls how many of them are assigned to each BEU. This does not guarantee deterministic block assignment, but it does guarantee parallel execution, even for inhomogeneous setups, as long as enough pending blocks are left in the queue.

The automatic translation of code using the CUDASA interface into code which can be executed in parallel by multiple CPU threads handles the parameter passing, built-in variables, and the invocation of the *task* scheduler. Parameters and built-ins are grouped into a combined structure to meet the requirements of the underlying POSIX threads. The CUDASA compiler builds wrapper functions for each user-defined *task*, which perform the following steps:

- Copy the function parameters into the wrapper structure.
- Populate the queue of the scheduler with all blocks of the *taskgrid*.
- Determine the built-ins for each block.
- Wake up BEU worker threads from the pool.
- Wait for all blocks to be processed (issuing a *taskgrid* is a blocking call).

Additionally, the signature of a *task* is modified internally to accept the wrapper structure. The parameters as well as built-ins are then reconstructed from the data structure.

The following simplified example demonstrates the code transformation of a function definition: The compiler translates the definition of a *task*, written in CUDASA code

```
__task__ void tfunc(int i, float *f) {...}
```

into the following internal structure and modified function:

```
typedef struct {
    int i; float *f;  // user-defined
    dim3 taskIdx, taskDim;  // built-ins
} wrapper_struct_tfunc;

void tfunc(wrapper_struct_tfunc *param) {
    int     i = param->i;
    float *f = param->f;
    dim3 taskIdx = param->taskIdx;
    dim3 taskDim = param->taskDim;
    {...}
}
```

Please note that the semantics of pointer-typed parameters is consistent with the CUDA parameter handling and the validity of pointer addresses is ensured. The result of the transformation is plain CUDA code and can be passed into the standard CUDA tool chain.

CUDASA also adds support for atomic functions on the bus parallelism level to enable thread-safe communication between multiple *task* invocations. The implementation of those atomic functions is straightforward using the lock instruction in assembler code.

5 Network Parallelism

The network layer is very similar to the bus layer, not only conceptually, but also regarding its implementation. *Jobs* are the pendants of the *tasks* on the bus layer. The difference is that *jobgrids* are not executed by operating system threads, but on different cluster nodes. The parallelization on the network level is implemented using MPI2 as we require remote memory access capabilities for implementing our distributed shared memory manager. Invocations of a *job* (issued by the application running on the head node) are translated by the CUDASA compiler into a broadcast instructing all nodes to run a *job*. The transfer of function parameters is realized – analogously to the bus layer – by packing them together with the built-in variables, e.g. the `jobIdx`, into a structure and handing it over to the network.

In order to listen for function calls, all nodes except for the head node run an event loop waiting for broadcast messages. The corresponding code is automatically generated by the CUDASA compiler and includes the parameter serialization for all *jobs*. Besides the invocation of *jobs*, the event loop also handles collective communication operations required for the distributed memory manager of CUDASA described next.

5.1 Distributed Shared Memory

The network layer of CUDASA allows for computations which do not fit into the main memory of a single node. By design, the head node solely controls the job distribution and does not participate in any computation. Please note that the head node of course can run as a thread on any node within the cluster. Each other cluster node makes a part of its memory available to the distributed shared memory pool, which is exposed as a continuous virtual address range to the application. Allocations in shared memory are split into evenly sized segments and one is stored on each node.

Access to distributed shared memory is not opaque via variables and a paging mechanism: The programmer explicitly requests specific memory ranges to be cloned from the Global Partitioned Address Space to a node as it is the case with Global Arrays [15]. CUDASA provides `memcpy`-style functions for accessing shared memory from the head node and special mapping functions for all other nodes. The mapping functions also handle the write-back for mappings that are not read-only when the mapping is closed.

Besides explicitly mapping shared memory segments using the corresponding API functions in a *job function*, CUDASA now allows to declare which distributed memory segments are required for a specific *job* at compile time. For this, the programmer provides an additional *memory descriptor function* (MDF) that returns offset and size of the memory range to be mapped. MDFs are identified using the `__descriptor__` modifier introduced with our latest extensions to CUDASA and must comply with a pre-defined signature: They take a *job* index and the grid dimension as input and return the aforementioned list of mapping constraints. When defining a *job function*, any parameter which is a pointer to distributed shared memory (`dsm` in the following example) can be marked for automatic mapping using a function. We opted for specifying a function pointer in the definition rather than an expression (directly describing the memory ranges), for the sake of readability. Otherwise, even the simple mapping description in the following example would have totally cluttered up the definition of `jfunc`.

```
__descriptor__ void memDesc(
        unsigned int *outStart,
        unsigned int *outSize,
        dim3 idx, dim3 grid) {
    *outStart = (idx.y * MATRIX_SIZE
        * JOB_SIZE + idx.x
        * JOB_SIZE * JOB_SIZE
        * sizeof(float);
    *outSize = JOB_SIZE * JOB_SIZE
        * sizeof(float);
}

void jfunc(__using__ memDesc float *dsm);
```

The descriptor `memDesc` maps from a logical, two-dimensional, quadratic data field to uniformly sized blocks, which are stored linearly in distributed shared memory. For simplicity of the code example, the total data size is assumed to be a multiple of the size of the sub-blocks.

The CUDASA compiler uses an MDF in a parameter declaration to automatically map the requested memory range for each *job* execution before the actual user code is executed. In fact, the compiler inserts the same memory mapping instructions as the user would have to specify within the function body before the actual implementation of the *job function*. The pointer to the distributed shared memory which is passed as parameter to the function is replaced with the pointer to the locally mapped memory on the node. The automatic mapping is valid as long as the current invocation of a *job function* is being executed. Consequently, the compiler inserts the corresponding unmapping instructions where the control flow may leave the user-defined function body. By using this new language construct of CUDASA, access to distributed shared memory becomes completely transparent within a *job function*. The benefit for the programmer is obvious: In contrast to prior CUDASA versions, the implementation of the algorithm and the data selection are now separated. However, it remains possible to choose the memory access paradigm individually for each parameter.

If a formal parameter is declared as a pointer to a constant memory range, then the memory is regarded as read-only and all changes made to the mapped memory are not written back to distributed shared memory, but discarded after execution. This behavior mimics the unmapping API functionality from the CUDASA runtime library to discard data.

In addition, the CUDASA compiler has the option to delay the unmapping of automatically mapped distributed memory segments until the current node enters the next *job* invocation. At this point, the system can determine whether the mapping of the prior invocation may be re-used. This allows CUDASA to decrease the network traffic for (un-)mapping operations with applications accessing the same input data from multiple *jobs*. However, this behavior is optional and recommended solely for read-only mappings. Furthermore, it is only beneficial if either all *jobs* are using the same part of distributed memory, or if all *jobs* doing so are executed consecutively. At the moment, the CUDASA scheduler does not guarantee this. It solely uses the declared access patterns prior to a *job* invocation to optimize the workload distribution based on data locality.

This is implemented in a new CUDASA runtime library function, which evaluates the MDF for all parameters of a *job* and for all *jobs* of a grid. The compiler inserts a call to this function at the start of the execution of a *jobgrid* on each node. Using the information on per-*job* shared memory accesses, a node is able to compute which of these *jobs* have the most overlap of required memory ranges with locally resident parts of the distributed shared memory. It communicates these *jobs* as its preferred work items to the head node. Thus, instead of waiting for the head node to assign an arbitrary *job* to an idle BEU, the node formulates a list of preferred *jobs* from which the head node can choose.

If the head node cannot fulfill the request, e.g. because all requested *jobs* have already been assigned to other nodes, it orders the requesting node to send another packet of work items to choose from until an appropriate one was found. By choosing a reasonable number of *jobs* offered to the head node, e.g. the number of nodes in the cluster, such message round trips can be effectively reduced to a minimum in most cases. Our experiments showed that the head node is able to fulfill the requests from the first list in most cases. Multiple messages are usually required only if the algorithm is prone to load imbalance caused by the computational complexity rather than by distributed memory accesses. In these situations, the head node cannot serve a client immediately when the execution of the *job* grid ends: All *jobs* using distributed memory segments resident on the requesting node have already been processed by other nodes, e.g. because they have been faster in processing their preferred work items. In this case, the master requests the node to prepare a new list of preferred items.

There are mainly two reasons for a node not being able to provide a list of preferred *jobs* for the head node. First, if all *jobs* using memory which is locally resident on the node have already been completed. And second, if the *job function* does not use distributed shared memory which is mapped using an MDF. In both cases, the *job* assignment is done arbitrarily. Especially for high-speed interconnects, it is favorable to prevent nodes from becoming idle (even if none of the work items uses locally present data) than strictly sticking to the optimal access patterns. For increased flexibility, we introduced a mechanism which allows to give hints to CUDASA about the required distributed shared memory segments without mapping them automatically. This is convenient for cases such as the one illustrated in the following example: Imagine a *job function* accessing a large amount of memory, which cannot be mapped as a whole, but which will be accessed sequentially using the mapping and unmapping functions provided by the CUDASA runtime library. The hints and information given to CUDASA can be used to optimize the *job* scheduling, although the mapping is done manually. For this, CUDASA provides the `__nomap__` keyword: If a `__using__` directive for MDFs is followed by `__nomap__`, then the descriptor function is used during the computation of the per-node memory overlap, but the distributed shared memory pointer is not mapped automatically. The user has to map and unmap the pointer using the mapping functions in the *job function* as if no `__using__` directive had been specified.

CUDASA's shared memory manager is implemented using the MPI Remote Memory Access (RMA). It distinguishes between two classes of operations: Collective and single-sided. Allocation and deallocation, which may only be invoked from the head node, are collective operations and therefore must be executed by all nodes at the same time. For collective operations, the head node posts a corresponding request into the event loop of all nodes. This is necessary as we need to ensure a consistent view of allocations across all nodes and this reflects in MPI as well: All nodes accessing a memory window need to be involved in the (de-)allocation process.

Access to existing allocations is fully single-sided on both, the head and the compute nodes. With the coherent view on the global allocation state, all nodes can access, lock, and read from/write to distributed shared memory (through `MPI_Get` and `MPI_Put`). We group these operations for each memory segment (remember, one segment resides in the local memory of one node). Thus, an access to memory ranges spanning more than one segment is not atomic. This could be achieved with a two-phase locking protocol at the expense of greatly slowing down the accesses. For the sake of speed, CUDA does not make any guarantee regarding concurrent accesses to global memory and we decided to adopt this for CUDASA's shared memory manager as well.

5.2 Atomics

CUDASA also extends the concept of atomic functions to distributed shared memory. They are implemented using the memory window locking mechanism of MPI2. Several preconditions for atomic functions must be met to avoid a two-phase locking protocol for multiple segments: Firstly, atomic functions are only allowed for 32-bit words, which must be aligned on a word boundary within the allocation. This is a reasonable constraint which normally also applies to atomic operations in main memory. And secondly, each aligned word must not span across segment boundaries. This precondition can be easily enforced by CUDASA using a segment size of multiple of the word length.

For most atomic functions, we can limit the communication cost to a single `MPI_Accumulate` call (with the corresponding operation code) – in the worst case an `MPI_Get`/`MPI_Put` pair within an exposure epoch suffices.

6 Discussion and Implementation Details

In this section, we want to point out interesting aspects which deserve discussion. In particular, the newly-introduced layers raise new questions on synchronization, distributed shared memory, and the compile process.

Synchronization: CUDA offers synchronization of threads within a single block, but synchronization between blocks is not possible. This is due to the fact that only a limited number of blocks can be executed in parallel: Block synchronization would require the suspension of blocks and storing of their complete state. Only in doing so, all blocks can be executed until they reach the synchronization barrier. Due to limited onboard memory, this would imply a high memory transfer overhead and thus simply becomes impractical. Basically the same holds for higher levels of parallelism. Although it would be easy to provide a synchronization mechanism for blocks within a *taskgrid* (and analogously for *jobgrids*), storing the state of a single block, e.g. after the execution of a kernel, means that potentially the total memory of a GPU needs to be transferred to the host and back to the GPU.

Scheduling: Especially on large clusters, which run huge *jobgrids*, the sequential computation of preferred work items may take a prohibitively long time as it requires the MDF being evaluated for each possible *job*. Therefore, we opted to implement this computation in CUDA and run it on the GPU. The evaluation of the descriptor function fits extremely well into the CUDA architecture as the only input data required consists of the base distributed memory pointers that have been passed as actual parameters to the *job*. These require a very limited amount of memory and can therefore be stored in CUDA's shared memory. The actual evaluation of the descriptor function requires only access to shared memory and a large number of arithmetic operations, and thus benefits from the large number of stream processing units. Without this large degree of parallelism, *job* grids of millions of entries would be impractical. However, this implementation poses the same restrictions on `__descriptor__` functions that are defined for CUDA `__device__` functions. As long as the required memory segments can be computed using rather simple expressions, this does not turn out to be a problem for the practical use of our system.

To perform the actual automatic mapping of distributed shared memory, the memory descriptor function must be evaluated once more directly before calling a *job* invocation. In this case, the descriptor is only called for a single *job* index and therefore it is not advisable to use CUDA here as setting up the environment for evaluating a single expression introduces a high overhead. Hence, we execute the memory descriptor function on the CPU in this case.

Scheduling on the network layer, as described above, does not scale optimally for very large cluster environments as the job queue is solely managed by the head node. A large number of idle nodes asking for new *jobs* simultaneously may congest the network communication with the *job* queue and hamper parallel *jobgrid* execution. Hierarchical load balancing within a network or the assignment of multiple *jobs* per query bypasses this bottleneck. A head node can reasonably estimate the number of blocks to be processed based on the number of compute nodes and the size of the grid. However, this approach has negative impact on the effectiveness of the load balancing.

Compilation: The CUDA compile process itself introduces specific preconditions on the CUDASA environment. The separation of code fragments for CPU and GPU execution is partly based on specific comments in CUDA headers. Hence, any pre-compiler to CUDA is required to maintain these code lines in contrast to the common compiler behavior of ignoring comments. More important, it is worth to note that the current CUDA compiler does not support exception handling. Consequently, CUDASA requires an MPI implementation that does not use this language concept. In our work, MVAPICH2 [8] has been used with the necessary flags set accordingly.

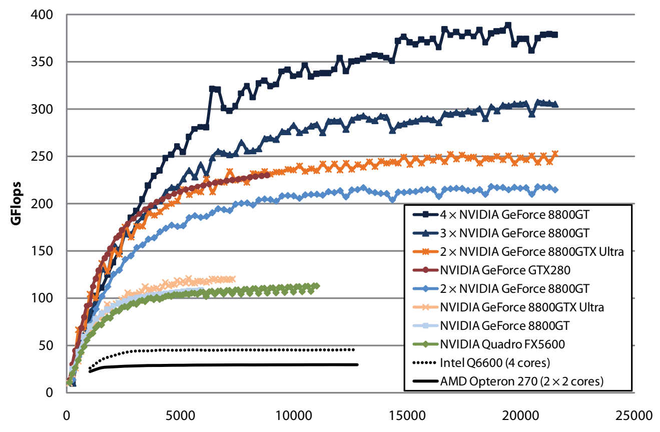

Fig. 3. Illustration of the SGEMM scaling behavior with CUDASA for various system configurations on the *bus* layer. With SGEMM, we achieve nearly optimal scaling with the number of GPUs. The slight performance fluctuations in multi-GPU configurations stem from the imperfect load balancing (see Section 6).

7 Results

For *bus* parallelism, we evaluate scaling behavior of CUDASA applications on up to four GPUs in a single machine for a variety of problem sizes. For network parallelism, we show scalability on up to eight machines for different applications.

7.1 Bus Parallelism

In order to compare performance and efficiency of CUDASA generated code to other parallel execution environments, i.e. multiple CPU cores and the intrinsic parallelism of a single GPU, we use the single precision general matrix multiply (SGEMM) subroutine of the level 3 BLAS library standard. For each processor, the vendor-specific performance-optimized implementation is used to guarantee optimal usage of each hardware. Namely, we use Intel's Math Kernel Library 10.0 (MKL), the AMD Core Math Library 4.0 (ACML), and NVIDIA's CUBLAS Library 1.1. The measurements on the GPU are performed using CUDA version 1.1 for Linux (display driver version 169.04).

Our CUDASA implementation of SGEMM uses CUBLAS as building block for the *task* level. The workload distribution on the upper levels employs the same block building approach as used in NVIDIA's matrix

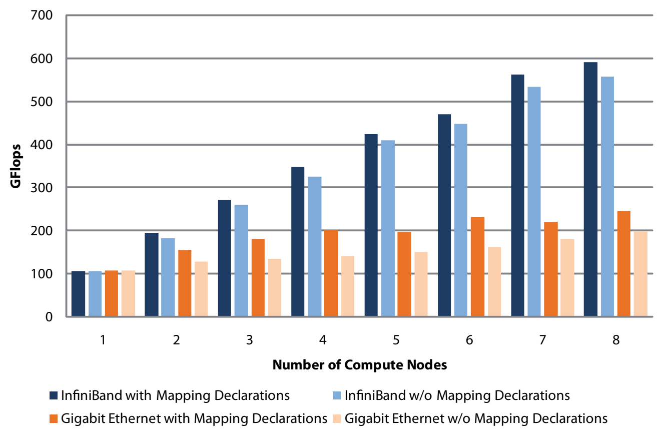

Fig. 4. Illustration of the SGEMM scaling behavior using the CUDASA *network* layer. The differences in the overall performance depict the improvements of the new locality-aware scheduler.

multiplication example [17] with increasing sub-problem sizes.

Figure 3 summarizes the results for SGEMM *bus* level parallelization (colored lines) compared with the above-mentioned CPU implementations (gray lines). Our measurements on the multi-GPU systems demonstrate excellent scaling behavior for both test cases, two 8800GTX Ultra (orange lines) and up to four 8800GT (blue lines) cards, especially for large problem sizes. In the former setup, we achieve a speedup of 1.95 when comparing pure CUBLAS running on one GPU with our CUDASA implementation using two cards. In the latter case, distributing the work over all four cards results in a speedup of 3.60. Please note that a better scaling with the second setup is hindered by the physical PCIe lanes of the mainboard, which offers a 16/4/4/4 layout only.

7.2 Network Parallelism

We test the network layer of CUDASA on an eight node GPU cluster with an SDR InfiniBand interconnect. Each node is equipped with an Intel Core2 Quad CPU, 4 GB of RAM and an NVIDIA GeForce 8800GTX GPU. The cluster tests are performed using CUDA 1.1 and the Linux display driver version 173.14.

SGEMM also serves as first test case for the network layer. As with the *bus* level parallelism test, we use the CUBLAS library as building block for our test program. However, as our cluster nodes do not have multiple GPUs, we chose to omit the *task* layer and let the *jobs* work directly on the graphics card for this test. We run the SGEMM test with $16,000^2$ sized matrices on two to eight physical machines with one to eight computation nodes. In order to carry out the same test in all cases, we have to restrict the matrix size such that the computation can be performed with the amount of distributed shared memory provided by a single compute node. Note that one of the machines serves as dedicated head node, except for the case of eight computing nodes: Here, the head node and one compute node are executed on the same physical machine. This machine only contributes a single share to the distributed memory (in its role as a compute node). Figure 4 shows the scaling behavior of CUDASA's cluster level for both InfiniBand interconnect (blue bars) and Gigabit Ethernet (orange bars). The diagram also highlights the impact of our new locality-aware scheduling mechanism. Except for using only one compute node – when minimizing of network traffic between the compute nodes is irrelevant – using memory descriptor functions generally performs better than arbitrary work item assignments. Note that the memory descriptors used for these benchmarks are very similar to the example given in section 5.1.

We also compared the distributed memory transfer volumes of the original memory access method (Figure 5) and the new locality-aware scheduling mechanism (Figure 6). The new scheduler significantly reduces the amount of data transferred via the network. While being obvious that a smaller number of computation nodes increases the probability that the required data already resides in a local distributed memory share, the data transfer with five compute nodes is – on first sight – unexpectedly low (Figure 6). The explanation is that the $16,000^2$ test matrix is subdivided into *jobs* of $3,200^2$ elements yielding a grid of 5×5 *jobs*. As the distributed shared memory allocations for the matrices are also evenly distributed to all five compute nodes, it becomes possible to store them such that the result data blocks for each *job* are exactly aligned with the allocation segments.

Also for the general case where only "non-optimal" scheduling is possible, the locality-aware scheduling mechanism significantly reduces the network traffic and consequently improves the overall system performance. For example, for three compute nodes, the traffic is reduced to two thirds of the original 25 GB. The average reduction (excluding the special case of one node) is about 25 percent for the SGEMM test case.

7.3 Path Tracing

Path tracing [9] solves the rendering equation numerically by effectively computing a probabilistic estimate of the light intensity flowing to the eye point along a light path. For this, a large number of light paths per pixel is traced, which makes path tracing a second (and real world) example for CUDASA's network layer. We implemented an iterative path tracer, which uses a uniform grid as acceleration structure. Geometry (triangles, vertices, normals) and materials are handed over to the CUDA *kernels* in one-dimensional textures.

The primary rays from the view point are randomly shot through pixels and local illumination is evaluated at intersections with the scene geometry using the Phong model. For this, we trace a shadow ray to a randomly chosen location on a light emitting triangle and sample

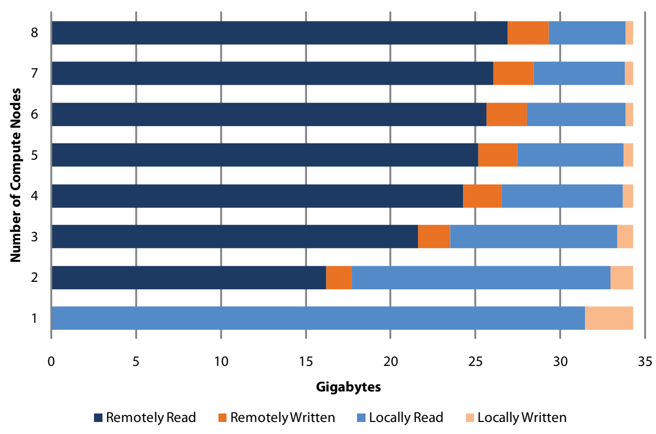

Fig. 5. Total data transfer from and to distributed shared memory for SGEMM on a $16,000^2$ matrix without using *memory descriptor functions*. CUDASA therefore can only assign work items arbitrarily to the compute nodes.

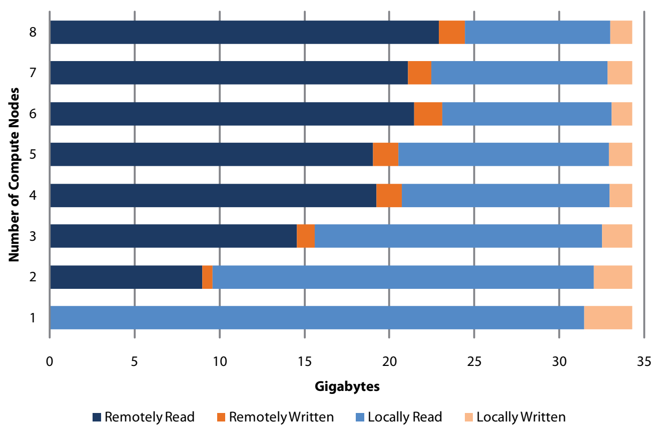

Fig. 6. Total data transfer from and to distributed shared memory for SGEMM on a $16,000^2$ matrix when automatically mapping the result matrix and declaring access to the other matrices in advance. The new scheduling mechanism greatly reduces the network traffic in this case.

the bi-directional reflectance distribution function to determine the reflected ray and thus the next segment of the light path. As generating random numbers on the GPU is costly, we store precomputed pseudo-random numbers in a texture and copy them to shared memory as a first step in the path tracing CUDA *kernel*.

The workload distribution to different *jobs* and *kernels* takes place in image space. The *job* size specifies the number of output pixel lines and the *job* invocation generates a thread block to always render 64 contiguous pixels. Each *kernel* thread computes one image pixel by tracing all paths which contribute to the pixel color.

We rendered the crank data set (310,000 triangles), surrounded by a box and illuminated by an area light source on the ceiling, with our path tracer for benchmarking (Figure 11). The surface materials exhibit varying material properties (colors and glossiness). The uniform grid acceleration is chosen for simplicity and obviously is sub-optimal for such scenes: Most of the 128^3 cells are empty, while cells enclosing finely tessellated regions contain 900 and more triangles. All renderings have been computed at a resolution of $1,024 \times 1,024$ using 150 rays per pixel and restricting the light paths to six indirect bounces.

Figure 7 shows the results of the scaling behavior: We chose the number of *jobs* equal to the square of the number of compute nodes. By this, we generate a sufficiently large number of *jobs* to observe load balancing effects. The overall scaling behavior matches our expectations considering the increasing number of distributed shared memory accesses with an increasing number of *jobs*. However, there are two anomalies worth mentioning: In the case of six and eight compute nodes, the computation takes an unexpectedly long time. This is due to the fact that good scaling highly depends on the actual GPU computation time being evenly distributed to the cluster nodes. A node may have a disproportionate influence on the overall execution time, e.g. if a node receives a computationally expensive *job* while the other nodes are just finishing their last *job*. CUDASA's scheduling mechanism has no possibility to avoid such an unfortunate *job* assignment as its decisions are solely based on the memory access specifications and not on estimations of the computational complexity of a *job*. Figure 8 illustrates this: When using six nodes, the slowest (green) node takes nearly twice the time of all other node. For eight nodes, this becomes even more obvious for the red node.

Increasing the number of *jobs* by decreasing their size can mitigate this problem, as costly *jobs* are more probably distributed across different nodes. We did not include measurements with manually optimized *job* sizes in Figure 7 as this would disguise the pure scaling behavior of the system.

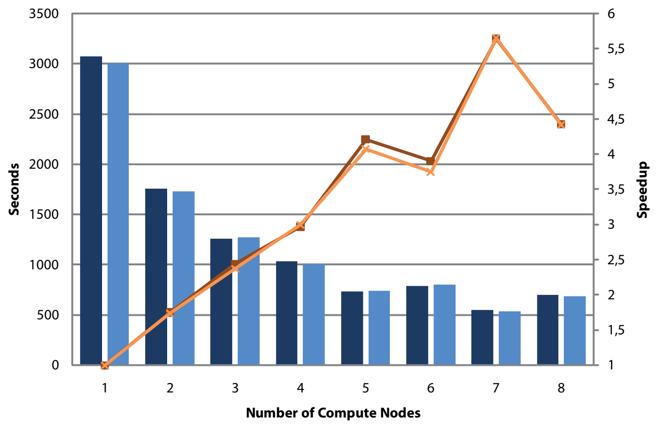

Fig. 7. Overall rendering time and speedup of the path tracer rendering the crank model on different numbers of nodes. The dark-colored bars and lines show the standard unmapping behavior, while the light-colored ones depict the delayed unmapping feature.

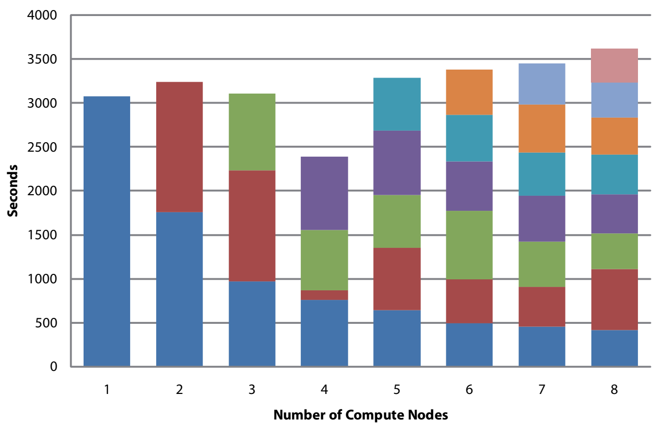

Fig. 8. Contribution of every node to the accumulated computation time of the path tracer. In particular when using six and eight compute nodes the slowest node (green, respectively red) takes much more time than the others, which reduces the overall performance.

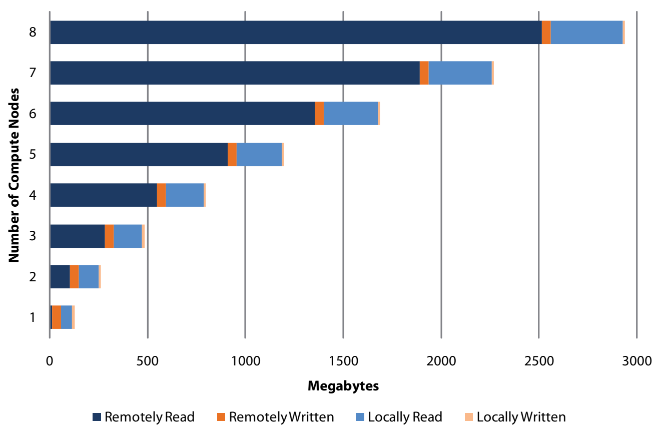

Fig. 9. Total data transfers from and to distributed shared memory for the path tracer. The number of *jobs* was chosen to be the number of compute nodes squared for this test.

Of course, the accumulated execution time (Figure 8) also depends on the amount of distributed shared memory copied for each *job* invocation and on degree of data locality. Figure 9 shows that the amount of data which has to be read remotely swiftly increases with the number of nodes. This is due to the fact that the complete scene has to be read for each *job*. However, the influence of the distributed shared memory transfers on the overall computation time is rather small for a path tracer, because of the high complexity of the computations. Figure 10 underlines this interpretation as it shows the significantly reduced data transfer due to the delayed unmapping feature of the CUDASA compiler. As all *jobs* use the same scene data, the mapping of the first *job* can be re-used in all consecutive ones. Decreasing the *job* size to only a few thousand pixels minimizes the load imbalance to about 1 % (compared to 30 % in other configurations). However, it does not yield the best overall performance, mainly because the GPU power cannot be fully exploited anymore. This is in line with the results of our SGEMM scaling tests on the bus level (Figure 3), which show that a certain amount of parallel work for each GPU is required for optimum performance.

Obviously, there is a trade-off between load balancing on the one hand, which can be achieved using more *jobs*, and the increasing time required for the remote memory accesses and decreasing GPU utilization on the other hand. A second series of measurements was performed to investigate the influence of different *job* counts using eight nodes (Figure 10).

8 Conclusions

We introduced a new version of CUDASA, an extension to CUDA, to achieve higher levels of parallelism. Only few additional language elements are required, thus keeping the programming and learning overhead for the user very low. We showed that this allows for tackling computations which are too large for single-GPU CUDA-programs and demonstrate that our system shows the expected, and desirable, scaling behavior. The new, data locality-aware scheduling mechanism proved to be useful to increase the performance on the cluster level – particularly for low-bandwidth networks like Gigabit Ethernet – as long as the algorithms allow the programmer to specify the memory access patterns and load imbalance is caused by distributed memory accesses rather than by the computational load. However, especially in the context of GPU programming, specifying the access pattern usually does not add significant overhead, as it normally only requires shifting the computation of required memory ranges out of the *job function* into a separate descriptor function. For maximum flexibility, the programmer can opt on a per-parameter basis whether

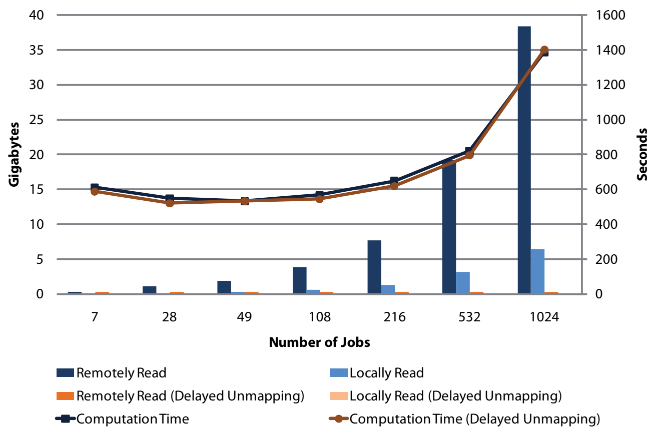

Fig. 10. The rendering time of the path tracer with eight nodes depends on the number of *jobs* and thus indirectly on the quality of load balancing, data traffic and GPU utilization.

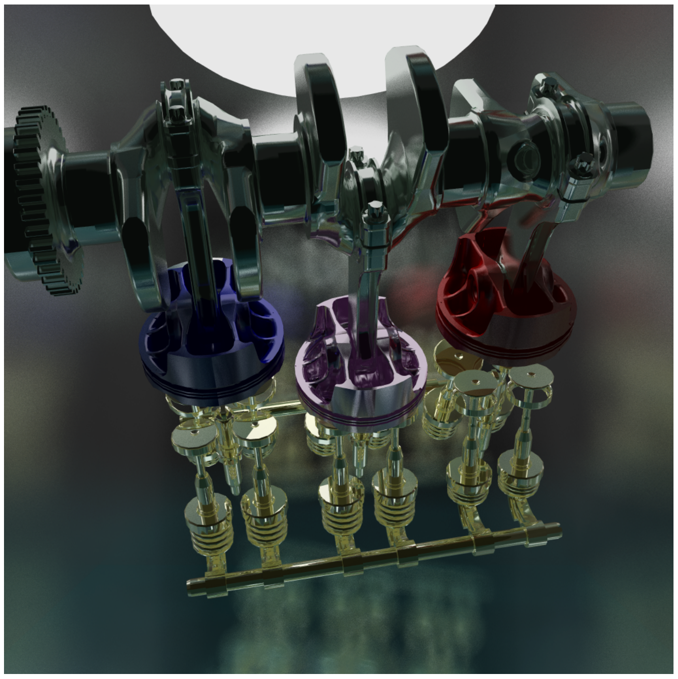

Fig. 11. Final rendering result of the crank model with 150 rays per pixel and a maximum of six ray bounces.

this is possible and to which extend the system utilizes the descriptor function. As shown exemplary for two different test scenarios, the system performs well for multi-GPU systems and cluster setups.

Applications using the same input data for all *jobs*, such as our path tracer, could further benefit from a concept of distributed constant memory that is filled and copied to all nodes once before the start of the *jobgrid* and remains constant over the whole grid execution. This concept would provide more control over the data distribution than the current automatic delayed unmapping to the programmer, but remains for future work. Furthermore, we would like to extend the scheduler to group *jobs* that actually benefit from the delayed unmapping for consecutive execution. New versions of CUDASA might also profit by notification events informing the user about the completion of jobs. This allows for the implementation of more complex patterns than the current synchronous execution of *task-* and *jobgrids*.

ACKNOWLEDGMENTS

The crank data set is courtesy of MAHLE International GmbH, Stuttgart, and Madness GmbH, Göppingen. This work has partially been financed by the state of Baden-Württemberg in context of the BW-FIT research initiative. We want to thank Guido Reina for his assistance in setting up our graphics cluster and the anonymous reviewers for the valued comments.

REFERENCES

[1] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. M. Amato, and L. Rauchwerger. STAPL: A Standard Template Adaptive Parallel C++ Library. In *Proceedings of the International Workshop on Advanced Compiler Technology for High Performance Embedded Systems*, 2001.

[2] H. E. Bal. A Comparative Study of Five Parallel Programming Languages. *Future Gener. Comput. Syst.*, 8(1-3):121–135, 1992.

[3] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Comput. Surv.*, 21(3):261–322, 1989.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[5] Z. Fan, F. Qiu, and A. E. Kaufman. Zippy: A Framework for Computation and Visualization on a GPU Cluster. *Computer Graphics Forum*, 27:341–350, 2008.

[6] K. Fatahalian, D. Reiter Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[7] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Publishing Co., 2003.

[8] W. Huang, G. Santhanaraman, H. W. Jin, Q. Gao, and D. K. x D. K. Panda. Design of High Performance MVAPICH2: MPI2 over InfiniBand. *ccgrid*, 00:43–48, 2006.

[9] J. T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, 1986.

[10] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Trans. Graph.*, 25(1):60–99, 2006.

[11] M. D. McCool, Z. Qin, and T. S. Popa. Shader Metaprogramming. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 57–68, 2002.

[12] P. McCormick, J. Inman, J. Ahrens, J. Mohd-Yusof, G. Roth, and S. Cummins. Scout: A Data-Parallel Programming Language for Graphics Processors. *Parallel Comput.*, 33(10-11):648–662, 2007.

[13] S. McPeak and G. C. Necula. Elkhound: A Fast, Practical GLR Parser Generator. In *Proceedings of the Compiler Construction (CC'04)*, pages 73–88, 2004.

[14] A. Moerschell and J. D. Owens. Distributed Texture Memory in a Multi-GPU Environment. In *Proceedings of the 21st ACM SIGGRAPH/Eurographics symposium on Graphics hardware*, pages 31–38, 2006.

[15] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A Portable Shared-Memory Programming Model for Distributed Memory Computers. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 340–349, 1994.

[16] NVIDIA. CUDA for Rocks Cluster User Guide. http://developer.nvidia.com/ object/cuda_1_0.html, 2007.

[17] NVIDIA. CUDA Programming Guide. http://developer.nvidia.com/object/cuda.html, 2007.

[18] M. Peercy, M. Segal, and D. Gerstmann. A Performance-Oriented Data Parallel Virtual Machine for GPUs. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 184, 2006.

[19] K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan. A Real-Time Procedural Shading System for Programmable Graphics Hardware . In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 159–170, 2001.

[20] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A High-Level, Machine-Independent Language for Parallel Programming. *Computer*, 26:28–38, 1993.

[21] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, 2007.

[22] J. W. Sheaffer, D. P. Luebke, and K. Skadron. A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 55–64, 2007.

[23] M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl. CUDASA: Compute Unified Device and Systems Architecture. In *Proceedings of Eurographics 2008 Symposium on Parallel Graphics and Visualization (EGPGV'08)*, pages 49–56, 2008.

PLACE
PHOTO
HERE

Christoph Müller received his diploma in computer science from the Universität Stuttgart, Germany. He is now a PhD student at the Visualization Research Center of the Universität Stuttgart (VISUS). His research interests include graphics clusters and tiled displays.

PLACE
PHOTO
HERE

Steffen Frey is a PhD student at the Visualization Research Center of the Universität Stuttgart (VISUS). His current research interests include the reconstruction and rendering of large volume data sets as well as GPU clusters and CUDA. He received his diploma in computer science from the Universität Stuttgart.

PLACE
PHOTO
HERE

Magnus Strengert is a PhD student in the Visualization and Interactive Systems Group at the Universität Stuttgart, Germany. His current research interests include volume rendering, GPU-based raycasting, and parallel visualization using GPU cluster computers. He has a MS in computer science from the Universität Stuttgart.

PLACE
PHOTO
HERE

Carsten Dachsbacher is an assistant professor at the Visualization Research Center of the Universität Stuttgart, Germany. He received the diploma in computer science and a PhD in computer graphics, both from the University of Erlangen-Nuremberg. He has been researcher at INRIA Sophia Antipolis, France, within a Marie-Curie Fellowship, and visiting professor at the Konstanz University. His research focuses on real-time computer graphics, interactive global illumination, GPGPU, and perceptual rendering.

PLACE
PHOTO
HERE

Thomas Ertl is a full professor of computer science at the Universität Stuttgart, Germany, and the head of the Visualization and Interactive Systems Institute (VIS) and the Visualization Research Center (VISUS). He received a MS in computer science from the University of Colorado at Boulder and a PhD in theoretical astrophysics from the University of Tübingen. His research interests include visualization, computer graphics, and human computer interaction.