

CUDASA: Compute Unified Device and Systems Architecture

M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl

Visualization Research Center (VISUS), University of Stuttgart

Abstract

We present an extension to the CUDA programming language which extends parallelism to multi-GPU systems and GPU-cluster environments. Following the existing model, which exposes the internal parallelism of GPUs, our extended programming language provides a consistent development interface for additional, higher levels of parallel abstraction from the bus and network interconnects. The newly introduced layers provide the key features specific to the architecture and programmability of current graphics hardware while the underlying communication and scheduling mechanisms are completely hidden from the user. All extensions to the original programming language are handled by a self-contained compiler which is easily embedded into the CUDA compile process. We evaluate our system using two different sample applications and discuss scaling behavior and performance on different system architectures.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Languages C.2.4 [Computer-Communication Networks]: Distributed applications

1. Introduction

Each new generation of GPUs provides flexible programmability and computational power which exceeds previous generations. Nowadays, graphics hardware is capable of executing increasingly costly and complex algorithms formerly only practicable with CPUs. Processing non-graphics tasks on GPUs 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 [NVI07b], AMD's CTM [PSG06], Brook [BFH*04] and Sh [MQP02] 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 (1.5GB for

NVIDIA QuadroFX GPUs and 2.0GB 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 sequentialization 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 connected via networks. 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 work load across cluster nodes. Our unified approach 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.

The remainder of this paper is organized as follows. Next, we give an overview over related work and GPU languages in particular. In Section 3 we present our system in detail and introduce the programming model and the compile processes. Providing parallelism across buses and networks is described in Section 4 and 5. Finally we analyze our system with different synthetic and real-world test cases and demonstrate that it provides nice scaling behavior.

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 [MQP02] and C for Graphics [FK03] 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 [BFH*04] extends C with simple data-parallel constructs to make use of the GPU as a streaming coprocessor. Glift [LSK*06] and Scan Primitives [SHZO07] focus on convenient data structures and facilitate the implementation of various algorithms on GPUs. Scout [MIA*07] goes one step further and even provides modules for scientific visualization techniques. The probably most commonly used high-level GPGPU language is NVIDIA's CUDA [NVI07b], 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 [PSG06] breaks ranks and provides a 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.

In this work, we focus on this 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. [SLS07]. Both directions benefit from ROCKS clusters and CUDA Roll [NVI07a]: A live boot system which easily and quickly sets up network clusters with CUDA support.

Languages for stream processing on GPUs profit by experiences from parallel programming with CPUs and network

clusters. This is a mature research area beyond the scope of this work and we refer the interested reader to Bal et al.'s comprehensive overview [BST89] and their comparison of parallel programming paradigms [Bal92].

3. System Overview

In this section we introduce the CUDASA programming environment and its programming model. Both are tightly coupled to the schematic overview in Fig. 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 Fig. 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 Fig. 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 (Fig. 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

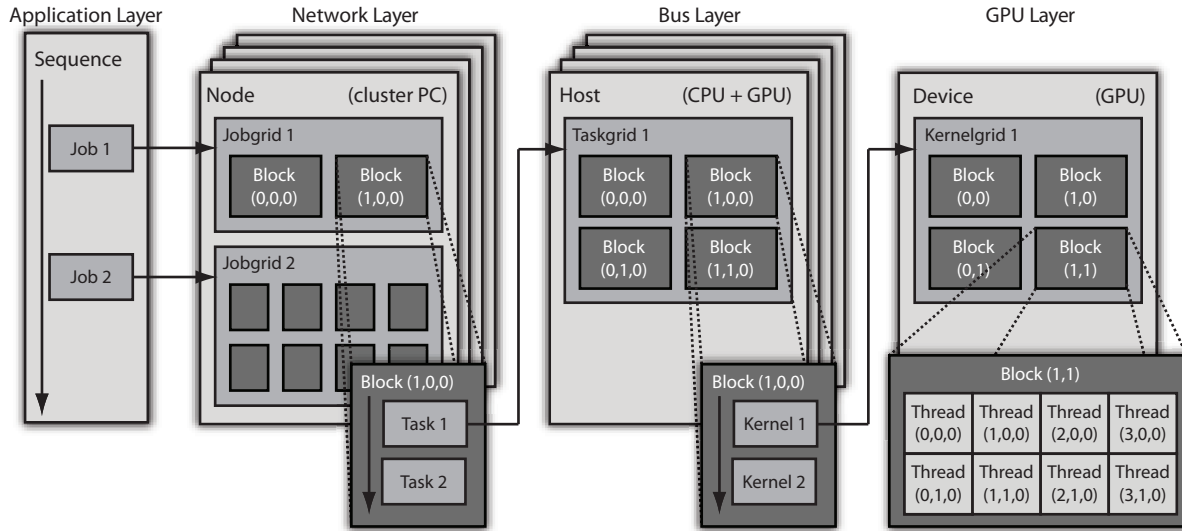


Figure 1: Schematic overview of all four abstraction layers of the CUDASA programming environment. The topmost layer is placed left, with decreasing level of abstraction from left to right.

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 *job grid* (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.

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*.

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

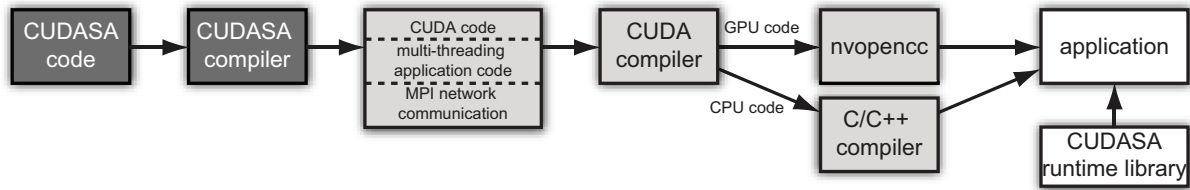


Figure 2: The CUDASA compiler processes a program and outputs standard CUDA code but also generates multi-threading and network code providing higher levels of parallelism. The CUDA compiler separates GPU and CPU code and hands it over to the corresponding compilers (nvopenc and any standard C/C++ compiler).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

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, right column), each accessible to functions of the corresponding and the underlying layers.

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 (Fig. 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 variables types and function scopes and can only be obtained from a full grammatical analysis. The code translation process is de-

scribed 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 [MN04], 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, 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

Abstraction	Exposed	Internal	Built-ins
application layer		sequence	
network layer	job	node	jobIdx, jobDim
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.

is particularly important to avoid the costly initialization of CUDA for every function call.

The polling mechanism achieves load balancing on a 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 (see Section 6 for a discussion), 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 to the standard CUDA compiler.

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. 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 computations. 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 to a node as it is the case with the Global Arrays paradigm [NHL94]. 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 when the mapping is closed.

CUDASA's shared memory manager is implemented using MPI Remote Memory Access (RMA). It distinguishes between two classes of operations: Collective and single-sided operations. 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, as 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 guarantees 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 never span across segment boundaries. This precondition can be easily enforced by CUDASA using a segment size that matches a multiple of a 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 throw up new questions on synchronization, distributed shared memory, and the compile process.

CUDA offers a synchronization of threads of 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 the storage of their complete state. Only in doing so can all blocks be executed until they reach the synchronization point. Due to limited on-board memory this would imply a high memory transfer overhead and is thus simply impracticable. Basically the same holds for higher levels of parallelism: In principle, it would be easy to provide a synchronization mechanism for blocks within a *taskgrid* (and analogously for *jobgrids*). However, 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.

The newly-introduced network layer shares the workload transparently across nodes of clusters. For this, our implementation of the distributed shared memory functionality partitions the memory pool evenly across all nodes of a cluster, i.e. each node makes the same amount of memory available to the memory pool. The order of executed blocks of a *jobgrid* is non-deterministic as idle nodes simply query for the next pending block. Preferably the query mechanism assigns a block which (mainly) requires a portion of the distributed shared memory already present within the node's system memory and thus minimizes costly memory transfers. In order to enforce such locality-aware assignments, we would like to implement a callback mechanism where blocks use the *execution configuration* to indicate which portion of distributed memory they intend to access. This concept is not anchored in CUDASA and remains for future work.

Scheduling on the network layer as described 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.

Finally, 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 [HSJ*06] is used with the necessary flags set accordingly.

7. Results

For bus parallelism we evaluated 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 practicability of our approach and further discuss communication overhead issues. All measurements were performed using CUDA version 1.1 for Linux (display driver version 169.04).

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 was used to guarantee optimal usage of each hardware. Namely, we used Intel’s Math Kernel Library 10.0 (MKL), AMD Core Math Library 4.0 (ACML), and NVIDIA’s CUBLAS Library 1.1.

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 multiplication example [NVI07b] 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 demonstrate excellent scaling behavior for both test cases, two 8800GTX Ultra (blue lines) and up to four 8800GTs (red 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 both 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.

As a second test case, we use CUDASA for implementing a part of Dachsbacher et al.’s [DSDD07] recently published method for interactive global illumination on GPUs. They use a finite-element radiosity method together with a reformulation of the rendering equation [Kaj86] which expresses light transport as a global energy transfer and a local operation that convolves incoming radiance with the surface BRDF. We distributed the computation of the “local pass”, which is the most demanding operation of the original algorithm, by populating the *taskgrid* with multiple blocks of surface elements. Table 2 shows the computation times in milliseconds for two typical scene sizes on our four-GPU system.

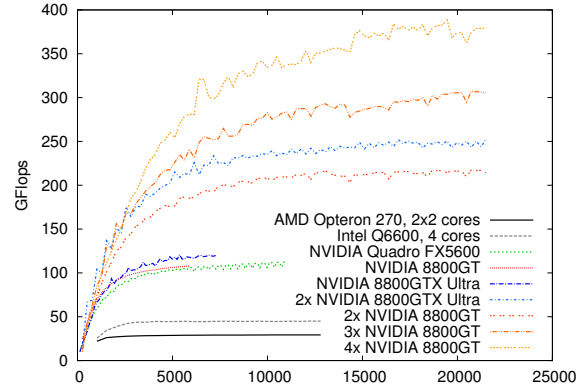


Figure 3: This figure illustrates the SGEMM scaling behavior with CUDASA for various system configurations on the bus layer. With SGEMM we achieve nearly optimal scaling with the number of GPUs, the slight performance fluctuations in multi-GPU configurations stem from the imperfect load balancing (see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

7.2. Network Parallelism

As the additional network layer adds a high communication overhead, the actual type of interconnect is an important factor. However, in the context of graphics clusters, which are the target platform of CUDASA, high-speed networks like InfiniBand or Myri-10G are commonly used. Running our tests for detailed timings on such systems remains for future work.

In the meantime, the proof-of-concept for the network layer is given by running SGEMM on two machines each equipped with two GPUs connected via a Gigabit Ethernet only. Compared to high-performance interconnects, this leads to an additional slowdown by approximately a factor of twenty. Therefore, the communication costs are expected to have major impact on the total system performance. We achieved 192 GFlops for 25000^2 matrix multiplication on all four GPUs. Detailed timings disclose that a *job* spends an average of only 1.6 s on computations, but 4.9 s on reading data from and writing to distributed shared memory.

We simulated an “optimal network” of four single-GPU nodes by using the full layer stack, but running all compute instances on a four-GPU machine ($4 \times 8800GT$). In this configuration, we achieve 236 GFlops for a 10240^2 matrix. When parallelizing this computation without the network layer, we get a performance of 314 GFlops on the same machine. By this we can estimate the overhead introduced by the network parallelization layer. It is mostly caused by the shared memory accesses, which take 1.5 times longer than the actual computation for this problem size – reading and writing distributed shared memory introduces an additional copy operation per *job*, which in turn also includes inter-process communication.

number of scene elements	1 GPU	2 GPUs	4 GPUs
32768	526	263	129
131072	1030	520	265

Table 2: This table illustrates the scaling behavior of the local pass, i.e. a convolution of a uniform directional radiance distribution (128 samples) with a glossy BRDF, as described in [DSDD07]. Timings are given in milliseconds for a single-node multi-GPU system with four NVIDIA 8800GTs.

8. Conclusions and Future Work

We introduced 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.

For future work we want to improve the load balancing on the network layer and implement a block assignment strategy which is aware of data-locality on nodes. Having the information of memory usage also available per block of a kernelgrid, would allow CUDASA to automatically utilize asynchronous data transfers to the GPUs and speed up the distributed memory accesses. Especially for large target sizes we expect further major performance benefits using this newly introduced feature of CUDA 1.1, as kernel block execution and data transfer can be parallelized. Furthermore, we plan to test the CUDASA environment on clusters with high-speed network interconnects, such as InfiniBand or Myri-10G. We also intend to make CUDASA publicly available and provide it for download on our webpage.

References

- [Bal92] BAL H. E.: A Comparative Study of Five Parallel Programming Languages. *Future Gener. Comput. Syst.* 8, 1-3 (1992), 121–135.
- [BFH*04] BUCK I., FOLEY T., HORN D., SUGERMAN J., FATAHALIAN K., HOUSTON M., HANRAHAN P.: Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.* 23, 3 (2004), 777–786.
- [BST89] BAL H. E., STEINER J. G., TANENBAUM A. S.: Programming Languages for Distributed Computing Systems. *ACM Comput. Surv.* 21, 3 (1989), 261–322.
- [DSDD07] DACHSBACHER C., STAMMINGER M., DRETTAKIS G., DURAND F.: Implicit Visibility and Antiradiance for Interactive Global Illumination. *ACM Transactions on Graphics (SIGGRAPH Conference Proceedings)* 26, 3 (August 2007).
- [FK03] FERNANDO R., KILGARD M. J.: *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Publishing Co., 2003.
- [HSJ*06] HUANG W., SANTHANARAMAN G., JIN H. W., GAO Q., X D. K. PANDA D. K.: Design of High Performance MVAICH2: MPI2 over InfiniBand. *ccgrid 00* (2006), 43–48.
- [Kaj86] KAJIYA J. T.: The Rendering Equation. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (1986), pp. 143–150.
- [LSK*06] LEFOHN A. E., SENGUPTA S., KNISS J., STRZODKA R., OWENS J. D.: Gflit: Generic, efficient, random-access GPU data structures. *ACM Trans. Graph.* 25, 1 (2006), 60–99.
- [MIA*07] MCCORMICK P., INMAN J., AHRENS J., MOHD-YUSOF J., ROTH G., CUMMINS S.: Scout: A Data-Parallel Programming Language for Graphics Processors. *Parallel Comput.* 33, 10-11 (2007), 648–662.
- [MN04] MCPPEAK S., NECULA G. C.: Elkhound: A Fast, Practical GLR Parser Generator. In *Proceedings of the Compiler Construction (CC'04)* (2004), pp. 73–88.
- [MQP02] MCCOOL M. D., QIN Z., POPA T. S.: Shader Metaprogramming. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2002), pp. 57–68.
- [NHL94] NIEPLOCHA J., HARRISON R. J., LITTLEFIELD R. J.: Global Arrays: A Portable Shared-Memory Programming Model for Distributed Memory Computers. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing* (1994), pp. 340–349.
- [NVI07a] NVIDIA: CUDA for Rocks Cluster User Guide. http://developer.nvidia.com/object/cuda_1_0.html, 2007.
- [NVI07b] NVIDIA: CUDA Programming Guide. <http://developer.nvidia.com/object/cuda.html>, 2007.
- [PSG06] PEERCY M., SEGAL M., GERSTMANN D.: A Performance-Oriented Data Parallel Virtual Machine for GPUs. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches* (2006), p. 184.
- [SHZ07] SENGUPTA S., HARRIS M., ZHANG Y., OWENS J. D.: Scan primitives for GPU computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (2007), pp. 97–106.
- [SLS07] SHEAFFER J. W., LUEBKE D. P., SKADRON K.: 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* (2007), pp. 55–64.