

SAARLAND UNIVERSITY
Faculty of Natural Sciences and Technology I
Department of Computer Science

BACHELOR THESIS

Parallel Processing on GPUs within Distributed Multimedia Middleware

submitted by

Martin Beyer

on May 28, 2009

Supervisor:

Prof. Dr.-Ing. Philipp Slusallek

Advisor:

Dipl.-Inf. Michael Repplinger

Reviewers:

Prof. Dr.-Ing. Philipp Slusallek

Prof. Dr. Raimund Seidel

STATEMENT

I assure that this work has been done solely by me without any further help from others except the listed literature in the bibliography.

Saarbrücken, May 28, 2009

Martin Beyer

DECLARATION OF CONSENT

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, May 28, 2009

Martin Beyer

Abstract

Using the GPU for non-graphics computations, known as *General Purpose Computation on Graphics Processing Units* (GPGPU), becomes increasingly more attractive since the floating point performance of recent GPUs greatly exceeds comparable CPUs. With CUDA (*Compute Unified Device Architecture*), Nvidia offers a powerful programming model to leverage the performance of their GPUs for general purpose computations.

CUDA enabled GPUs can be found in many desktop and even mobile devices and constitute a highly parallel co-processor that is well suited for multimedia processing. However, the number of graphics cards in a system is limited and CUDA does not provide abstractions to automatically use all GPUs in a system or even use remote GPUs of different hosts. Furthermore, integrating CUDA into existing applications can be time consuming and is accompanied by restrictions on scheduling and different memory spaces. According to this, many applications do not provide reusable components that can be integrated or combined efficiently in other applications.

The goal of this thesis is to build a framework to integrate CUDA into a distributed multimedia middleware that offers support for distributed processing. This allows to distribute processing tasks to GPUs located on remote hosts. Furthermore, the problem of different memory spaces is also an issue solved by a distributed middleware. The general idea is to map concepts of a distributed middleware to a co-processor like the GPU and to treat it similar to a distributed system.

In particular, a separation of data transfer and data processing allows for an efficient combination of modular components that implement algorithms for the GPU. Additionally, collocated CPUs and GPUs can be used concurrently for computationally demanding tasks. By efficient use of the features provided by CUDA, performance measurements will show that using a distributed middleware even for locally running applications does not necessarily induce an overhead.

Acknowledgement

First, I would like to thank Prof. Dr.-Ing. Philipp Slusallek for allowing this bachelor thesis. Moreover, I thank my advisor Michael Replinger for the excellent supervision and his support in any problem I had. I thank the whole NMM team and all the people at the Computer Graphics Lab for a kind working atmosphere and their competent support. I also have to thank Prof. Dr. Raimund Seidel for accepting my request to be my second reviewer. Over and above, I thank my friends and my family for their continuous support.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Goals	2
1.3	Outline	2
2	Parallel Computing	5
2.1	Parallel Architectures	5
2.2	Parallel Memory Architectures	6
2.3	Parallel Programming Models	7
2.4	Introduction to GPGPU	8
2.5	Summary	9
3	CUDA	11
3.1	Tesla Architecture	12
3.2	CUDA Programming Model	13
3.2.1	Thread Hierarchy	15
3.2.2	Memory Hierarchy	17
3.2.2.1	Off-Chip Memory	17
3.2.2.2	On-Chip Memory	18
3.2.3	Synchronization	18
3.3	Features	19
3.3.1	Asynchronous Operations	19
3.3.2	Streams	19
3.3.3	Events	20
3.3.4	Contexts	21
3.3.5	Page-locked CPU Memory	22
3.3.6	Multiple GPUs	23
3.4	Summary	23
4	Related Work	25
4.1	Distributed Middleware Solutions using CUDA	25
4.2	GPU Programming Frameworks	26
4.3	GPU-Accelerated Multimedia Libraries and Applications	27
4.4	Summary	28

5	NMM - Network-Integrated Multimedia Middleware	29
5.1	Overview	29
5.2	Distributed Flow Graph	30
5.3	Nodes	30
5.4	Formats	31
5.5	Messages	31
5.5.1	Buffers and Buffer Managers	31
5.5.2	CEvents	32
5.6	Transport Strategies	32
5.7	States	33
5.8	NMM Services and Applications	34
6	Design of a CUDA-NMM Framework	35
6.1	CUDA Streams	36
6.1.1	Approach 0: One global Stream	36
6.1.2	Approach 1: One Stream per NMM Buffer	36
6.1.3	Approach 2: One Stream per NMM Node	37
6.1.4	Performance Comparison	38
6.2	Key Design Decisions	39
6.3	Three Layer Approach	40
6.3.1	Flow Graph	40
6.3.2	Memory Management	41
6.3.3	Scheduling	41
6.4	Scheduling Strategies	41
6.4.1	Context Management	42
6.4.1.1	Stateful and Stateless Nodes	42
6.4.2	Stream Management	43
7	CUDA-NMM Framework	45
7.1	Buffer Processing using CPU and GPU Nodes	45
7.2	Hide different Memory Spaces	46
7.3	GPU Nodes	46
7.4	Scheduling CUDA Worker Threads	48
7.5	Data Transfers between Nodes	49
7.6	Asynchronous CUDA Operations	50
7.7	Minimize Overhead of Memory Transfers	50
7.8	Multiple GPUs within a System	50
7.9	Use GPUs of Remote Systems	51
7.10	Shareable CudaBuffers	51
7.11	Handling Instream Events	51
8	CUDA Plugins for NMM	53
8.1	Implementing GPU Nodes	53
8.1.1	Node Development	53
8.1.1.1	Initialization	53
8.1.1.2	Buffer Processing	54
8.1.1.3	Instream CEvents	56
8.1.2	Kernel Development	56
8.2	Application Development	56
8.3	Implemented GPU Nodes	57

8.3.1	IdNodeCuda	57
8.3.2	DenoiseNodeCuda	57
8.3.3	OverlayNodeCuda	57
8.3.4	SimpleRayTracer	58
9	Performance Measurement	59
9.1	Benchmark Setup	59
9.2	Test Cases	59
9.3	Performance Measurement	60
10	Results	63
10.1	Summary	63
10.2	Achieved Goals	65
10.3	Future Work	66
10.3.1	Additional Memory Spaces	66
10.3.2	OpenGL Integration	66
10.3.3	Stateful GPU Nodes	66
10.3.4	Extensions to Transport Mechanism	67
10.3.5	Pipelined Parallel Binding	67
	List of Figures	69
	List of Tables	71
	Bibliography	73

Chapter 1

Introduction

1.1 Motivation

Multimedia processing, such as de- and encoding of high-definition video material, is a compute intensive and time consuming task. In the past few years, CPUs commence to stagnate on speed for serial code execution, resulting in multi-core CPUs even in common desktop PCs. This indicates the trend towards increasing parallelization to meet the demands of compute intensive applications. However, upcoming many-core technologies such as the IBM Cell Broadband Engine [30], Intel's Larrabee [52] or recent GPUs are specialized architectures that offer very high floating point performance at rather low costs. They are ideally suited for highly parallel computations and provide a floating point performance that greatly exceeds up-to-date desktop CPUs.

Especially GPUs have made fundamental improvements and today they are full-fledged many-core processors that are freely programmable. Using the GPU for non-graphics computations is usually summarized by the term GPGPU (*General Purpose Computation on GPUs*). Since GPUs are designed for processing large data streams they are well suited to be used in multimedia applications. For example image or video processing is a natural fit for data-parallel processing on the GPU [56].

With CUDA, Nvidia proposes a very flexible programming model that enables to write algorithms for the GPU directly in the C programming language. CUDA uses a stream processing approach, where a series of highly parallel functions, called *kernels*, is executed on the GPU to process different data items in parallel. But GPUs only attain their performance on highly parallel computations which requires a reasonable fast CPU for the application logic.

However, since GPUs operate in a different memory space than the CPU and CUDA places special requirements on scheduling, the integration of CUDA algorithms into existing applications can be quite time consuming. Furthermore, the number of GPUs in a system is very limited which motivates the usage of all GPUs in a networked environment for distributed computing. But CUDA does not provide sufficient abstractions to use multiple GPUs in a single system or even use graphics cards of remote systems.

The concept of a distributed multimedia middleware with a *flow graph* that encapsulates processing steps in modular *nodes* allows applications to specify

processing on different PCs. Data that traverses the flow graph is processed by the nodes while data transport is handled in the *edges* connecting the nodes. Integrating CUDA support in such a middleware allows to use GPUs of remote systems, but also to distribute work to the GPUs within a single machine. Actually, the model of a flow graph perfectly fits to the stream processing model of the GPU: Each node implements kernel functions to process the data that passes through the flow graph. A framework built on top of such a middleware can hide CUDA specific requirements from the application. This dramatically simplifies the integration of CUDA algorithms into existing applications. Furthermore, problems such as different memory spaces are also solved by a distributed middleware. The general idea of this thesis is to treat the GPU similar to a distributed system in order to increase modularity and reusability of algorithms running on the GPU. Separating data processing on the GPU from data transfers between CPU and GPU allows to efficiently combine processing steps on the GPU in collaboration with the CPU.

1.2 Goals

The overall goal of my thesis is to integrate CUDA into the *Network-Integrated Multimedia Middleware* (NMM). This middleware focuses on multimedia processing using a distributed flow graph that specifies processing steps within nodes running locally or on a remote host. A framework has been developed that allows to integrate new nodes in NMM to process data on the GPU. To achieve this, the following goals are defined:

1. The CUDA programming model, in particular its features and requirements have to be analyzed in order to allow for an efficient integration in NMM.
2. Existing CUDA kernels provided by multimedia libraries can be integrated and reused within the presented framework.
3. A suitable design of the framework has to be developed in order to apply the features of CUDA to NMM. In particular, CUDA specific requirements have to be abstracted from the application to integrate CUDA algorithms seamlessly into existing applications.
4. The framework has to allow the encapsulation of CUDA kernels into the nodes of a flow graph, which can be efficiently combined for multimedia processing using both the GPU and the CPU. This requires a separation of data transport and data processing.
5. All GPUs within a machine should be automatically used for processing and applications can specify processing on GPUs of remote hosts.
6. Benchmarks are necessary to measure the overhead of using a middleware compared to a plain CUDA program.

1.3 Outline

Chapter 2 starts with the fundamentals of parallel programming and an introduction to GPGPU programming. In chapter 3, I will introduce the Nvidia

Tesla GPU architecture and the CUDA programming language. Additionally, features of CUDA are presented that are used for an efficient integration of CUDA in NMM. Related work is presented in chapter 4. This chapter contains an overview on GPGPU programming techniques and CUDA enabled multimedia libraries as well as a short overview on CUDA-enabled frameworks for distributed computing. Chapter 5 highlights the most important aspects of the Network-Integrated Multimedia Middleware which is used as a basis to integrate CUDA. Subsequently, chapter 6 describes the overall design of the framework and the considerations that lead to that particular design. Chapter 7 then describes the implementation in more detail. The general idea of this framework is already described in [47]. Plugin development using NMM and the CUDA framework is described in chapter 8. Furthermore, the nodes that have been implemented in the scope of this thesis are presented. Chapter 9 presents performance measurements showing that multimedia applications using CUDA can even improve their performance when using the presented approach. Finally, chapter 10 summarizes the results of this work and highlights future work and possible extensions.

Chapter 2

Parallel Computing

Traditionally software has been written for serial execution where a problem is solved by a discrete series of instructions. Today we face the fact that building ever faster serial processors is infeasible for physical and practical reasons. Limits in transmission speeds between hardware components and necessary miniaturization to allow for a higher transistor density impose significant hardware constraints. Making a single processor faster is increasingly expensive with respect to the achieved performance and is accompanied by thermal problems [19]. In fact, many algorithms can be expressed in terms of parallel computations which can be solved more efficiently using many processor cores with moderate performance.

As a consequence, computer architectures are increasingly relying upon hardware level parallelism to improve performance. Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem. A computational problem is split up into discrete parts that can be solved concurrently by multiple processors [19]. Typical desktop CPUs are equipped with up to four cores that employ multiple floating point ALUs (Arithmetic Logic Unit) used for SIMD processing. Current GPUs push these counts to an extreme scale: High-end graphics cards provide up to 30 processors and a total of 240 floating point ALUs (Nvidia GeForce GTX 280).

This chapter gives an overview of parallel architectures and programming models for multi- and many-core processors. An introduction to parallel computing using commodity graphics cards can be found at the end of this chapter.

2.1 Parallel Architectures

The most widely used classification of parallel architectures is Flynn's taxonomy [24]. It classifies parallel computers into 4 groups, according to the two independent dimensions of instruction stream and data stream, which can have the values single or multiple:

- **Single Instruction, Single Data (SISD):**
a serial (non-parallel) computer
- **Single Instruction, Multiple Data (SIMD):**
each processor executes the same instructions on different data items

- **Multiple Instruction, Single Data (MISD):**
each processor executes different instructions on a single data item
- **Multiple Instruction, Multiple Data (MIMD):**
each processor can execute different instructions on different data streams

Multi-core CPUs can be classified as MIMD architectures since each processor core can execute completely different instructions on different data items [19]. Each CPU core can further employ SIMD subcomponents such as SSE (*Streaming SIMD Extensions*) to process multiple data items by the same instructions. SIMD can be subdivided into *Vector Processing* which exposes the vector width to the software (e.g., SSE) and the *SIMT (Single Instruction, Multiple Threads)* class. In the SIMT model, the hardware vector width is abstracted from the software by spawning a thread for every data item where every thread executes the same code. This allows programmers to write parallel code on thread level to process independent data items as well as to coordinate work between multiple threads [37].

2.2 Parallel Memory Architectures

Multi-core systems need access to some memory space to read and write data and to coordinate work of different processor cores. This memory can either be globally accessible and shared across all processor cores or local to each processor core. Thus, multi-core systems are further categorized according to the memory layout and the access pattern of its processor cores [19].

In **shared memory** architectures all processor cores have access to a global memory region. The processor cores operate independently but share the same memory resources. Write operations to the global memory are visible to all other processors.

Shared memory architectures provide user-friendly memory access and allow to cooperate work between different processor cores through global memory. On the other side, the programmer is responsible for synchronizations to avoid race conditions when accessing global memory. Since all processor cores have to share the bandwidth to the global memory space, the connection between the processors and the memory can easily become the bottleneck.

In contrast to shared memory architectures, the processor cores in a **distributed memory** architecture have their own local memory space and there is no concept of a global memory space. To exchange data between different processors a communication network is required. Also, it may be difficult to map existing data structures, based on global memory, to a distributed memory organization [19]. The main advantage of a distributed memory architecture is the scalability to an increasing core count.

There also exist **hybrid distributed-shared memory** architectures where multiprocessors with access to a local memory space are interconnected by a network with other multiprocessors. Additionally, the main memory is accessible by all processor cores. One example for a hybrid distributed-shared memory architecture is the *Cell Broadband Engine* [51].

2.3 Parallel Programming Models

Data-parallel [5] programming models are designed to process independent pieces of data on multiple processing cores. Listing 2.1 illustrates data-parallel processing on an array `d` where a function `foo` is applied to every data item. If `CPU_A` and `CPU_B` execute the same code, `CPU_A` operates on the first and `CPU_B` on the second half of the array. Certainly, this concept can be generalized to any number of processors. Typically, data-parallelism is achieved by using a SIMD approach either in the form of vector processing or by independent threads executing the same instructions (SIMT). As independent data items can be distributed to different processor cores it usually scales well with the problem size. Data-parallel computations can be realized using for example OpenMP [54].

Listing 2.1: Data-parallel processing. [5]

```

1  if CPU = "CPU_A"
2      lower_limit := 1
3      upper_limit := round(d.length/2)
4  else if CPU = "CPU_B"
5      lower_limit := round(d.length/2) + 1
6      upper_limit := d.length
7
8  for i from lower_limit to upper_limit by 1
9      foo(d[i])

```

Task-parallel [7] programming models can execute completely independent operations. Each thread follows its own path of execution and can be distributed to different processor cores. Listing 2.2 illustrates task parallelism. According to its name, each CPU is assigned another task which can be completely independent from each other. Examples are PThreads or Java which provide a fork-join parallelism where each thread has to be managed explicitly. These models are not particular convenient for data-parallel operations [37].

Listing 2.2: Task-parallel processing. [7]

```

1  if CPU = "CPU_A"
2      do task "A"
3  else if CPU = "CPU_B"
4      do task "B"

```

Data parallelism requires that the same function is repeatedly applied to different data items whereas task parallelism provides greater flexibility. However, task parallelism is more complex since its up to the developer to define the tasks executed by different threads [29].

In the *stream processing* [6] model a set of data items (a *stream*) is processed by a series of operations (*kernel functions*). Parallelism and data dependencies are exposed in the application by structuring data items into streams that are processed by a pipeline of kernel functions [44]. Typically, the same kernel function is applied to every data item of a stream while operations on a single stream entity are independent from the others. Thus, the kernel stages can easily be parallelized and distributed to different processor cores using a SIMD approach. Task parallelism is available across different pipeline stages.

By structuring data into streams many aspects of parallel execution are abstracted from the developer. Kernels define processing of independent data units which allows the hardware to efficiently manage resources for parallel

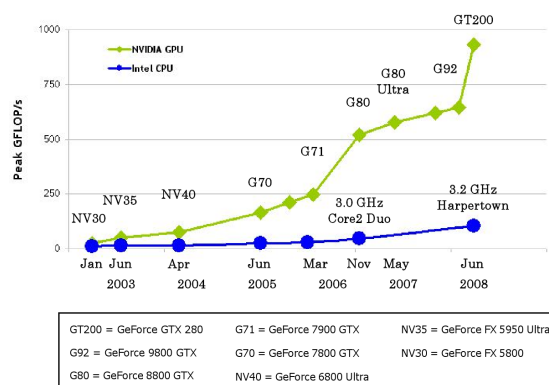


Figure 2.1: Comparison of floating point performance of CPU and GPU. [38]

execution [6]. This simplifies the programming model, but also restricts its generality. Data is expected to be streamed through processing units without the necessity of a communication between different threads.

2.4 Introduction to GPGPU

This section gives a short overview on general purpose computing on a GPU (GPGPU), more detailed information on the graphics pipeline and the historic development of GPGPU can be found in [23] and [44].

The most striking advantage of up-to-date GPUs compared to their CPU counterparts is the high floating point performance and its rapid increase, as can be seen in figure 2.1. The reasons for this are fundamental architectural differences: CPUs are optimized for serial code execution and lots of techniques have been developed to speed this up. Today many of the transistors of a standard desktop CPU are dedicated to non-computational tasks like branch prediction and caching. Whereas the highly parallel nature of graphics computations enables GPUs to use more transistors for computation, achieving higher arithmetic intensity with the same transistor count [44].

In general, GPUs are designed in favor of high throughput on many parallel operations instead of a low latency of execution [23]. Graphics computations on a GPU such as rendering tasks are specified through APIs like Direct3D or OpenGL. These APIs represent rendering by a stream processing model where a stream of vertices and fragments flows through several stages of a graphics pipeline. Data-dependencies within a programmable pipeline stage are in general not allowed and different pipeline stages only communicate via FIFO (First In, First Out) streams [37] which seriously restricts the generality of the stream programming model. Early generations of GPUs only provided fixed-function pipelines, but every new hardware generation added additional programmable processing units. Today, most of the processing resources of a GPU exist within programmable processors, only some fixed function stages remained that perform difficult to parallelize work and resolve data dependencies [23].

Shader languages [32] have been developed to write shader programs (or kernels in stream processing terminology) that are executed in the programmable

stages of the pipeline. A shader function is merely a C like function that computes one output from one input entity. A function invocation is abstracted as an independent sequence of control that executes in complete isolation [23]. Consequently, a shader function can be invoked many times in parallel to operate on different stream entities. The graphics API does not provide concepts to explicitly manage parallel execution in threads but rather manages thread distribution to available processing units in hardware. This *hardware multithreading* together with SIMD processing is ideally suited for the highly parallel task of rendering computations and greatly simplifies the parallel execution. [23]

But the restricted stream processing model through the graphics pipeline is not very well suited for general purpose computations. However, early GPGPU approaches used shader languages to perform non-graphics computations within a graphics pipeline. A catalog of the current and historical use of GPUs for general-purpose computations can be found in [14]. A major improvement to the GPGPU programming convenience was the introduction of the Nvidia CUDA programming model and the Tesla architecture of recent Nvidia GPUs. Chapter 3 presents an introduction to the CUDA programming model and highlights its advantages over general purpose computations using shader languages.

Although GPGPU developers came up with a high variety of applications that benefit from the GPU's arithmetic power, there are applications like word processing which are dominated by memory communication and difficult to parallelize, and thus not suited for the GPU [44]. GPUs are best suited for data-parallel algorithms with high arithmetic intensity. Non-data-parallel algorithms perform poorly on the GPU [23] and have to be processed by the CPU. Since many algorithms in the field of multimedia processing can be expressed as data-parallel operations, they are well suited for the GPU.

GPUs have their own memory space and cannot access the CPU's memory. This requires data transfers between CPU and GPU which is of course accompanied by an overhead. Thus, only computational problems with a high ratio of computations compared to data communication are candidates for GPU processing.

2.5 Summary

GPUs are specialized processing units which are suitable for algorithms that can be expressed in a form of stream processing. In particular, data-parallel operations are candidates for stream processing. Unlike the *Cell Broadband Engine*, GPUs do not include a generalized controller core that is able to perform the application logic and explicitly manages processing on the GPU, this task is up to the CPU. Furthermore, GPUs are not designed for a low latency execution of a single task, but rather in favor of high throughput when processing large data sets in parallel.

Chapter 3

CUDA

CUDA (*Compute Unified Device Architecture*) [38] is Nvidia's most recent programming model for general purpose computations on GPUs supporting the *Tesla Unified Graphics and Computing Architecture* [41]. CUDA aims to preserve the performance of the GPU while increasing the generality of the programming model. With CUDA, the GPU is represented as an array of processor cores and computations are abstracted as large batch operations that involve many function invocations in parallel [23]. Typically a serial CPU program copies data to the GPU and invokes a parallel function, called *kernel*, to process data.

CUDA allows to program the GPU directly in the C programming language without a graphics API in between. The programming model is strongly related to the stream processing model (Section 2.3): A series of kernel functions is launched on a set of data items. A kernel invocation spawns a huge number of concurrent threads that execute the same kernel function on each data item (SIMT, section 2.1). However, CUDA greatly extends the capabilities of the stream processing model by providing flexible thread creation and explicit synchronization to cooperate work between threads within a hierarchy of shared memory. This is not possible in typical stream programs that do not allow dependencies among kernel threads [37].

CUDA treats the GPU as a highly parallel co-processor that solves compute intensive algorithms much faster than the CPU is capable. But the GPU can only achieve its performance when the algorithm can be decomposed into many independent subtasks that can be executed in parallel. Although CUDA increases the flexibility to program the GPU, the hardware is only suited for data-parallel operations. Thus, the GPU cannot replace the CPU in a sense of a general purpose main processor that performs the application logic.

In CUDA the CPU is usually called *host* and the GPU is called *device*. For the scope of this thesis I will keep up with the CPU and GPU terminology in order to avoid confusions with *host* denoting a networked computer system.

To develop CUDA programs, a CUDA enabled graphics adapter with corresponding driver, the CUDA Toolkit and the CUDA SDK are required that can be downloaded from the *CUDA Zone* at the Nvidia website [42]. The Toolkit contains the `nvcc` compiler that is used to generate binaries that are executable on the GPU. CUDA provides two APIs that have to be used by an application running on the CPU to access the GPU's compute resources: A low-level *Driver*

API and a *Runtime API* that is built on top of the Driver API. Programs using the Runtime API can be compiled to a single executable that contains the CPU as well as the GPU code. Programs using the Driver API need to separate CPU and GPU code which are compiled a separate binary for the CPU and a *kernel module* for the GPU. The Runtime API is easier to program but lacks some features that are only available through the Driver API. These APIs are mutually exclusive, so applications can only use one of them. In general, the Driver API should be preferred when a high level of control is required. The CUDA SDK is optional, but since it provides many libraries, examples and utilities for developers, it is a very useful extension to the CUDA Toolkit.

The following section gives an overview on the hardware architecture of recent Nvidia GPUs. Section 3.2 introduces the CUDA programming model and section 3.3 presents features of CUDA concerning CPU-GPU interaction.

3.1 Tesla Architecture

Since the introduction of the GeForce 8 series in 2006, Nvidia calls the underlying hardware architecture of their GPUs the *Tesla Unified Graphics and Computing Architecture*. This architecture significantly extends the capabilities of the GPU beyond traditional graphics computations. Tesla GPUs basically consist of a scalable array of highly multithreaded processor cores that can equally be used for graphics as well as general-purpose computations. A unified processor model superseded the distinction between specialized fragment and vertex processors with limited capabilities [37].

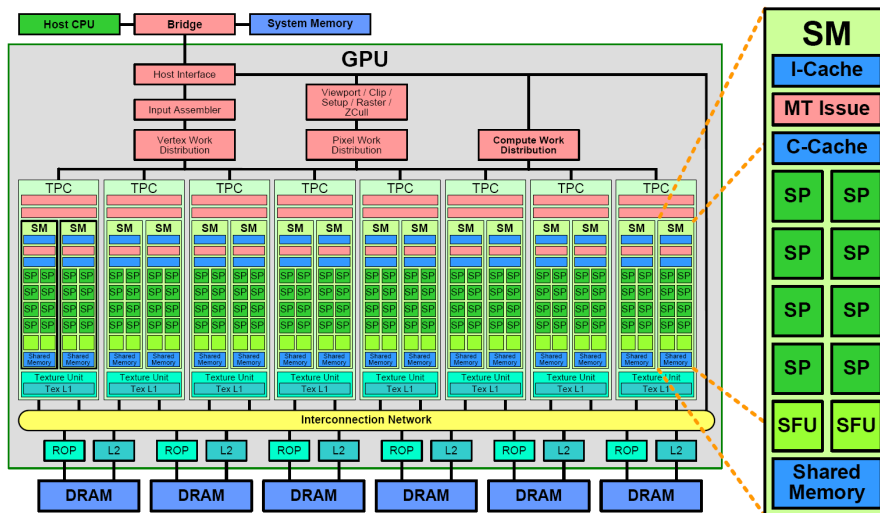


Figure 3.1: Tesla GPU Architecture: Array of arbitrary many streaming multiprocessors (SM), each consisting of 8 scalar processors (SP). A global memory region (DRAM) is accessible by all processor cores whereas the on-chip shared memory is only accessible within an SM. [50]

GPUs of the GT200 series (e.g., GeForce GTX 280) contain up to 30 streaming multiprocessors (SM), each consisting of 8 scalar processors (SP) - a total of

Memory	Location	Size	Hit Latency	Read-only
Global DRAM	off-chip	up to 1.5 GB	200-300 cycles	no
Shared	on-chip	16 KB per SM	\simeq register latency	no
Constant (C-Cache)	on-chip cache	64 KB total	\simeq register latency	yes
Texture (Tex L1)	on-chip cache	up to global	> 100 cycles	yes

Table 3.1: The different memory spaces of Tesla GPUs and their properties. [49]

240 SPs. Each SM is equipped with a multithreaded instruction unit (MTIU), 16 KB of on-chip shared memory, 2 special function units (SFU) and 16384 registers, illustrated in figure 3.1. A multiprocessor is capable of managing up to 1024 concurrent threads and distributes the threads to its SPs. A detailed list of the hardware capabilities for each graphics card can be found in the CUDA Programming Guide [38] (Appendix A).

Thread creation and management to map the threads to the physical processor cores is entirely done in hardware by the MTIU resulting in almost zero scheduling overhead. While CPUs use large caches to hide latencies to external DRAM, Tesla GPUs use a large number of threads combined with lightweight thread switching if they are stalled.

CPU and GPU cannot share memory spaces, so data has to be explicitly copied to the global memory region of the graphics card before it can be processed by kernel functions. Besides the external DRAM of the graphics card, which is globally read and writeable by all SPs, there exist several other memory regions to reduce the bandwidth usage for accesses to the global memory. These are summarized in table 3.1. Data is always copied to the global DRAM before it can be processed by the GPU. The global memory region is also the backing store for constant and texture memories. Shared memory is located on-chip, that is inside an SM, and used as scratchpad memory. The constant and texture memory regions originally reside in global memory, but accesses to these memories are cached by an on-chip cache.

3.2 CUDA Programming Model

CUDA employs a heterogeneous programming model, where a serial application calls parallel kernels on the GPU for data-parallel operations (Figure 3.2). The kernel functions executed on the GPU are written in the C programming language while CUDA provides some extensions concerning memory and thread management. A kernel specifies the execution and branching behavior of a single thread and is invoked on a huge number of threads to process different data items in parallel.

Listing 3.1 shows a simple C function and its counterpart in CUDA (Listing 3.2) that computes the result of $y \leftarrow \alpha x + y$, where x and y are arrays of equal size and α is a scalar value. This function is the so-called `saxpy` kernel defined in the BLAS (*Basic Linear Algebra Subprograms*) library of the CUDA SDK. The example shows a very common and mechanically applicable pattern for data-parallel computing: Since each iteration of the `for` loop in the C version is independent from the others it can easily be split into N independent operations - each loop iteration becomes an independent CUDA thread. In this example each thread computes just a single value of the output array. [37, 38]

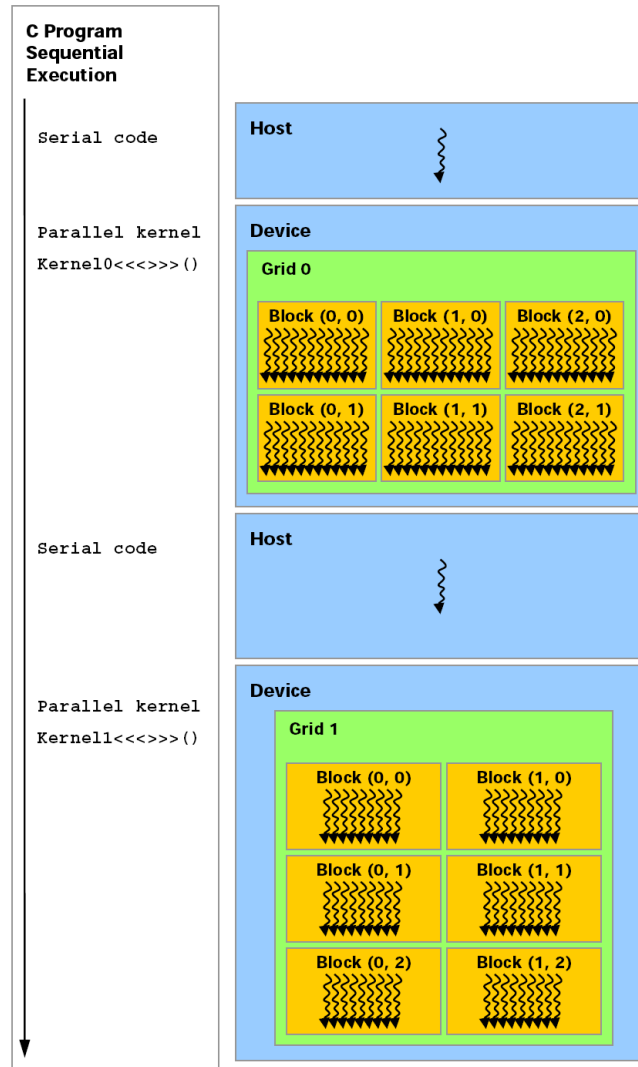


Figure 3.2: The heterogeneous programming model of CUDA: A serial application invokes parallel kernel functions on the GPU. [38]

Listing 3.1: C function: $y \leftarrow \alpha x + y$

```

1 void saxpy_serial(int N, float alpha, float* x, float* y)
2 {
3     for (int i = 0; i < N; ++i)
4         y[i] = alpha*x[i] + y[i];
5 }
6
7 // invocation
8 saxpy_serial(N, 2.0, x, y);

```

Listing 3.2: CUDA kernel: $y \leftarrow \alpha x + y$

```

1 __global__
2 void saxpy_parallel(int N, float alpha, float* x, float* y)
3 {
4     // compute thread ID based on build-in variables
5     int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
6
7     if (thread_id < N)
8         y[thread_id] = alpha*x[thread_id] + y[thread_id];
9 }
10
11 // invocation by CPU application
12 int nblocks = (N + 255) / 256;
13 saxpy_parallel<<<nblocks, 256>>>(N, 2.0, x, y);

```

The keyword `__global__` (line 1 in listing 3.2) specifies the function as a kernel entry point that can be invoked by the CPU. Since this kernel function is executed many times in parallel, the `thread_id` is computed (line 5) to determine the tasks for a particular thread. In this case, the `thread_id` is used to select the elements from the input array and write the results to the output array (line 8). Note that each thread only computes a single value from the output array. The degree of parallelism is explicitly specified by the application depending on the number of data items (line 12) before the kernel function is executed (line 13).

One limitation of CUDA is that there is no program stack, so it does not support recursive kernel functions. It is not suggested to use C++ code in kernel functions, although this partly works. For example in some cases it is possible to use templated versions of a kernel, but most parts of the C++ STL cannot be used in kernel code because they need a stack.

3.2.1 Thread Hierarchy

Nvidia offers CUDA enabled GPUs spanning a wide market range from inexpensive mainstream GPUs to high-end graphics cards. Although the basic hardware architecture is identical, the number of multiprocessors varies between different models. This requires CUDA algorithms to scale to an arbitrary number of processor cores. In CUDA's SIMT model, the threads are grouped to *thread blocks* of up to 512 threads. Threads within a thread block are guaranteed to be executed on the same SM and can cooperate through barrier synchronization and on-chip shared memory. The thread blocks, in turn, are further grouped to build a single *grid*. Launching a grid corresponds to a kernel invocation where the same function is executed by a specified number of threads.

The size of the thread blocks and the grid are specified by vectors with up to three dimensions. Figure 3.3 shows two-dimensional thread blocks within

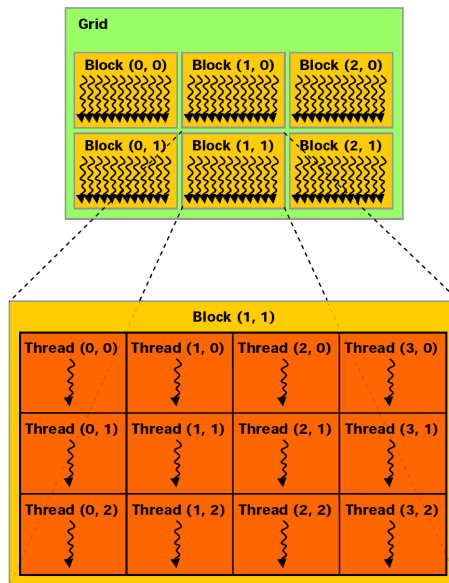


Figure 3.3: CUDA thread hierarchy: Threads are grouped in thread blocks running on the same multiprocessor. These thread blocks are arranged in a grid containing all threads that execute the same kernel function. [38]

a two-dimensional grid. The degree of parallelism is explicitly specified when launching the kernel using an extended function call syntax:

Listing 3.3: CUDA kernel invocation

```
1 kernel<<<dimGrid,dimBlock>>>( ... parameter list ... )
```

Kernel functions can compute their thread ID within the grid based on the built-in variables `blockIdx`, `blockDim` and `threadIdx`. Usually the kernel function uses the thread ID to specify the data item a thread should process.

Only the threads within a thread block can work cooperatively by accessing the same shared memory and by explicit synchronization. Dependencies between different thread blocks are not allowed, as they are executed in any order and automatically distributed to different SMs. The number of thread blocks, however, can greatly exceed the number of physically available multiprocessors. This makes CUDA kernels scalable to an increasing number of multiprocessors.

To manage the large thread population, the Tesla GPUs execute threads in groups of 32 called *warps*. The warps are the scheduling units of the MTIU and can be considered as the SIMD width of the Tesla architecture. The distribution of threads within a warp to the 8 available SPs is done in hardware. The threads within a warp always execute the same instruction, but they are free to perform data dependent branches. If a warp diverges, the branches are processed one after another while the other branches are idle. Divergence can be safely ignored when designing for correctness, to achieve peak performance kernels should avoid divergence within a warp [37]. Warps are swapped out if the threads are stalled waiting for memory access, so developers have to specify threads at a very fine granularity in order to provide enough of them to keep resources busy.

3.2.2 Memory Hierarchy

Specialized pixel and vertex processors of previous GPU generations and their APIs supported only restricted memory access patterns such as sequential writes to a linear array [49]. The unified processor model of Tesla GPUs enables each processor core to perform accesses to arbitrary memory blocks, with some restrictions concerning performance considerations.

Figure 3.4 presents an overview of the different memory spaces that are available in CUDA. The off-chip or global memory can be used for communication between CPU and GPU whereas the on-chip memories are only accessible by the GPU's processor cores. Guidelines on how to make efficient use of the different kinds of memory can be found in [49] and in chapter 5 of [38]. Memory blocks in off-chip memory can only be allocated by the CPU application, that means, input and output memory have to be known before the kernel function is executed.

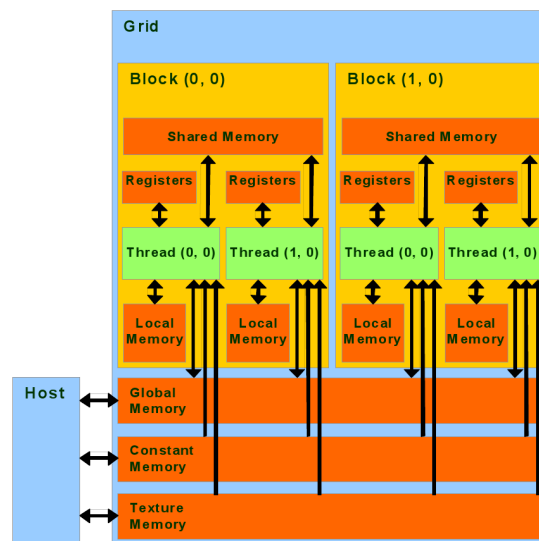


Figure 3.4: The memory hierarchy of Tesla GPUs. CPU and GPU have to exchange data via GPU memory. The CPU can allocate data in global memory as linear memory or as CUDA arrays. The global memory region is also the backing store for constant and texture memory. [26]

3.2.2.1 Off-Chip Memory

The global GPU memory is the only memory accessible by the CPU and the GPU. The application has to allocate the memory for the input and output data and passes pointers to these as arguments to the kernel function. Memory blocks can be allocated as *linear memory* and as one- or multi-dimensional *CUDA arrays*. These are read and writable by the application through memory copies, but only linear memory is writable by the kernel.

If a warp of threads accesses consecutive blocks of linear memory this can be coalesced and processed at once whereas accesses to scattered memory blocks

have to be serialized. Depending on the hardware generation - the so-called *compute capability* - there exist several access patterns that can be coalesced. These patterns are listed in the CUDA Programming Guide [38] (section 5.1.2).

The application can bind any region of global memory to *textures* that are read-only for the kernel. Read accesses to textures are cached and optimized for access patterns with a high spatial locality that cannot be coalesced. CUDA arrays are opaque memory layouts that are optimized for texture fetches. In contrast to linear memory which is specified by its size, CUDA arrays require a more complex *description* to be allocated. The same holds true for textures which require a *texture reference* to perform texture fetches. Texture fetches from CUDA arrays can use the texturing hardware of the GPU to perform interpolation between neighboring texture elements (*texels*). The CUDA array descriptors as well as the texture references have to be created and maintained by the CPU application.

Finally, the application can specify a small fraction of global memory (64 KB) as constant memory which is read-only for the kernel, but writeable by the CPU. Read accesses to constant memory are cached with 8 KB single ported cache per SM.

In general, memory copies between CPU and GPU are rather expensive and should be minimized if possible.

3.2.2.2 On-Chip Memory

During execution the threads within the same thread block can access the same on-chip shared memory with very low latency. This shared memory can be used as a cache for data originally residing in global memory and as a scratchpad memory for intermediate results. Shared memory is not accessible by the CPU, so results have to be copied to global memory before the kernel terminates. To place a variable in shared memory, the `__shared__` variable type qualifier is used in kernel code. Variables private to a thread are placed in registers. Registers are a scarce resource since they are partitioned among all threads executing on a SM.

3.2.3 Synchronization

Essentially there are two types of synchronization: Between CPU and GPU or synchronizations among the CUDA threads within the same thread block.

Synchronization between CPU and GPU is necessary to ensure the GPU has finished all computations and results can be copied back to the CPU. A simple call to `cudaThreadSynchronize()` by the application blocks the CPU thread until all operations are finished. Sections 3.3.2 and 3.3.3 will present more fine-grained synchronization methods.

To manage the cooperation between threads of the same thread block through shared memory, the `__syncthreads()` intrinsic is used in kernel code. This barrier synchronization ensures that all GPU threads within a thread block have reached this point before proceeding further. Since thread blocks have to be independent, there is no global synchronization barrier for all threads. If this is necessary, the algorithm has to be split into several consecutive kernel calls.

3.3 Features

In the following, some features of CUDA will be presented that I used for an efficient integration of CUDA in NMM. Most of these features target performance, whereas *contexts* are needed to use CUDA in a multi-threaded application such as NMM. The supported features may vary depending on the compute capability which defines the core architecture of the GPU. A list of existing compute capabilities and corresponding GPUs is given in [38] (Appendix A). If not stated otherwise, the following features are supported by all GPUs.

3.3.1 Asynchronous Operations

In CUDA the GPU is used as a co-processor for the CPU, but the CPU also has significant computational power that should not be wasted by just waiting for the GPU. The wasting of computational power can be minimized by using asynchronous operations.

Some operations in CUDA, for example a memory copy between CPU and GPU, block the CPU thread until the GPU carried out the request before proceeding with the next instruction. For many operations, CUDA provides asynchronous counterparts that do not block the CPU thread. Then, the application has to ensure synchronization in order to get correct results. Since synchronizations are busy-waits, the challenge is to minimize the necessary synchronizations in order to achieve best utilization of the CPU and the GPU for processing tasks.

Figure 3.5 illustrates the benefits of using asynchronous operations: The CPU issues the operations asynchronously, which only takes up a small fraction of time. Then the CPU is free to do arbitrary operations while the GPU is working off the tasks. By default, asynchronous operations are queued in a default *stream*. A call to `cudaStreamSynchronize()` is blocking the CPU thread to achieve synchronization between CPU and GPU.

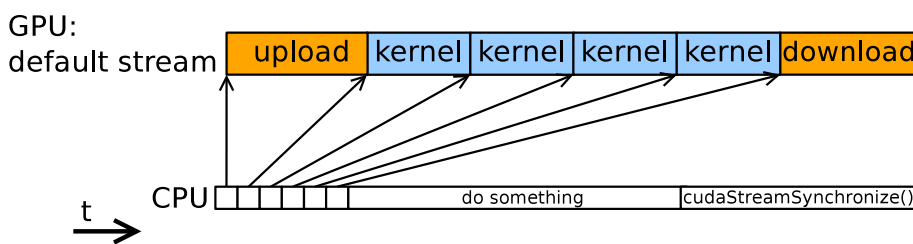


Figure 3.5: Asynchronous operations do not block the CPU thread until an explicit synchronization. *Upload* means a memory copy from CPU to GPU, *download* vice versa.

3.3.2 Streams

Asynchronous operations like memory copies or kernel invocations can be assigned to streams by passing an additional stream parameter to the calling function. Streams can be considered as independent sequences of operations. Operations assigned to the same stream are executed in exactly the same order

they have been added to the stream without the necessity of an explicit synchronization. Operations in different streams should be independent and can be executed in parallel. As already shown in figure 3.5, individual streams can be synchronized while other streams are not affected. Alternatively, a stream can be queried for completion which is a non-blocking operation.

If there exists more than one stream, CUDA switches between the streams in a round-robin style and executes the next operation in it. At the moment (with CUDA 2.0) it is possible to interleave memory copies with kernel executions of another stream, as illustrated in figure 3.6. GPUs with compute capability 1.0 are not capable to overlap kernel executions with memory copies [38] (Appendix A). Performing multiple memory copies at the same time is not supported. In future releases the parallel execution of multiple kernels might become possible by using streams.

Streams are a concept to minimize the overhead induced by the memory copies. This is especially valuable for small data items that have to be copied to the GPU because they induce a lot of overhead to set up the memory transfer.

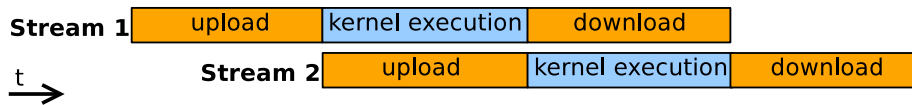


Figure 3.6: CUDA Streams: Overlapping of kernel executions with memory copies of another stream. Total execution time is $1 * upload + 2 * kernel + 1 * download$

3.3.3 Events

In some cases a very fine grained synchronization between the CPU and the GPU might be desirable because a global synchronization using `cudaThreadSynchronize()` or even a stream-based synchronization could be inappropriate or just too expensive. Then the use of *events* can be more adequate.

CUDA events can be added to streams and are recorded if the preceding operations in the stream have been finished. Analogously to streams, the application can query an event if it has been recorded and it can explicitly synchronize for an event. So events can be used for very lightweight synchronizations as shown in figure 3.7 or for profiling execution times.

Unfortunately, events in CUDA 2.0 are not implemented as expected when using them in combination with multiple streams. Listing 3.4 shows a scenario where events fail. Both events are recorded in `stream_0` with two kernel calls in between. Since only one kernel is executed in `stream_0` the elapsed time between the events should only profile a single kernel call. But actually, it profiles both kernel calls. A similar test application can be found in the Nvidia CUDA forums¹.

Nvidia states that the `cuEventRecord()` function basically ignores the stream parameter in order to allow profiling of events from different streams. They propose to use the workaround presented in listing 3.5 where the events

¹<http://forums.nvidia.com/index.php?showtopic=81823>

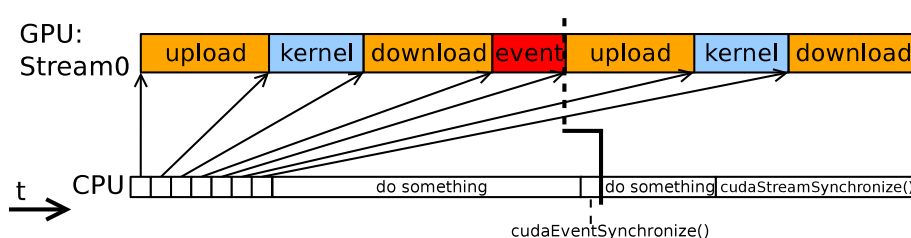


Figure 3.7: CUDA Events can be used as marker points for synchronizing CPU and GPU execution. In this example, the event has been recorded before the CPU issues the synchronization, so the synchronization returns immediately.

only bracket the code that should be profiled (regardless of streams). This workaround works quite well for serial applications, but it fails in multithreaded applications where this execution order can not be guaranteed.

Listing 3.4: Recording CUDA Events with multiple streams which does not yield the expected results (pseudocode)

```

1 cuEventRecord(start_event, stream_0);
2 cuLaunchGridAsync(kernel, stream_0);
3 cuLaunchGridAsync(kernel, stream_1);
4 cuEventRecord(stop_event, stream_0);

```

Listing 3.5: Workaround to use events with multiple streams (pseudocode)

```

1 cuEventRecord(start_event, stream_0);
2 cuLaunchGridAsync(kernel, stream_0);
3 cuEventRecord(stop_event, stream_0);
4 cuLaunchGridAsync(kernel, stream_1);

```

3.3.4 Contexts

CUDA resources are bound to a specific *context* which is roughly equivalent to a CPU process. In particular, memory in GPU or in page-locked CPU memory (Section 3.3.5) can only be accessed within the context in which it has been allocated. The context is determined by the CPU thread that issues the function call to the CUDA runtime. When a CPU thread invokes a kernel function, it has to be ensured that all memory resources that are accessed by the kernel have been allocated within the same context.

In general, each CPU thread has its own context which makes it impossible to share CUDA resources among different application threads. For serial applications this is unproblematic, but it seriously complicates the integration of CUDA into multithreaded applications where many CPU threads invoke kernels on the same data. When using the *CUDA Runtime API* it is not possible to access multiple GPUs by the same CPU thread. Different GPUs always have different contexts which requires one CPU thread per GPU.

An explicit context management is only available through the low-level *CUDA Driver API*, not through the *CUDA Runtime API*. The Driver API allows to create contexts for a specific GPU and to assign it to a CPU thread. Furthermore, it allows to release a context from a CPU thread. This context

is then floating and can be assigned to another CPU thread. The contexts assigned to a CPU thread are organized in a stack, but only the top-most is actually active in the CPU thread. A context can only be active in exactly one CPU thread at the same time, which is suited for a peer model of threading. To my best knowledge there is no mechanism that prevents to attach a single context to multiple CPU threads, potentially resulting in undefined behavior. Additionally, releasing a context from a CPU thread in CUDA 2.0 does not return a handle to the floating context as described in the CUDA reference manual [39]. This further complicates using the context management and is a known bug². Most likely the context management will be changed in future releases of CUDA.

Due to the difficulties and limitations with context management a widely used approach is to statically assign a context to a worker thread. Application threads then use the worker threads to call CUDA functions. To realize this concept, there exists a nice helper class called *GPUWorker*³ which is built on top of the *Boost C++ library*. Recall that a context is always bound to a GPU, so at least one worker thread for each GPU is required. For the rest of this thesis I will assume contexts to be statically assigned to worker threads and neglect the explicit context management as described in this section.

One should mention that there is a limit of 16 contexts per GPU in Windows and probably there will be a limit in Linux, too⁴. Using multiple worker threads (i.e. multiple contexts) per GPU could have the benefit that a worker thread can issue new operations to the GPU while another worker thread is blocked waiting for synchronization. In any case, it turned out that using multiple contexts and quickly switching the thread context imposes a significant overhead on the GPU, but this applies only to a single GPU⁵.

3.3.5 Page-locked CPU Memory

The memory on CPU side has to be allocated as page-locked in order to use fast DMA (*Direct Memory Access*) transfers between CPU and GPU memory spaces, and to overlap kernel execution with memory copies using the concept of *CUDA Streams*. Page-locked memory is bound to a context, so only the CPU thread that allocated the memory is able to use fast DMA transfers to copy data to the GPU or vice versa. Consequently, the allocation of page-locked CPU as well as GPU memory and the corresponding memory copy has to be performed by the same CPU thread. Nevertheless, every CPU thread can read and write page-locked memory since it is available in the address space of the application.

The recent release of CUDA 2.2 solved this problem and allows to share page-locked memory among different contexts⁶. Even a *zero-copy* feature is available that enables the GPU to directly access certain CPU memory without an explicit copy to global memory. When writing this thesis, these features were not available in CUDA 2.0 and could not be considered.

²<http://forums.nvidia.com/index.php?showtopic=81300>

³<http://forums.nvidia.com/index.php?showtopic=66598>

⁴<http://forums.nvidia.com/index.php?showtopic=80221>

⁵<http://forums.nvidia.com/index.php?showtopic=88753>

⁶<http://forums.nvidia.com/index.php?showtopic=96303>

3.3.6 Multiple GPUs

It is required that SLI (*Scalable Link Interface*) is disabled in systems with multiple GPUs in order to use them with CUDA. It is not possible to link multiple GPUs such that they act as one GPU with multiplied compute resources. Furthermore, there is no memory copy between different GPUs without an intermediate copy to CPU memory. Application developers have to distribute independent tasks to multiple GPUs explicitly.

3.4 Summary

CUDA enables to use the GPU as a co-processor that offers a very high floating performance on data-parallel algorithms, but the GPU can not be used as an independent processor. The CPU is always required to manage processing on the GPU.

Memory blocks on the GPU have to be allocated and managed by the application running on the CPU. Input and output data of a kernel function always reside in global GPU memory and the application has to copy data between CPU and GPU memory. There exist different types of memory in the global memory region of the GPU, but only linear memory is read and writeable by a kernel. Since the global memory region is shared among all processor cores it easily becomes the bottleneck in a shared memory architecture (Section 2.2). Thus, kernel functions can use different types of on-chip memory and caches to save bandwidth to the global memory region.

Since CUDA allows to invoke operations on the GPU asynchronously, these should be used whenever possible to free CPU time for other computational tasks. Combining dependent operations to streams ensures the ordering of the operations. When used together with page-locked memory, kernel executions and memory copies in different streams can be overlapped to reduce the overhead. Nevertheless, memory copies between CPU and GPU memory are rather expensive and should be minimized. Note that a memory copy from pageable to page-locked memory on CPU side is inefficient and would reduce the benefit of the fast DMA transfers. When integrating CUDA into a multithreaded application, it has to be considered that CUDA resources are bound to a specific context. Finally, CUDA does not provide abstractions to automatically use multiple GPUs, so applications have to handle work distribution to the GPUs explicitly.

In order to use these features efficiently and to fulfill the goals stated in section 1.2, this places some requirements on the framework to integrate CUDA into the NMM middleware. First of all, processing on the GPU has to be hidden from the applications built on top of NMM. In particular the problem of different contexts has to be handled automatically. Moreover, CUDA kernels have to be integrated into modular nodes which provide abstract interfaces to the application. These nodes have to perform the control logic on the CPU because the GPU can only execute algorithms for data processing. Data has to be copied to GPU memory before it can be processed. However, this is only necessary when connecting nodes that process data in different address spaces (e.g. on the CPU or the GPU). Thus, a separation of data transfer and data processing is required in order to minimize memory copies between the CPU

and the GPU. Furthermore, memory on CPU side has to be allocated as page-locked in order to use DMA transfers. An efficient combination of processing on the GPU *and* the CPU within different nodes of a flow graph requires that CUDA operations are invoked asynchronously. A strategy is required in order to automatically use the concept of streams to minimize the necessary synchronizations. Finally, the framework has to distribute work to all locally available GPUs so that applications can scale to any number of GPUs within a single system.

Chapter 4

Related Work

Section 1.1 motivated the integration of CUDA into a distributed middleware for multimedia processing. This chapter presents related work with an overview on distributed middleware solutions that integrated CUDA support. Section 4.2 presents different GPGPU programming frameworks. Section 4.3 gives an overview of existing multimedia libraries and applications using CUDA for processing on the GPU.

4.1 Distributed Middleware Solutions using CUDA

The BOINC project [18] (*Berkeley Open Infrastructure for Network Computing*) integrated CUDA in their grid computing framework. BOINC is used by the SETI@Home and the GPUGRID project [27] which achieve a two- to ten-fold speed-up over desktop CPUs when using CUDA¹. The BOINC framework and its related projects focus on grid computing where independent tasks are processed on remote machines. Despite the exchange of data that should be processed by the clients, BOINC does not provide a communication infrastructure and thus it is not suited for multimedia processing.

The *distributed.net* project [16] offers CUDA enabled clients that leverage the computational power of their GPU for distributed computing. It focuses on solving cryptographic and mathematical problems. Data pieces are sent out by a server to a huge number of participants that process these data and send back the result. But there is no way to control processing on the clients.

Hartley et al. [28] developed a framework for distributed processing on a cluster built of multi-core CPUs and GPUs to speed-up compute intensive tasks in the field of biomedical image analysis. It is built on top of the DataCutter middleware [20] to distribute data pieces to multiple compute nodes. Similar to the BOINC project it focuses on processing of independent data pieces and thus it is not suitable for multimedia processing.

¹http://www.nvidia.com/object/io_1229516081227.html

4.2 GPU Programming Frameworks

Early approaches to use the GPU for non-graphics computations use shader languages such as *Cg* [33] or the *OpenGL Shading Language* [13] to access the compute resources of the GPU through a graphics API. The major drawback of these approaches is that developers have to think in terms of graphics computations such as drawing geometric primitives. Additional programming effort is necessary to learn new programming languages and concepts in order to rewrite algorithms. Since shader languages provide insufficient abstractions for GPGPU developers and are outdated for most application scenarios, they are not considered for this work.

The *BrookGPU* [21] stream programming language, which is an extension to the *Brook* streaming language, is especially designed for general purpose computations on the GPU. Streams in BrookGPU are conceptually similar to ordinary arrays but kernel functions can process the elements in parallel [44]. BrookGPU supports several graphics APIs as backend for the computations and automatically compiles kernel functions to shader programs within a graphics pipeline. With BrookGPU parallel algorithms can be readily expressed much more intuitive as known from programming for the CPU.

Recent programming models such as *CUDA* or *ATI Stream* make the GPU directly accessible to developers without the need of a graphics API in between. This allows to write much more intuitive code than using shader languages, because no abstraction towards graphics computations are needed. CUDA is proprietary software, so only Nvidia GPUs are supported but there exists a source-to-source translation framework for multi-core CPUs, called MCUDA [53]. Nevertheless, CUDA is used for this work because it is freely available, well advanced and supported by many developers that integrated CUDA into their applications.

ATI provides an alternative to CUDA, formerly known as CTM (*Close To Metal*) that completely exposes the architecture-level ISA to the developer, whereas CUDA has a more general, unified model [49]. The commercial successor of CTM is now available in the *Stream SDK* [8] that ships with an ATI optimized version of Brook, called *Brook+*. The Stream SDK, however, does not get the public attention that CUDA gets, probably because it is considered more difficult to program.

OpenCL [12] is an open standard developed by the Khronos Group. Programs written with OpenCL can be executed on different platforms such as GPUs, multi-core CPUs and Cell-type architectures. Nvidia announced that OpenCL will be supported by CUDA². Since a first implementation will be released in 2009, it could not be used for this thesis.

Rapidmind [35] is a multi-core development platform that supports a variety of many-core technologies. Programs can be developed in standard C++ while the underlying hardware is abstracted from developers. Since it is a commercial product it is not used for this work.

²http://www.nvidia.com/object/io_1228825271885.html

4.3 GPU-Accelerated Multimedia Libraries and Applications

Due to the wide acceptance of CUDA there already exist some multimedia libraries that achieve significant speed-ups using the GPU for compute intensive algorithms.

The *OpenVidia* [25] project provides computer vision libraries that are using the GPU and CUDA to allow real-time imaging much faster than CPUs are capable. The usage of multiple GPUs to build up a cheap and very powerful parallel architecture is also supported.

The *Institute for Computer Graphics and Vision* of the *Graz University of Technology* develops algorithms for computer vision that are publically available in the *GPU4Vision* [10] project. Another computer vision library is *GpuCV* [17] which is compatible to the OpenCV standard.

Due to the coding efficiency of high-definition video codecs such as H.264/AVC, de- and encoding such video streams is a compute intensive task. Pieters et al. [46] evaluated the performance of H.264/AVC decoding and visualization using the GPU. They developed a CUDA enabled decoder, that achieves significant speed-ups compared to a CPU-only solution.

DGAVCDec [9] is a freely available AVC/H.264 decoder realized as a plug-in for the AviSynth frameserver. This decoder uses the *CUDA Video Decoder API* (NVCUVID) to get access to the hardware decoder of the GPU. Unfortunately, it is currently only available on Windows.

There also exists a GPU accelerated version of the fairly new and highly efficient Dirac video codec [55]. Using CUDA, a sevenfold speed-up in decoding could be achieved on a GeForce 8800 graphics card compared to a CPU implementation.

In the scope of a CUDA coding contest [4] initiated by Nvidia a LAME MP3 encoder has been developed using CUDA.

There also exist some applications for multimedia processing that integrated CUDA support. Unfortunately, all of them are commercial products and only available for Windows.

The *Badaboom Media Converter* [22] by Elemental Technologies uses CUDA to speed-up the transcoding tasks of high definition video material into a format suited for mobile devices. There is a review³ that demonstrates that using the GPU for video transcoding can even outperform state of the art CPUs like the Intel Core i7.

The commercial *TMPEGEnc Xpress 4.0* [45] video encoding software can offload decoding and video filter tasks to the GPU. For certain algorithms the software producer claims to achieve a speed-up of more than 4 times using a Nvidia GTX 260 GPU compared to a Intel Core2 Quad 2.66 GHz.

Cyberlink offers a video editing software called *PowerDirector* [15] that accelerates video encoding into the H.264 format and video rendering tasks.

MotionDSP developed a video enhancement software called *vReveal* [36] which should achieve a performance speed-up of up to 5 times. No information is given on which tasks this speed-up can be achieved.

³<http://www.anandtech.com/video/showdoc.aspx?i=3475>

4.4 Summary

The presented middleware solutions that integrated CUDA mainly target grid computing. They do not support dependencies between the computations on different computing nodes as the concept of a distributed flow graph in NMM does. Despite that, they do not provide a transparent communication between the nodes and the application. Thus, they are not suited for multimedia processing.

Among the various GPGPU programming systems, CUDA offers the most convenient abstractions for developers and is already adopted in many applications. Thus, CUDA is used to integrate processing on the GPU into NMM.

The presented libraries act as black box solutions providing an interface that completely hides the usage of the GPU. While this makes it easy to integrate GPU support into applications it makes it hard to combine them with other libraries *efficiently*, because memory transfers between the CPU and the GPU are handled within the libraries. This will result in unrequited overhead by unnecessary memory transfers. Furthermore, it is noticeable that there only exist commercial video encoders for CUDA.

As already mentioned, CUDA distinguishes between the Driver API and the Runtime API. In any case, the implemented kernels are identical and can be reused within the presented framework. Since the functionality of the Driver API includes the functionality of the Runtime API, programs can be ported by replacing the CPU code with corresponding functions from the Driver API. The kernel functions can be extracted from the source code by the `nvcc` compiler in order to build the necessary kernel module.

Chapter 5

NMM - Network-Integrated Multimedia Middleware

The *Network-Integrated Multimedia Middleware* (NMM) has been developed at the Computer Graphics Lab [3] at the Saarland University and by the Motama GmbH [1]. It is available as an open-source project [2] under the terms of the GNU General Public License (GPL), but also as a commercial version distributed by the Motama GmbH. NMM supports different platforms such as Linux, Windows, MacOS and there even exist versions for the Playstation 3 and the Apple iPhone.

NMM is designed under the aspect of openness and extensibility [31] such that new technologies can easily be integrated. NMM provides abstractions to access resources in a network transparent way, but also allows to configure lower layers manually to maintain a high level of control if necessary. Due to its extendable communication mechanisms it is ideally suited to integrate processing on the GPU in the sense of a distributed system as stated in section 1.1.

In the scope of a Diploma thesis NMM has already been used to integrate processing on the Cell Broadband Engine (CBE) [51]. The Cell Broadband Engine uses specialized processing cores that are explicitly designed for multimedia and gaming applications. Due to its hard technical constraints, programming the CBE is a difficult task.

5.1 Overview

According to [31], a middleware provides an abstraction layer between the operating systems of distributed systems and the applications running on them. The Network-Integrated Multimedia Middleware targets the distributed processing of multimedia data in which all components in the network can be transparently controlled by the application. A multimedia stream consists of the actual multimedia data and control information that have to be exchanged between distributed systems.

In contrast to client/server streaming (Figure 5.1 (a)) the middleware approach of NMM shown in figure 5.1 (b) allows applications to take full control over all components in a distributed environment. Streaming applications do in general not support this or they have to provide external management tools.

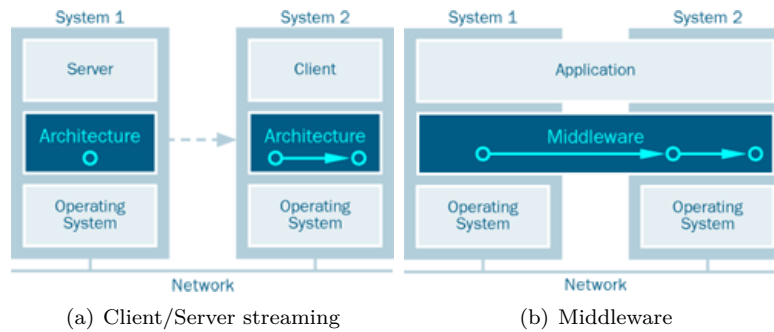


Figure 5.1: Client/server approach and middleware. [2]

NMM, however, provides well defined interfaces and transparently integrates communication aspects with the actual multimedia processing.

The following sections will present a brief overview of the components of NMM that are relevant for this work. A detailed description can be found in the online documentation of NMM [2].

5.2 Distributed Flow Graph

NMM provides modular processing components, called *nodes*, that provide input and output *jacks* that can be connected by *edges* to build a flow graph. A flow graph specifies how multimedia data is processed when it passes through the flow graph from *source* node to *sink* node. The source nodes are used to produce data, while the sink nodes consume data. The nodes in between usually implement self-contained processing steps. Figure 5.2 presents a simple flow graph for MP3 playback. It consists of a `MP3ReadNode` source, a `MPEGAudioDecodeNode` to decode the MP3 file and an `ALSAPlaybackNode`, which is the sink node used for playback over a sound device.



Figure 5.2: Simple MP3 decoding flow graph in NMM with 3 nodes.

5.3 Nodes

Nodes can provide several input and output jacks that are used to receive input or send output data respectively. For each jack, a node provides a list of supported *formats*, only nodes with at least one matching format can be connected. The nodes are classified by different types, depending on their functionality and the number of jacks [31]. These are shortly summarized in table 5.1.

The nodes in NMM encapsulate hard- or software functionality and hide all implementation details. To control nodes from the application they provide

Type of node	Input jacks	Output jacks	Description
Source	-	1	produces data (e.g., MP3ReadNode)
Sink	1	-	consumes data (e.g., ALSAPlaybackNode)
Filter	1	1	processes data without changing its format (e.g., DeInterlaceNode)
Converter	1	1	processes data and changes the output format (e.g., YUVtoRGBConverterNode)
Multiplexer	>1	1	takes data from several input jacks and combines them to create new data (e.g., OverlayNode)
DeMultiplexer	1	>1	splits data from the input jack into its components (e.g., MPEGDemuxNode)
MuxDemux	>1	>1	combination of Multiplexer and DeMultiplexer

Table 5.1: Different Types of Nodes in NMM

interfaces that can be requested by the application. Node developers can create new interfaces using an *Interface Definition Language* (IDL). By using interfaces, for the application it does not make any difference if a node is running locally or on a remote host. Necessary communication is automatically handled by the middleware. This allows applications to take full control over all components in a flow graph.

5.4 Formats

As already mentioned, node developers have to specify a list of supported *formats* for their input and output jacks. A format describes the encoding of the media data and contains information such as `video/raw` for uncompressed video streams, information on the colorspace (e.g. `rgb24`) and the resolution (e.g., of a video stream). Nodes with a matching pair of input/output formats can be connected in a flow graph.

5.5 Messages

A multimedia stream consists of the actual multimedia data and control information that signal for example the end of a file or changes in format. In NMM, a multimedia stream is represented by two types of messages: The *buffers* contain the multimedia data whereas *events* represent control information.

5.5.1 Buffers and Buffer Managers

Buffers are messages that are streamed through the multimedia flow graph and contain the actual multimedia data plus some additional header information such as timestamps. Buffers received at an input jack are processed by the node and forwarded to the output jack. A node can also forward a buffer to multiple output jacks, in this case the buffer is shared in the flow graph. Hence, a node that wants to modify a buffer has to request a writeable instance of it.

Buffers are always managed by a *buffer manager* which allocates memory blocks and provides a pool of reuseable buffers. Nodes can be configured to use a specific buffer manager to store their results. Additionally, buffer managers can be shared among multiple nodes.

5.5.2 CEvents

The other type of messages are the so-called *Events*, which are used to send control information. Control information are very important for a multimedia stream, since they can notify a node if for example the format of a video stream changed.

Events can either be sent *instream*, that is along the flow graph, or *out-of-band* which means between the application and the nodes. For instream communication so-called *CEvents* (*Composite Events*) are used that contain a number of events to be handled within a single step of execution. CEvents can be generated by every node and inserted in the message stream. Furthermore, instream events can be sent in downstream direction (from source to sink node) or in upstream direction. A strict message order of buffers and CEvents is ensured by the transport mechanism.

5.6 Transport Strategies

The *Transport Strategies* build the edges of the flow graph and connect the nodes of a distributed flow graph to forward instream messages. There exist transport strategies to realize network connections over TCP, UDP or RTP as well as a *LocalStrategy* to connect nodes on the same host by pointer forwarding. Figure 5.3 shows a sample flow graph with a source node that sends an encoded MP3 stream over a TCP connection. A decode node running on another host is used to decode the MP3 stream and sends it to a local sink node that realizes playback over an ALSA audio device.

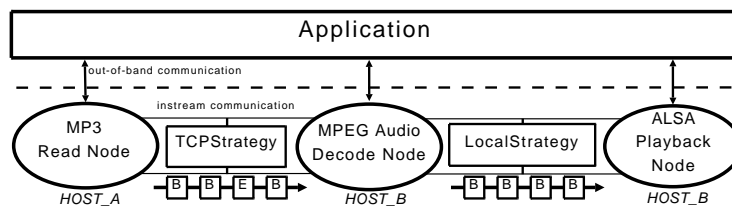


Figure 5.3: A distributed flow graph with transport strategies connecting the input and output jacks of the nodes running on different hosts. Buffers *B* and events *E* are forwarded instream. Communication with the application is handled out-of-band.

The nodes of a flow graph do not know about the transport strategies connecting their input and output jacks. A suitable transport strategy is automatically proposed to the application according to a *UserInfo* but can be reconfigured using *explicit binding*. The *UserInfo* contains information about the domain, the host a node is running on and the process ID of the application. Nodes provide this *UserInfo* while a *connection service* can choose the most suitable transport strategy that meets the specified requirements. The strict separation of media transport and processing allows to easily setup distributed applications.

In some cases, buffers can be sent over an unreliable connection such as UDP whereas reliable control events need to be transferred over TCP. NMM supports a *parallel binding* of transport strategies [48], which allows for example to send

events over a TCP connection and buffers over a UDP connection. Figure 5.4 shows a separation of event and buffer transmission while a strict message order is automatically preserved.

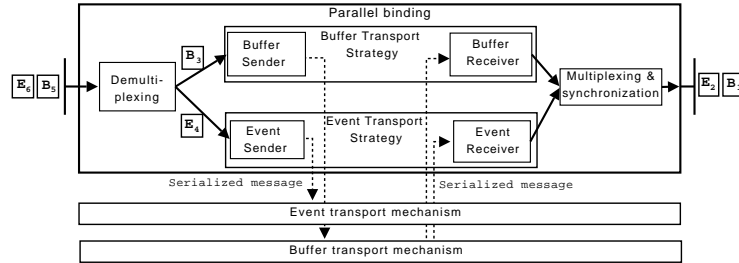


Figure 5.4: The buffers B and events E can be sent by different transport strategies within a parallel binding. The message order of the multimedia stream is automatically preserved. [47]

5.7 States

Nodes in NMM have several states and corresponding state transitions, which are dedicated to specific actions of a node. Figure 5.7 shows a schematic overview of the available states. The `do`-methods can be implemented by a node to perform tasks during the state transition.

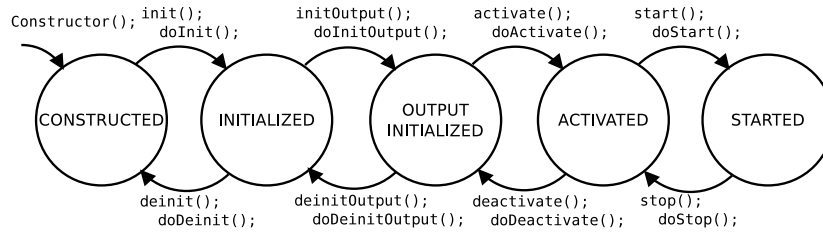


Figure 5.5: States and state transitions of NMM nodes. [11]

The intended purpose of the state transitions is summarized below:

- `init()`: reserve resources; specify generally supported formats
- `initOutput()`: reserve resources; specify supported formats for this instance of the node
- `activate()`: reserve resources depending on the chosen formats
- `start()`: start processing of instream events and buffers
- `setEnabled(true | false)`: enable or disable filtering (only filter nodes)
- `stop()`: stop processing of instream events and buffers
- `flush()`: release all internal resources (queues, internal buffers, ...)
- `deactivate()`: free resources allocated in activate
- `deinitOutput()`: free resources allocated in initOutput
- `deinit()`: free resources allocated in init

5.8 NMM Services and Applications

This section shortly presents some services and applications of NMM.

Registry The *registry service* is used to provide nodes on a local or remote host. Therefore, each host has to run a unique *serverregistry* which provides a *node description* for each locally available node [11]. Applications can request nodes from the registry service to build a flow graph with nodes available on the local or a remote host.

Session sharing This service allows to share nodes of a running flow graph between different applications. Users can share limited hardware resources such as TV cards while different displays can be used to watch TV. In contrast to streaming applications, the NMM middleware allows each user to control the nodes, for example to switch the program.

Clic For applications that do not need user-interaction, the application *clic* can be used to easily setup a distributed flow graph from a textual description, called *graph description*. Listing 5.1 shows the graph description for the flow graph seen in figure 5.3.

Listing 5.1: Graph description for the flow graph in figure 5.3

```
1  MP3ReadNode # setLocation("host_a")
2  ! MPEGAudioDecodeNode # setLocation("host_b")
3  ! ALSAPlaybackNode # setLocation("host_b")
```

Chapter 6

Design of a CUDA-NMM Framework

In the previous chapters, concepts of the NMM middleware and particular features of CUDA have been presented. This chapter describes the design of the framework to integrate CUDA into NMM and highlights the considerations that lead to that particular design. This was done under the aspect of an efficient integration of CUDA using the features presented in section 3.3, while the goals stated in section 1.2 should be achieved. Concepts of NMM have been reused or extended and necessary components were added to neatly couple the framework with existing NMM concepts.

The design of this framework is also described in [47]. Implementation details are presented in chapter 7. A guideline on how to implement new NMM nodes using the CUDA-NMM framework can be found in chapter 8.

The general idea of the framework is to have kernel functions encapsulated in the nodes of a flow graph. In the following, nodes using the CPU for media processing are called *CPU nodes* whereas nodes using the GPU are called *GPU nodes*. Transport strategies named *CPUtoGPUStrategy* and *GPUtoCPUStrategy* perform the necessary memory copies between CPU and GPU memory. This realizes a separation of data transfer and data processing because GPU nodes have input and output data in GPU memory. A *LocalStrategy* only performs a pointer forwarding between adjacent GPU nodes since no memory copies are needed at all. A schematic overview is presented in figure 6.1. A buffer that passes through the flow graph is copied to the GPU and processed by a series of GPU nodes that issue the kernel functions. Finally the results are copied back to CPU memory.

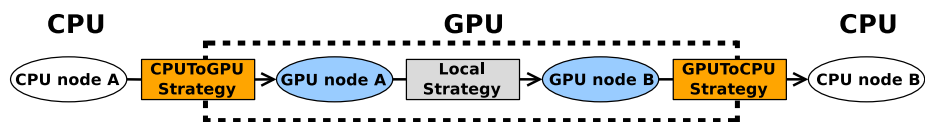


Figure 6.1: Overview of NMM flow graph with GPU nodes. The transport strategies copy data between CPU and GPU memory.

Among the various features of CUDA, the concept of streams is the most considerable one. The usage of streams constitutes the basis for asynchronous operations and the overlapping of kernel execution with memory transfers. Due to their importance, I will start with an examination of how streams can be applied when integrating CUDA in NMM.

6.1 CUDA Streams

Using streams to overlap memory transfers and kernel executions can significantly improve performance. Different approaches to use streams in the CUDA-NMM framework are presented in this section. A performance comparison can be found at the end of this section. All approaches use streams to queue asynchronous operations.

6.1.1 Approach 0: One global Stream

Although all CUDA enabled GPUs support the concept of streams and asynchronous operations, only GPUs with a compute capability 1.1 or above can benefit from streams to interleave memory transfers and kernel executions. This approach mainly targets GPUs with compute capability 1.0 and only uses the default stream to queue asynchronous operations. Possible overheads by using multiple streams could be avoided.

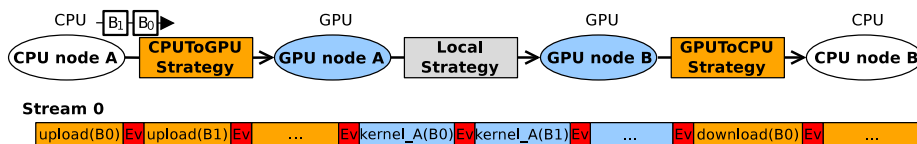


Figure 6.2: Approach 0: Using a single CUDA stream to queue all CUDA operations asynchronously. CUDA events are abbreviated by *Ev*, NMM buffers by *B*. *Upload* means a memory copy from CPU to GPU, *download* vice versa.

As shown in figure 6.2 the transport strategies and the nodes queue their operations to a single stream. Streams are per context which limits the number of worker threads to one (for each GPU). Adding a CUDA event after each asynchronous operation enables for an efficient synchronization between CPU and GPU based on events.

It has to be stressed that using one stream prevents any overlapping to occur. Furthermore, all nodes in NMM are running in their own CPU thread. Most probably this would result in jittering because the CPUToGPUStrategy could issue many upload operations before the first buffer is actually processed. Workarounds to avoid this jittering could seriously complicate buffer processing.

6.1.2 Approach 1: One Stream per NMM Buffer

This approach assigns a unique stream to every buffer that is requested from a buffer manager. Consecutive transport strategies and nodes would queue asynchronous memory copies and kernels in the stream of the buffer that is currently processed. Figure 6.3 illustrates that operations on different buffers

can overlap. This approach is the most natural way to use CUDA streams because dependent operations on a buffer are assigned to the same stream. Benchmarks will show that this is also the fastest approach.

As described in section 3.3.3, CUDA events cannot be used reliably with this approach because multiple streams are used.

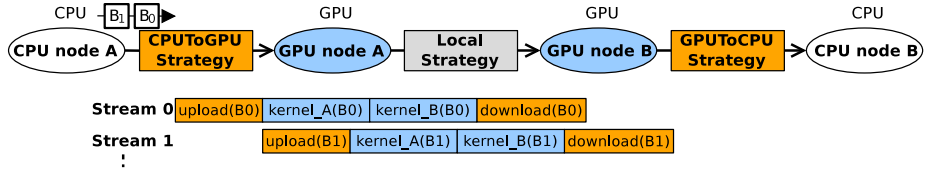


Figure 6.3: Approach 1: Each buffer is assigned its own stream. CUDA operations can be queued asynchronously to a buffer specific stream.

6.1.3 Approach 2: One Stream per NMM Node

The last approach can be considered as the reverse to approach 1. Each transport strategy and each node maintains its own CUDA stream. Since dependent operations on a buffer are not in the same stream anymore, some kind of notification is required to inform the successor that a certain operation is finished and data is ready to be processed.

This notification could be realized with CUDA events that are recorded after every asynchronous operation. The next component could synchronize for these events to ensure the predecessor is done with a buffer. This is shown in figure 6.4.

This approach relies on the combination of multiple streams using events for synchronization. As already stated, CUDA does not implement this functionality as expected. A workaround would be to synchronize the whole stream of the predecessor before processing a buffer. Of course, this is accompanied by a high overhead.

However, this approach is promising because it allows to copy data to the GPU long before it will be processed. This kind of 'pre-caching' can eliminate latencies if a node has to wait for copy operations to complete.

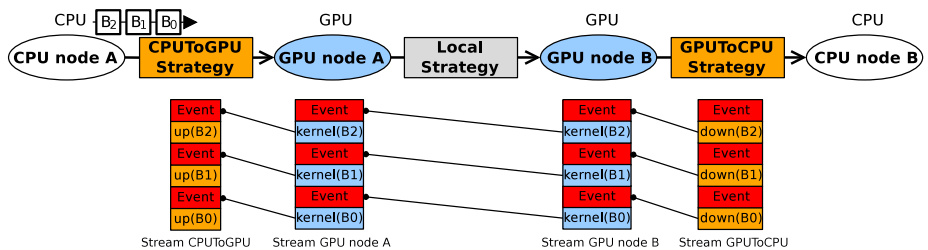


Figure 6.4: Approach 2: Each NMM object uses its own stream for CUDA operations. Synchronizations are performed based on events.

6.1.4 Performance Comparison

Although using one stream per buffer is the most native approach, the others also have their benefits, either by allowing for event based synchronization or by the concept of pre-caching.

I wrote a small test program that simulates buffer processing and tests the different approaches for their performance: Data is copied to the GPU, processed by 2 consecutive kernel functions and finally copied back to the CPU. The possible overlap seriously depends on the ratio between execution time of the kernel and the memory copy of the data. Figure 6.5 shows the processing time for all 3 approaches on a Nvidia GeForce 9600 GT graphics card (Compute Capability 1.1). Using one stream per buffer is the fastest approach for every buffer size. Approach 2 only works because in a serial application the workaround proposed by Nvidia is applicable. However, in a multithreaded application it does not work (Section 3.3.3).

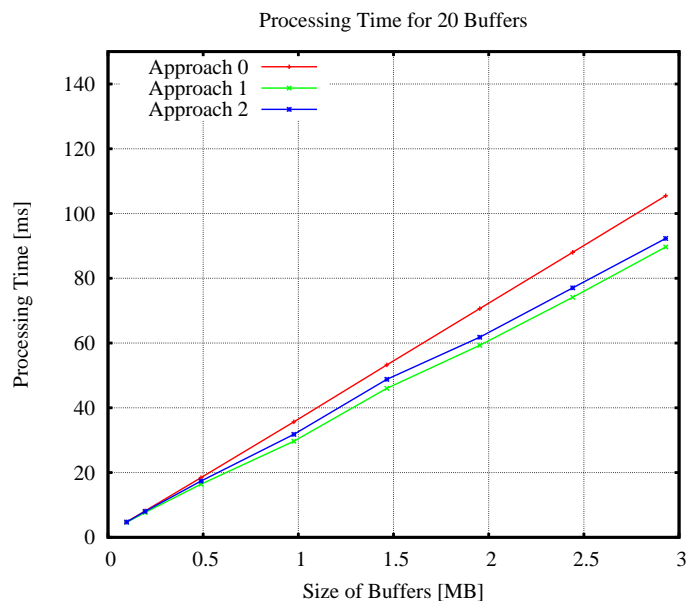


Figure 6.5: Benchmark of the 3 approaches. Measures the times (in ms) to process 20 buffers: copy the buffers to the GPU, process them by 2 consecutive kernels, copy the results back to the CPU. Benchmark is repeated for buffer sizes between 100 KB and 3 MB. Approach 1 with one stream per NMM buffer scales best for all buffer sizes. (GPU: GeForce 9600 GT)

Figure 6.6 shows the same test on a Compute Capability 1.0 card. On this GPU all approaches deliver the same performance, so even on compute capability 1.0 GPUs the usage of streams does not add a detectable overhead.

The framework will be implemented using one stream per buffer, because this is the fastest approach in all cases and approach 2 cannot be implemented reliably.

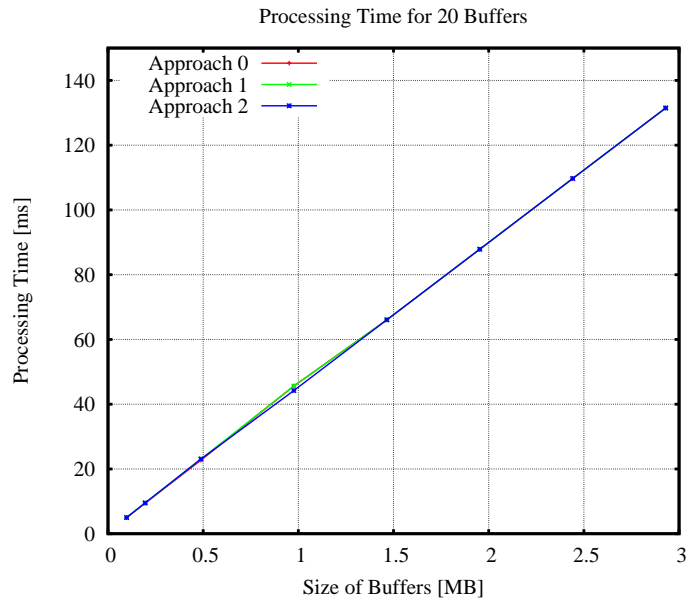


Figure 6.6: Benchmark of the 3 approaches to use streams on a Tesla C870 GPU. This GPU only has Compute Capability 1.0, so it supports streams but it can not use streams to overlap memory transfers with kernel execution. All 3 approaches show almost the same performance.

6.2 Key Design Decisions

First of all the *CUDA Driver API* has been chosen to implement the framework because it allows better control than the *Runtime API*. For example, the Driver API provides features such as access to contexts and better control over GPU initialization. The differences between the Driver and the Runtime API only affect the CPU code, the actual kernel code is unchanged. Certainly it is harder to program, but if a particular feature will be needed that is not available in the Runtime API, the whole framework would have to be ported to the Driver API, this is completely unacceptable.

Processing on the GPU should be transparently for application developers. This is achieved by integrating CUDA kernels into the nodes of a flow graph which provide an abstract interface to the application. In this way existing applications can be easily modified to use nodes that process data on the GPU.

Memory transfers between CPU and GPU are encapsulated in the edges of a flow graph, so input and output data of a GPU node are always in GPU memory. Adjacent GPU nodes can then be connected by pointer forwarding so that no additional memory transfers between CPU and GPU have to be added.

The concept of asynchronous CUDA operations that do not block the CPU should be used whenever possible. This saves CPU time and allows to combine processing on the CPU and the GPU in different nodes of the flow graph. The elementary synchronizations of asynchronous operations should be handled automatically.

Further reducing the overhead by memory transfers is possible using CUDA

streams together with page-locked memory on CPU side. This allows to overlap kernel execution and memory transfers but requires a scheduling mechanism that assigns CUDA streams to NMM buffers. GPU nodes can then queue their operations asynchronously in the stream provided by the buffer. Nodes do not need to synchronize asynchronous operations before they forward a buffer to the next node. Synchronization is performed automatically when data is copied back to the CPU and forwarded to a CPU node.

Nodes should be able to operate on different GPUs within the same system which requires different CUDA contexts. Context management is hidden in a scheduling mechanism that provides worker threads for different contexts. If work distribution to locally available GPUs is done automatically, nodes can scale to any number of GPUs. However, all necessary information for allowing arbitrary optimizations, such as specialized memory spaces or synchronizations, have to be provided to the node.

Applications may specify a distributed flow graph with CPU and GPU nodes running on different hosts. A separation of data transfer and data processing allows to connect arbitrary nodes by choosing appropriate transport strategies.

6.3 Three Layer Approach

To hide CUDA related aspects from the application, the framework is implemented in a three layer approach, as depicted in figure 6.7. All three layers can be accessed by the application, but only the flow graph is visible to the application by default.

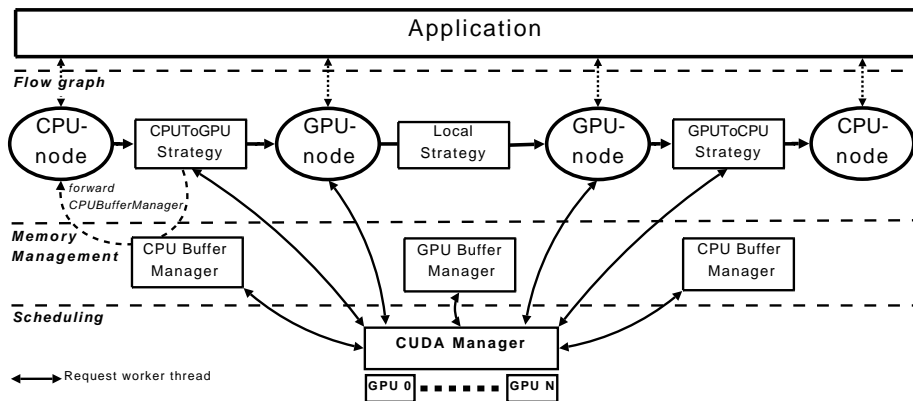


Figure 6.7: CUDA is integrated into NMM using a three layer approach: The actual multimedia flow graph with CPU and GPU nodes, the memory management and the scheduling layer. All layers can be accessed by the application, but only the distributed flow graph is seen by default.

6.3.1 Flow Graph

The first layer consists of the distributed flow graph where CPU and GPU nodes can be accessed and configured through interfaces provided by NMM. The nodes

are connected by suitable transport strategies that perform the memory copies to connect CPU and GPU nodes.

Figure 6.7 shows two CPU and two GPU nodes in a purely local flow graph. The CPU nodes are connected to GPU nodes by a `CPUToGPUStrategy` that copies data from CPU to GPU memory. Analogously, GPU nodes are connected to CPU nodes by a `GPToCPUStrategy`. The `GPToCPUStrategy` ensures that asynchronous operations affecting a particular buffer are finished before a buffer is forwarded to a CPU node. Pointer forwarding by a `LocalStrategy` is sufficient to connect adjacent GPU nodes, since no memory transfers are needed at all.

Since neither the nodes nor the application have to cope with memory transfers to or from memory spaces they might not know, an efficient combination of CPU and GPU nodes is possible. Furthermore, the separation of data transfer and data processing allows to connect GPU nodes without an additional memory transfer.

6.3.2 Memory Management

The second layer is responsible for allocating memory on CPU and GPU side. A `CPUBufferManager` allocates memory in page-locked CPU memory and a `GPUBufferManager` allocates memory on the GPU. These buffer managers follow the concepts of NMM, so they provide a pool of reusable buffers which reduces the costs of expensive memory allocations. Furthermore, buffer managers in NMM are shareable among nodes which allows a `CPUToGPUStrategy` to forward a buffer manager to the predecessor of a GPU node. Since buffer managers provide a standardized interface, the preceding CPU node writes its output directly to page-locked memory without the knowledge of this kind of memory. This allows the `CPUToGPUStrategy` to use DMA transfers to GPU memory. Analogously, the `GPToCPUStrategy` uses a buffer with page-locked memory to copy data back to the CPU and forwards this to the next CPU node.

6.3.3 Scheduling

The third layer manages scheduling related issues. The multimedia data of a buffer is only accessible within the context its memory has been allocated. Therefore, the `CudaManager` at the scheduling layer of the framework has to provide worker threads for all contexts of the available GPUs. Each component of the framework that needs access to the GPU has to request a context specific worker thread from this central `CudaManager` to process a buffer. The `CudaManager` itself knows all locally available GPUs and internally uses a *scheduling strategy* to schedule operations to different GPUs. Additionally, the usage of CUDA streams is controlled at this layer.

6.4 Scheduling Strategies

This section describes the requirements to the scheduling layer of the framework. Mainly there are two things to consider: First, all GPUs within a system have different CUDA contexts. The contexts are mutual exclusive, so GPUs cannot access each others memory, hence, they cannot work cooperatively. The second

issue is the usage of CUDA streams to overlap memory transfers with kernel executions using asynchronous operations.

6.4.1 Context Management

The `CudaManager` is the central component to manage the CUDA capable GPUs in a system. For each GPU, one or more worker threads are provided that are responsible to access resources of a statically assigned context. If multiple worker threads for a single GPU are used, one thread can invoke new operations on the GPU while another thread is blocked and waits for synchronization.

The allocation of memory either in page-locked CPU memory or in GPU memory fixes the corresponding buffer to a CUDA context. The context has to be part of the scheduling information of the buffer and has to be used by the transport strategies and the nodes to request a worker thread from the `CudaManager` in order to process this buffer. Note that the context and thus the corresponding GPU is already fixed when the preceding CPU node writes its output to page-locked CPU memory. At this point, the `CudaManager` has to choose a context for a particular buffer. The context can be selected for each individual buffer such that buffers can be delegated to different GPUs.

6.4.1.1 Stateful and Stateless Nodes

One of the goals stated in section 1.2 is to automatically use all locally available GPUs. Since GPUs cannot share memory, this places some constraints on scheduling: GPU nodes, respectively their kernels, have to be classified into *stateless* and *stateful* nodes. Stateless nodes do not need information on already processed buffers, for example an image denoising node operates on independent frames of a video stream. On the contrary, stateful nodes such as video decoders need to keep information on previously processed frames, optimally in GPU memory.

If there are only stateless nodes in a flow graph, the buffers can be distributed arbitrarily to all GPUs available in the system. This is not possible if there is at least one stateful node in the flow graph. Preceding buffers needed by a stateful kernel might reside on a different GPU and are inaccessible by the kernel. There are two generic solutions for this:

1. All nodes in the flow graph that contains a stateful node have to be implicitly marked as stateful and fixed to operate on the same GPU. This guarantees that a node can keep data in GPU memory needed by subsequent kernel calls. This limits the number of GPUs that can be used, but avoids expensive memory copies between different GPUs.
2. If memory copies between different GPUs are allowed, then only the stateful nodes should be fixed to one GPU.

Seriously, this is a trade-off between a possible performance gain using multiple GPUs and the overhead that comes with additional memory transfers. Currently only stateless nodes are supported. Thus, stateful nodes will not be considered for the rest of this thesis and are left for future work.

6.4.2 Stream Management

As stated in section 6.1 the chosen approach is to use one stream per buffer which is used by the GPU nodes to call their kernel functions. The stream of a buffer has to be part of the scheduling information of a buffer to propagate it through the flow graph. Actually this means that the scheduling strategy creates a new stream whenever a new buffer is created. Since buffers in NMM are managed by a buffer manager and reused as soon as they are not needed anymore, the streams are also reused.

Chapter 7

CUDA-NMM Framework

To integrate CUDA the concepts of NMM are reused whenever possible. This chapter presents the developed extensions and modifications that were necessary to integrate CUDA support in NMM according to the design considerations presented in chapter 6.

7.1 Buffer Processing using CPU and GPU Nodes

The chosen approach is to use a single CUDA stream for each buffer (Section 6.1) where the components along the flow graph queue their CUDA specific operations asynchronously. This implies that resources such as additional memory can not be released immediately after an asynchronous CUDA operation has been called. They have to be stored until synchronization of the stream is ensured. A *CUDA-NMM Stream* wrapper class is provided where buffers can be registered for later release.

Figure 7.1 shows the steps for processing a buffer using CPU and GPU nodes. B_{CPU} indicates that the buffer contains memory in page-locked CPU

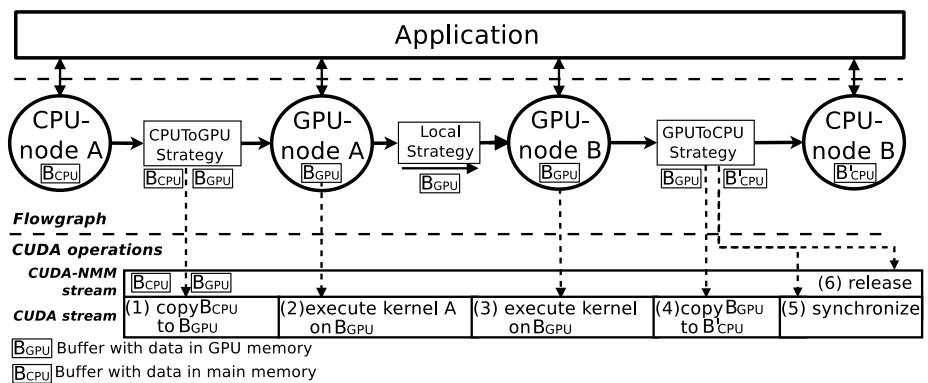


Figure 7.1: This figure shows buffer processing using a combination of CPU and GPU-nodes.

memory whereas B_{GPU} indicates that the buffer contains memory on the GPU. Since a buffer manager for page-locked CPU memory is forwarded to the CPU node A , the output of this node is automatically written to a buffer B_{CPU} . The following steps are performed to process a single buffer:

1. When the `CPUtoGPUStrategy` receives a buffer B_{CPU} it requests a suitable buffer B_{GPU} from a buffer manager and initiates an asynchronous copy from CPU to GPU memory. Since the buffer B_{CPU} cannot be released immediately, it has to be registered with the CUDA-NMM stream that stores the buffer until synchronization. Forwarding the buffer B_{GPU} to the next node (without synchronizing the stream) completes this step.
2. B_{GPU} is the input of GPU node A . The node uses the stream of the buffer to invoke its kernel functions on this buffer asynchronously. Finally, it forwards the buffer to the next node. A `LocalStrategy` is sufficient to connect two GPU nodes.
3. Analogously to GPU node A , GPU node B processes the buffer B_{GPU} and forwards it to the next node.
4. When the `GPUtoCPUStrategy` receives the buffer B_{GPU} , it requests a buffer B'_{CPU} and copies data from GPU to CPU memory.
5. The transport strategy synchronizes the stream to ensure that all operations on this buffer are finished.
6. The transport strategy releases all resources that have been registered in this stream. Finally, the buffer is forwarded to the next CPU node B .

7.2 Hide different Memory Spaces

The buffers B_{CPU} and B_{GPU} described in the previous section are both represented by a `CudaBuffer` class. A `CudaBuffer` essentially behaves like an ordinary NMM buffer, but contains memory in either page-locked CPU memory or in linear GPU memory.

Context specific `CPUBufferManager` and `GPUBufferManager` manage pools of reusable `CudaBuffers` and allocate memory when new `CudaBuffers` are requested. Compared to pageable memory, allocations in page-locked memory are slower by almost a factor of 100, so reusing these buffers is a valuable optimization. The context specific buffer managers are composed to a single `CudaCompositeBufferManager` using a Composite design pattern. This composition presents itself as a single NMM buffer manager to the nodes, but provides an extended interface to explicitly request `CudaBuffers` of a specific context. The `CudaCompositeBufferManager` is shared among all local GPU nodes and the `CPUtoGPUStrategy` forwards this buffer manager to the preceding CPU node A . An extract of this buffer manager can be found figure 7.2.

7.3 GPU Nodes

A `GenericCudaNode` base class provides the necessary abstractions for a node to process `CudaBuffers` on the GPU. Generic CUDA subclasses exist for the

CudaCompositeBufferManager
- cpu_buffer_manager : vector<CudaHostBufferManager*>
- gpu_buffer_manager : vector<CudaDeviceBufferManager*>
+ requestBuffer(size : size_t) : CudaBuffer*
+ requestHostBuffer(size : size_t, context : CudaContextWrapper*) : CudaBuffer*
+ requestDeviceBuffer(size : size_t, context : CudaContextWrapper*) : CudaBuffer*

Figure 7.2: The `CudaCompositeBufferManager` as a composition of context specific buffer managers for page-locked CPU as well as linear GPU memory.

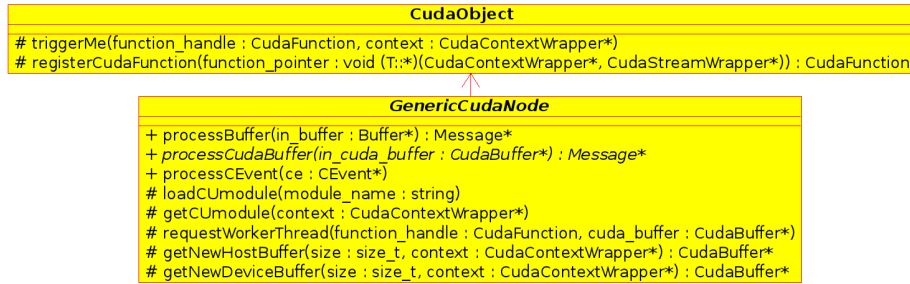


Figure 7.3: The `GenericCudaNode`, which is derived from a `CudaObject` class, provides the base abstractions to process `CudaBuffers`.

various types of nodes in NMM (e.g., `GenericCudaFilterNode`, ...). GPU nodes have to be derived from one of these generic base classes depending on the purpose of the node. The central component of a GPU node is the `processCudaBuffer()` function that gets a `CudaBuffer` as input and returns a `CudaBuffer` after the kernel functions have been invoked. Both, the input and the output buffer contain multimedia data in linear GPU memory.

Since nodes in NMM run in their own application thread, this thread can not be used to call the kernel functions that process the data of the incoming `CudaBuffer`. Instead, a GPU node provides arbitrary many `CudaFunctions` that implement the kernel setup and execution using function calls to the CUDA Driver API [39]. Since the `CudaFunctions` are executed within the context of the current buffer, additional resources such as constant memory or texture references can be allocated here and used by the kernel function. Figure 7.3 shows the most important methods of the `GenericCudaNode` base class. The `processBuffer()` method is only a wrapper and does not have to be implemented by the node.

The functions executed by a worker thread have the following signature: `void cudaFunction(CudaContextWrapper*, CudaStreamWrapper*)`. They have to be registered during the initialization of the node by `registerCudaFunction()`. This returns a handle to the `CudaFunction` which is used to request a worker thread that executes this function.

If the input and output data of a node differ in size, the GPU node can request a new `CudaBuffer` to store the results. The input and output `CudaBuffers` have to exist within the same context and can be requested by `getNewHostBuffer()` for page-locked CPU memory and `getNewDeviceBuffer()` for linear GPU memory.

The CUDA Driver API requires that the kernel module is loaded explicitly

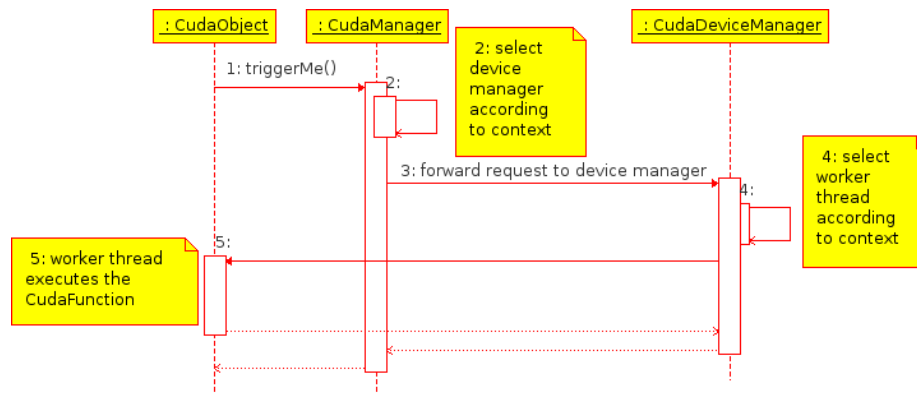


Figure 7.4: Cuda objects specify the CudaFunction handle and the context to request a worker thread.

within a CUDA context. An abstraction is provided that loads a kernel module into all contexts a node is operating on. This is done only once during the initialization of the node. When the node needs to call a kernel it retrieves the module by `getCUModule()` to access the kernel function.

7.4 Scheduling CUDA Worker Threads

Since memory allocation and access through CUDA operations are bound to a specific CPU thread, a `CudaBuffer` always stores the context in which it has been allocated. Each context is wrapped within a `CudaContextWrapper` class.

A unique `CudaManager` initializes CUDA and creates a *device manager* for each locally available GPU. The `CudaManager` is realized as a *singleton* so that only one instance exists. For each GPU at least one worker thread is created and assigned to a CUDA context, but a scheduling strategy can define multiple worker threads per GPU.

The GPU nodes in NMM can request a worker thread for a specific `CudaBuffer` by `requestWorkerThread(CudaFunction, CudaBuffer*)`. The worker thread is then requested according to the context specified by the `CudaBuffer`. If no `CudaBuffer` can be provided to request a worker thread, the `triggerMe()` function can be used which takes a `CudaContextWrapper` as argument instead of a `CudaBuffer`. Actually, the `requestWorkerThread()` method is just a wrapper for the `triggerMe()` function.

Figure 7.4 shows schematically how a context specific worker thread is requested. A call to `triggerMe()` (or `requestWorkerThread()`) forwards the request to the `CudaManager` (1). This, in turn, chooses the `CudaDeviceManager` according to the context (2) and forwards the request to this GPU specific manager (3). The `CudaDeviceManager` knows all worker threads for the specific GPU (4) and takes this worker thread to call the function specified by the `CudaFunction` handle (5). This is blocking the GPU node until the function has been executed.

By this means, the `CudaManager` takes care that operations are performed in the right context and enables for CPU-side multithreading. The nodes only

need to specify the `CudaBuffer` and do not care about context management at all.

The implemented strategy only provides one worker thread per GPU. Multiple worker threads could reduce latencies when a CPU thread is blocked by a synchronization. But it turned out that quickly switching the CUDA context causes a significant overhead on the GPU.

7.5 Data Transfers between Nodes

The transport strategies within the edges of the flow graph connect the nodes and perform the data transfers between CPU and GPU memory. This allows to keep the data in GPU memory as long as possible because no memory copy between adjacent GPU nodes is required.

The `CPUToGPUstrategy` connects CPU to GPU nodes and configures its CPU node to use a `CudaCompositeBufferManager` which instructs the CPU node to write its output to the page-locked memory of a `CudaBuffer` B_{CPU} . The transport strategy can then copy data from B_{CPU} to an adequate buffer B_{GPU} using an asynchronous DMA transfer.

In principle the `GPToCPUstrategy` works analogously and copies data from GPU memory to a `CudaBuffer` B'_{CPU} (see figure 7.1). Additionally it has to ensure that all operations on this buffer are finished before it can be forwarded to the next CPU node. The first approach was to invoke the download operation asynchronously and then synchronize the stream of the `CudaBuffer` which ensures that the results have been copied back to CPU memory. It turned out that this is inefficient if the preceding GPU node operates on two GPUs. Whereas the GPU node calls kernels on both GPUs which potentially doubles performance, the `GPToCPUstrategy` downloads the results separately. This blocks the CPU thread, waiting for the worker thread to synchronize. As a result, the synchronizations in the `GPToCPUstrategy` become the bottleneck of the whole flow graph.

Much better results could be achieved using an asynchronous approach. If the `GPToCPUstrategyAsync` receives a buffer, it issues the memory transfer asynchronously and caches the buffers in an internal queue. Instead of an explicit synchronization, the streams are queried for completion and buffers are forwarded when their stream is worked off. Nevertheless, the order of buffers and events is preserved. Synchronization is only necessary if the internal queue is full. Unfortunately, no stable version could be provided, but benchmarks in chapter 9 will show the advantage of this approach.

A simple pointer forwarding is sufficient to connect adjacent GPU nodes because no memory transfers are necessary. Thus, the `LocalStrategy` provided by NMM could be reused without any modifications.

There is no transport strategy to copy data between different GPUs because this is not supported by CUDA. If this is necessary, data has to be copied to CPU memory first.

7.6 Asynchronous CUDA Operations

Whenever a buffer manager creates a new `CudaBuffer` it asks the scheduling strategy for a new `CudaStreamWrapper`. Since each buffer should have its own stream, the strategy creates a new one for every buffer. The stream is then proposed in the flow graph as part of the scheduling information of the `CudaBuffer`.

Ressources such as memory allocated by the nodes or `CudaBuffers` that are used in asynchronous operations can be registered in the `CudaStreamWrapper`. If necessary, the nodes are able to synchronize the stream between consecutive kernel calls, but they do not have to synchronize when forwarding a `CudaBuffer` to the next node. Synchronization is ensured by the `GPToCPUStrategy` that also releases registered resources.

When a GPU node requests a worker thread with `requestWorkerThread()`, the stream is provided as argument to the specified `CudaFunction`. In the current implementation, this corresponds to the stream provided by the `CudaBuffer`. However, the scheduling strategy has the ability to change this behavior and can explicitly specify a stream that should be used by this node. In principle, this allows to implement all 3 approaches presented in section 6.1.

7.7 Minimize Overhead of Memory Transfers

Since a `CudaCompositeBufferManager` is forwarded to the preceding CPU node, page-locked memory can be used without any additional overhead by memory copies on CPU side. Together with the support for asynchronous operations in different stream, the GPU's processing power can be exploited to full extend while memory copies are performed. Separating buffer transmission and buffer processing enables to efficiently combine GPU nodes while data resides in GPU memory as long as possible.

7.8 Multiple GPUs within a System

The context of a `CudaBuffer` is determined when a CPU node writes its output to the page-locked memory of a `CudaBuffer` B_{CPU} . Forwarding a `CudaCompositeBufferManager` instead of a context specific `CPUBufferManager` to the preceding CPU node, enables to select a different context whenever the CPU node requests a new output buffer. The `CudaCompositeBufferManager` then asks the scheduling strategy of the `CudaManager` which context to use for a specific buffer. The context assigns the `CudaBuffer` to a particular GPU. Currently, a simple round-robin selection is implemented which distributes `CudaBuffers` equally to all available GPUs. This strategy is only applicable if all nodes of the flow graph are stateless (Section 6.4.1.1).

As long as there are no dependencies between the buffers, a node does not have to care on which GPU it is operating on. A suitable worker thread is requested according to the context specified by the `CudaBuffer`. Consequently, all CUDA operations performed by the GPU node are automatically directed to the available GPUs.

7.9 Use GPUs of Remote Systems

Using nodes on a remote host is an inherent feature of the distributed flow graph of NMM. Applications can specify GPU nodes to run on a remote host with a CUDA capable GPU. GPU nodes provide an `ICudaManager` interface that allows to specify the size and the number of `CudaBuffers` preallocated by the `CudaCompositeBufferManager`. Applications can also request a list of available GPUs and specify a set of GPUs that should be used for processing by the CUDA-NMM framework. In order to configure the scheduling strategy of a host, the `ICudaManager` provides a `setCudaSchedulingStrategy()` method which takes the name of the strategy as string argument. A factory object is then responsible to create this strategy on a (possibly remote) host and configures the `CudaManager` accordingly. The configuration specified through the `ICudaManager` interface is shared among all nodes running on a particular host. It only has to be performed once for each host before the initialization state of the GPU nodes.

However, the direct connection of two GPU nodes over network without CPU nodes in between is currently not supported. This requires the concept of pipelined parallel binding [47] and is left for future work.

7.10 Shareable CudaBuffers

If a GPU node forwards a `CudaBuffer` to multiple GPU nodes, the buffer is *shared* and can be accessed by multiple GPU nodes in different branches of the flow graph. If a `CudaBuffer` gets shared, synchronization on the `CudaBuffer`'s stream is performed to avoid unexpected side effects.

If a GPU node needs to modify a `CudaBuffer`, it has to request a writeable instance of this buffer. If this `CudaBuffer` is shared, a new one within the same context is requested and a memory copy in GPU memory is performed. This operation can not be done asynchronously by CUDA, so synchronization is required to create an actual copy of a `CudaBuffer`. Since the resulting copy is independent from the original `CudaBuffer` it gets its own CUDA stream.

7.11 Handling Instream Events

If a node in a flow graph receives an instream `CEvent`, this event can change the configuration of the node which might affect asynchronous buffer processing. Thus, a node has to ensure that buffers preceding the `CEvent` are worked off, before the event can be processed. In particular, a GPU node that calls CUDA operations asynchronously has to synchronize the operations related to the preceding `CudaBuffers`. This can be achieved by memorizing the streams of those `CudaBuffers` and synchronize them when an instream event is received. The problem is, that the `CudaBuffers` and their streams are reused in the flow graph, so a node does not know which streams it has to synchronize.

To solve this, a `CudaNMMEventManager` is provided that keeps track of the `CudaBuffers` that are active in the flow graph. `CudaBuffers` requested from a buffer manager are marked as *active* until they return to their buffer manager. If a `CudaBuffer` is processed by a GPU node, this is automatically memorized. That is, the `CudaNMMEventManager` knows which node has pending operations in

which streams. When a node receives an instream `CEvent`, it just has to instruct the `CudaNMEventManager` to synchronize all `CudaBuffers` that are still active in the flow graph and already passed that particular node.

Chapter 8

CUDA Plugins for NMM

The previous chapters presented the design and the implementation details of the CUDA framework for NMM. Since it is designed in a three layer approach and common NMM concepts are used, the development of nodes and applications using CUDA is very similar to the tutorials found in the NMM documentation [11].

This chapter will highlight specific extensions that have to be used to access and process media data on the GPU within the nodes of a flow graph. A tutorial on node development is found in section 8.1. Application development is described in section 8.2. I also implemented some proof-of-concept GPU nodes that should serve as templates for further nodes. These are presented at the end of this chapter.

8.1 Implementing GPU Nodes

Usually a GPU node consists of two parts: The actual node that is derived from the `GenericCudaNode` base class and the kernel functions executed on the GPU.

8.1.1 Node Development

Each NMM node that is derived from the `GenericCudaNode` class is able to process buffers on the GPU. The base class provides methods to interact with the CUDA framework and to call kernel functions. Since the framework uses the CUDA Driver API, no function calls to the CUDA Runtime API are possible in a GPU node.

8.1.1.1 Initialization

Beside the usual setup of supported input and output formats of the node, a GPU node needs to load its kernel module. The kernel module contains all kernel functions that are accessible by the node. A module is typically available in only one context, but CUDA nodes should operate on multiple contexts and GPUs. The `loadCUModule()` ensures that the module is available in every context the node is operating in. When processing buffers, a handle to the context specific module is returned by `getCUModule()`.

All methods of the node that invoke a kernel have to be executed by a context specific worker thread. Hence, node developers have to implement all CUDA functionality in `CudaFunctions` with the following signature:

```
void cudaFunction(CudaContextWrapper*, CudaStreamWrapper*)
```

These methods have to be registered during initialization. This returns a handle which is required to execute the `CudaFunction` by a worker thread.

Listing 8.1 shows the necessary steps during the initialization of a CUDA node.

Listing 8.1: CUDA node initialization

```
1 Result IdNodeCuda::doInit() {
2     /// Initialize CUDA before any other call to the CUDA framework
3         of NMM
4     GenericCudaNode::doInit();
5
6     /// Load the CUDA kernel module for all available contexts.
7     loadCUmodule("kernel_IdNodeCuda.cubin");
8
9     /// Register the CudaFunction of this node
10    cuda_function_handle = registerCudaFunction(&IdNodeCuda::
11        executeKernel);
12
13    /// Setup supported formats
14    ...
15    return SUCCESS;
16 }
```

8.1.1.2 Buffer Processing

The central part of each NMM node is the `processBuffer()` method which usually implements algorithms to process multimedia data. For GPU nodes this has been replaced by a `processCudaBuffer()` method with `CudaBuffers` as input and output.

The input `CudaBuffer` already contains data in linear GPU memory, but since the `processCudaBuffer()` method is executed by a node-specific thread, this thread cannot be used to access the data or to call kernel functions. By a call to `requestWorkerThread()`, the `processCudaBuffer()` method requests a context-specific worker thread and specifies a handle to the `CudaFunction` to process a `CudaBuffer`. Listing 8.2 shows an example.

Listing 8.2: A sample `processCudaBuffer()` method

```
1 Message* IdNodeCuda::processCudaBuffer(CudaBuffer *in_buffer) {
2     if (!in_buffer) {
3         return 0;
4     }
5
6     m_incoming_buffer = in_buffer->getWriteableInstance();
7
8     /// call the specified CudaFunction with the context of the
9         current CudaBuffer
10    requestWorkerThread(cuda_function_handle, m_incoming_buffer);
11
12    return m_incoming_buffer;
13 }
```

A call to `requestWorkerThread()` blocks the application thread until the `CudaFunction` is executed, but it does not ensure synchronization. This means, if all operations within the `CudaFunction` are performed asynchronously, the `requestWorkerThread()` function returns almost immediately.

A simple `CudaFunction` with one kernel invocation can be found in listing 8.3. The `CudaFunction` is executed by a context specific worker thread, so it has access to the resources within the `CudaBuffer`'s context.

Listing 8.3: A sample `CudaFunction` with one kernel call

```

1 void IdNodeCuda::executeKernel(CudaContextWrapper* cu_context,
2   CudaStreamWrapper* cu_stream)
3 {
4   CUdeviceptr tmp_deviceptr = m_incoming_buffer->getDevicePtr();
5   int length = m_incoming_buffer->getUsed();
6   int blockDim = 512;
7   int gridWidth = (length/blockWidth)+1;
8   int gridHeight = 1;
9   int offset = 0;
10
11   /// Get the CModule for the current context and the kernel
12   function
13   cuModuleGetFunction(&m_internal_kernel, getCModule(cu_context),
14     "kernel");
15
16   /// Begin > Setup execution configuration
17   cuFuncSetBlockShape(m_internal_kernel, blockDim, 1, 1);
18
19   /// first kernel argument: pointer to GPU memory
20   cuParamSetv(m_internal_kernel, offset, &tmp_deviceptr, sizeof(
21     void*));
22   offset += __alignof(void*);
23
24   /// second kernel argument: length of the input array
25   cuParamSeti(m_internal_kernel, offset, length);
26   offset += __alignof(int);
27
28   cuParamSetSize(m_internal_kernel, offset);
29   /// End < Setup execution configuration
30
31   /// Launch kernel asynchronously in the given stream
32   cuLaunchGridAsync(m_internal_kernel, gridWidth, gridHeight,
33     cu_stream->getStream());
34 }

```

Currently only page-locked CPU memory and linear GPU memory are supported by buffer managers. Context specific `CudaBuffers` containing these types of memory can be requested with `getNewDeviceBuffer()` and `getNewHostBuffer()` inside the `processCudaBuffer()` method. Resources that are not managed by a buffer manager have to be allocated inside the `CudaFunction`. Recall from section 7.6 that all resources that are requested by the node and used in asynchronous operations have to be registered in the `CudaStreamWrapper`. These resources are automatically released by the `GPToCPUStrategy`. Not all types of memory are implemented yet, so release methods might have to be added to the `CudaStreamWrapper` class.

8.1.1.3 Instream CEvents

Instream `CEvents` can potentially change the node's configuration. As already stated in section 7.11 this requires a synchronization of the CUDA streams that have been used by the node for asynchronous kernel invocations. This is achieved by a simple `syncBeforeEvent()` call to the `CudaNMMEEventManager` before processing an event. This is also the default behavior implemented in the `GenericCudaNode`.

Listing 8.4: Synchronization on instream events

```
1 Result GenericCudaNode::processCEvent(CEvent* ce) {
2     SCudaManager::getInstance().getEventManager()->syncBeforeEvent(
3         this);
4     return SUCCESS;
5 }
```

8.1.2 Kernel Development

All kernel functions of a node have to be specified within a single `.cu` file. By adding this file to the node's `Makefile.am` in the `KERNEL` section, it is compiled to a binary `.cubin` module.

The kernel function used in the above examples is defined in a `kernel_IdNodeCuda.cu` file.

Listing 8.5: Prototype for a kernel function

```
1 extern "C"
2 __global__ void kernel( char* g_data, int length )
3 {
4     ...
5 }
```

8.2 Application Development

Essentially there are no differences to normal NMM application development without CUDA. Local and remote nodes can be requested from the registry to build a multimedia flow graph.

But currently some limitations exist:

- GPU nodes running on different hosts cannot be directly connected without a CPU node in between. This would require a pipelined parallel binding [47] which has to be implemented.
- The `CPUToGPUStrategy` and `GPToCPUStrategy` have to be set up manually by the application.

The `ICudaManager` interface provided by the GPU nodes can be used to configure the number and the size of the `CudaBuffers` in page-locked CPU and linear GPU memory that are allocated by the respective buffer managers. These settings apply equally to all contexts. On systems with multiple GPUs the application can select the GPUs that should be used and globally limit processing

to one GPU as a workaround for stateful nodes. The scheduling strategy can also be configured through the `ICudaManager` interface by specifying the name of the strategy. These settings are shared among all nodes of a specific host, so the configuration has to be performed only once for each host.

8.3 Implemented GPU Nodes

To substantiate the operability of the presented framework, I have implemented four proof-of-concept NMM nodes that are considered as templates for further GPU nodes.

8.3.1 IdNodeCuda

The `IdNodeCuda` has already been presented in the previous sections as example for the most basic CUDA enabled NMM node. It launches a single kernel function on the input buffer and outputs the identity function applied to each byte of the media data.

8.3.2 DenoiseNodeCuda

This GPU node takes a `raw/RGB` video stream as input and outputs a denoised video stream of the same format.

Kernels for different image denoising algorithms are shipped with the Nvidia CUDA SDK [40] and are reused in this node. This node is stateless and can automatically use multiple GPUs because the kernel function operates on each frame separately.

This node also demonstrates the usage of the GPU's texturing hardware that is accessible from CUDA. Incoming `CudaBuffers` provide data in linear GPU memory while the kernels used here expect data in 2D textures. The node has to copy data from linear memory to a read-only 2D CUDA array using an asynchronous copy. This CUDA array can then be bound to a 2D texture. Although memory copies in GPU memory are usually very fast, proper support for CUDA arrays is preferable. This will be discussed in section 10.3.1.

8.3.3 OverlayNodeCuda

This node has two input jacks and one output jack. The first input jack is called *default* and takes a `raw/RGB` video, the other input jack called *image* takes a PNG image. The *default* output jack outputs the overlay of the video stream with the logo. Both input streams are expected in GPU memory.

The kernel provided by this node does hardly perform any computations so that the memory bandwidth becomes the bottleneck of this node. However, as stated in section 6.4.1.1 this node is stateful because the image and the video frames are potentially on different GPUs. Since the logo is sent only once, a workaround is possible here: When the node receives the logo image, it synchronizes the memory transfers (i.e. the stream of the buffer) and copies the image to all other contexts. This makes the node stateless, hence, the node can operate on all locally available GPUs. In such cases the overhead of the memory transfer between two GPUs is negligible compared to the performance

gain when using multiple GPUs. This node demonstrates how to make a stateful node stateless in order to use multiple GPUs.

8.3.4 SimpleRayTracer

This GPU node implements a very simple ray tracer based on the source code found at [34]. The demo program shows two animated bouncing balls. It has been integrated in a GPU node that acts as a source node and generates images in OpenGL Pixel Buffer Objects (PBO) that are accessible through the *CUDA OpenGL Interoperability API*. The OpenGL buffers can directly be displayed using an OpenGL window. Using NMM it is further possible to bind these PBOs to the linear GPU memory of a `CudaBuffer` and forward them to another node. Connecting the `SimpleRayTracer` node to the non-OpenGL `XDisplayNode` provided by NMM it is possible to display the images generated by the `SimpleRayTracer` node. However, implementing an OpenGL display node would be a possible extension to the CUDA-NMM framework. In doing so, data could be kept in GPU memory.

Chapter 9

Performance Measurement

Since using a middleware usually imposes an overhead, some benchmarks have to be performed that compare a simple flow graph in NMM with a plain CUDA program.

9.1 Benchmark Setup

The following setup has been used for the benchmarks:

- CPU: Intel Core2 Duo, 3.33 GHz
- 4 GB RAM
- 2 Nvidia GeForce 9600 GT
 - 512 MB GDDR3 RAM
 - Compute Capability 1.1
 - PCIe 2.0
- Ubuntu 8.10 (64 Bit)
- gcc 4.1.3
- Nvidia graphics driver 177.82
- Nvidia CUDA Toolkit 2.0
- Nvidia CUDA SDK 2.0

9.2 Test Cases

The NMM flow graph consists of a source and a sink node on CPU side and a single GPU node in between which calls a kernel function on the buffers. The source node is a `RawNode` which produces random data of a specified size, the sink node is a `DevNullNode` which just releases the buffers and does not perform any computations. The number of buffers per second that pass the sink node is counted to measure the throughput.

This throughput is compared to a single-threaded reference implementation using CUDA without any abstractions. The buffers are represented by `char` arrays and allocated in page-locked CPU memory. A memory copy to the GPU is performed asynchronously, followed by a single kernel call representing the NMM node. Afterwards, the buffer is copied back to the CPU and synchronization is ensured. As in the NMM flow graph, the number of buffers is measured that are processed within a fixed period of time. In both cases, the allocation of CPU and GPU memory are not considered because NMM reuses buffers to avoid expensive memory allocations.

The *Cuda Visual Profiler* [43] is used to measure the execution time that is spent by the CPU and the GPU to complete a kernel call in the CUDA-NMM framework. For some reason the Profiler did not provide the time spent for the memory copies, so only CPU and GPU times are considered in the benchmarks. The CPU time is the time it takes for the CPU to call a kernel function, since operations are performed asynchronously it is expected to be very low. The GPU time describes the time that the GPU needs to complete a kernel call.

9.3 Performance Measurement

The benchmarks have been performed using the stable version of the `GPToCPUStrategy` as described in section 7.5. Results from NMM using one and two GPUs are compared to the reference implementation. All benchmarks are using a stateless kernel.

The first benchmark uses an empty kernel representing the identity function. The kernel launches a thread for every item of the `char` array that contains the multimedia data. The results are presented in table 9.1. Overlapping of memory transfers with kernel executions of different streams is strongly limited since the kernel does not perform any computation. This results in significant overhead up to buffer sizes of 1500 KB because the kernel execution only takes up negligible time compared to the memory copy. As discussed in section 7.5 the synchronizations in the `GPToCPUStrategy` prevent an efficient use of multiple GPUs; in fact, it also adds some additional overhead. Since the framework allows to perform all kernel invocations asynchronously, the CPU hardly spends any time for kernel calls.

Buffer size	Reference in [Buf/s]	NMM:1 GPU in [Buf/s]	NMM:2 GPUs in [Buf/s]	GPU Time in [μ s]	CPU Time in [μ s]
50 KB	12354	7711 (62.4 %)	7576 (61.3 %)	3.27	2.5
100 KB	9786	6705 (68.5 %)	6364 (65 %)	5.57	2.49
200 KB	6514	5048 (77.5 %)	4886 (75 %)	10.18	2.76
300 KB	5091	4231 (83.1 %)	4099 (80.5 %)	14.8	2.68
400 KB	3841	2810 (73.2 %)	3454 (89.9 %)	19.47	2.54
500 KB	3578	3180 (88.9 %)	3103 (86.7 %)	24.23	2.6
1000 KB	2004	1891 (94.4 %)	1891 (94.4 %)	47.37	2.67
1500 KB	1358	1356 (99.9 %)	1350 (99.4 %)	70.53	2.75
2000 KB	1051	1087 (103.4 %)	1008 (95.9 %)	93.67	2.67
3000 KB	711	750 (105.5 %)	735 (103.4 %)	140	2.7

Table 9.1: Performance of the CUDA-NMM integration versus a single threaded reference implementation. In this benchmark an empty kernel is used which computes the identity function. Throughput is measured in buffers per second [Buf/s].

The second benchmark uses a kernel that represents a brightness modifier for video streams in the YV12 format. The number of pixels that store the brightness information in a YV12 video frame are increased by a certain value. No optimizations in kernel code are performed so that the kernel takes up a reasonable amount of time. The benchmark results can be found in table 9.2. The automatic usage of streams within the CUDA-NMM framework combined with a multithreaded application exceeds the results achieved by the reference implementation for almost all buffer sizes. But as in the previous benchmark, there is no performance gain by using 2 GPUs.

Buffer size	Reference in [Buf/s]	NMM:1 GPU in [Buf/s]	NMM:2 GPUs in [Buf/s]	GPU Time in [μ s]	CPU Time in [μ s]
50 KB	6593	6182 (93.8 %)	5978 (90.7 %)	71.37	2.69
100 KB	4064	3874 (95.3 %)	3979 (97.9 %)	145.4	2.66
200 KB	2290	2365 (103.3 %)	2246 (98.1 %)	296.19	2.58
300 KB	1596	1643 (102.9 %)	1678 (105.1 %)	443.08	2.58
400 KB	1225	1287 (105.1 %)	1291 (105.4 %)	590.43	2.59
500 KB	993	1060 (106.7 %)	1088 (109.6 %)	739.42	2.56
1000 KB	512	551 (107.6 %)	549 (107.2 %)	1483.91	2.55
1500 KB	342	372 (108.8 %)	379 (101.9 %)	2211.88	2.91
2000 KB	259	282 (108.9 %)	285 (110 %)	2956.91	2.67
3000 KB	172	185 (107.6 %)	191 (111 %)	4427.63	2.71

Table 9.2: Performance of the CUDA-NMM integration versus a single threaded reference implementation. In this benchmark a kernel computing an YV12 brightness algorithm is used. Throughput is measured in buffers per second [Buf/s].

The last benchmark compares the asynchronous `GPToCPUStrategyAsync` with the stable `GPToCPUStrategy` and measures the throughput of the YV12 brightness kernel. When using this transport strategy the results for all buffer sizes exceed the values achieved by the normal `GPToCPUStrategy` even for a single GPU. This transport strategy also scales very well with multiple GPUs and achieves almost a twofold speed-up for most buffer sizes.

Buffer size	GPToCPUStrategy		GPToCPUStrategyAsync	
	NMM:1 GPU [Buf/s]	NMM:1 GPU [Buf/s]	NMM:1 GPU [Buf/s]	NMM:2 GPUs [Buf/s]
50 KB	6182	6771 (109.5 %)	10762 (174.1 %)	
100 KB	3874	4139 (106.8 %)	7477 (193 %)	
500 KB	1060	1175 (110.8 %)	2317 (218.6 %)	
1000 KB	551	585 (106.2 %)	1173 (212.9 %)	
1500 KB	372	392 (105.4 %)	791 (212.6 %)	
2000 KB	282	297 (105.3 %)	592 (209.9 %)	
3000 KB	185	203 (109.7 %)	407 (220 %)	

Table 9.3: This benchmark demonstrates the performance gain using an asynchronous `GPToCPUStrategyAsync`. The kernel computes an YV12 brightness function. Throughput is measured in buffers per second [Buf/s]. Percentages are compared to the `GPToCPUStrategy` with 1 GPU.

Chapter 10

Results

The previous chapters presented a framework built on top of the NMM middleware to transparently use CUDA enabled GPUs in a network. Therefore, CUDA kernels that implement processing steps on the GPU have been integrated into the nodes of a distributed flow graph. This was done under the aspect of an efficient combination of nodes using the GPU and nodes using the CPU for media processing. The key concept was to treat the GPU similar to a distributed system by integrating data movements in the edges of a flow graph.

This chapter summarizes the steps that were necessary to integrate CUDA in NMM. Section 10.2 presents the results of this thesis with respect to the goals stated in section 1.2. The last section discusses the next steps to be done.

10.1 Summary

First, the overall goal to integrate processing on the GPU into NMM has been motivated in chapter 1. The massively parallel architecture of recent GPUs is ideally suited for multimedia applications like video processing. However, GPUs have their own memory space and CUDA places special requirements on scheduling which makes it difficult to develop reusable components that can be efficiently combined in an application. These issues can be solved by integrating processing steps into the nodes of a flow graph while data movements between different memory spaces are hidden in the edges between the nodes. Furthermore, using a distributed middleware almost automatically provides the possibility to use GPUs of remote systems for media processing.

Since this thesis deals with parallel processing on GPUs, an overview on parallel computing with the focus on general purpose computing on GPUs has been presented in chapter 2. GPUs are designed in favor of high floating point performance on many parallel tasks using a stream processing approach where a series of highly parallel kernel functions is applied to a large data set. This stream processing model of the GPU fits very well to the concept of a flow graph where different nodes execute functions on the data that passes through the flow graph.

The advantages of CUDA over other GPGPU programming models have been presented in chapter 3. In CUDA the GPU is represented as a co-processor that is well suited for compute intensive algorithms that can be expressed as

data-parallel operations. The SIMT threading model combined with a complex shared memory architecture (see section 3.2) offers a new level of flexibility to program the GPU for non-graphics computations. Features and requirements that are relevant for an efficient integration of CUDA have been analyzed in section 3.3.

The related work presented in chapter 4 contained an overview on distributed middleware solutions that integrated CUDA for grid computing. Furthermore, GPGPU programming frameworks despite CUDA and CUDA accelerated multimedia libraries have been presented. The algorithms implemented by these libraries can be reused in the CUDA-NMM framework.

Chapter 5 highlighted concepts of NMM that have been reused or extended to integrate CUDA support. The concept of a distributed flow graph allows to encapsulate algorithms for multimedia processing into modular nodes. The nodes hide implementation details which allows to integrate CUDA algorithms transparently into applications. Furthermore, nodes provide unified interfaces that enable the application to control nodes on remote hosts. Transport strategies in the edges of a flow graph provide a strict separation of communication and processing. Due to the extensibility of NMM, new transport strategies can be easily integrated.

Chapter 6 described the overall design of the framework and started with an evaluation of different approaches that allow an efficient integration of CUDA in NMM (see section 6.1). A three layer approach (see section 6.3) of the framework has been chosen to hide CUDA related aspects from the application. Furthermore, scheduling related issues have been delegated to a lower layer of the framework in order to simplify the development of new nodes.

Based on these considerations, the implementation of the framework has been described in chapter 7. The framework allows to combine processing on the CPU and the GPU using asynchronous CUDA operations. GPU nodes can use CUDA streams to call their kernel functions asynchronously while necessary synchronizations are performed automatically. A `CudaManager` realizes a scheduling strategy transparently for node and application developers. A separation of data processing and data transfer between different memory spaces has been realized by developing new transport strategies.

Chapter 8 described node and application development using the CUDA-NMM framework and showed that developing new GPU nodes only requires some easily understood abstractions. Developing applications using GPU nodes does not require any new concepts at all, but due to lack of time there exist limitations when connecting GPU nodes of different hosts (see section 8.2). The GPU nodes developed in the scope of this thesis have been shortly presented in section 8.3. They can be found in the source files of the CUDA-NMM framework and act as templates for further GPU nodes.

The performance measurements presented in chapter 9 showed that using a middleware does not add any overhead compared to a plain CUDA program for most use cases. Due to the efficient usage of the features provided by CUDA (Section 3.3) and the multithreaded design of the framework, applications can even improve their performance.

10.2 Achieved Goals

This section summarizes how the goals stated in section 1.2 could be achieved by the CUDA-NMM framework.

1. The first goal was to analyze the features and requirements of CUDA in order to integrate CUDA efficiently in NMM. CUDA enabled GPUs act as data-parallel co-processors that have to be controlled by an application running on the CPU. The overhead through memory transfers can be minimized using asynchronous operations in streams and page-locked memory on CPU side.
2. Existing kernels provided by multimedia libraries can be reused within the CUDA framework. The framework allows this because it is built on top of the CUDA Driver API which includes the functionality of the Runtime API. However, the application logic on CPU side has to be translated to the Driver API if the library is built on top of the Runtime API.
3. Applications built on top of NMM usually specify and configure a flow graph to process multimedia data. Since kernel invocations are encapsulated in GPU nodes, the application does not know whether data is processed by the CPU or the GPU. Furthermore, scheduling related aspects and synchronizations are handled by lower layers of the framework. This fulfills the third goal stated in section 1.2 since applications can seamlessly integrate CUDA support by replacing a node of the flow graph.
4. Using the CUDA-NMM framework enables kernel developers to encapsulate kernel functions into GPU nodes. Actually, kernel developers only need to specify the execution configuration of their kernels inside a node and can basically ignore scheduling related issues concerning stream and context management. A separation between data transfer through transport strategies and data processing in the nodes of a flow graph enables to efficiently combine adjacent GPU nodes since no memory transfer is needed at all. The concept of shareable buffer managers allows to configure CPU nodes to use page-locked memory which in turn enables for an efficient use of fast DMA transfers. Asynchronous operations can be used to save processing time on the CPU for other computational tasks. Therefore, an efficient combination of processing on the GPU *and* the CPU is possible, which fulfills the fourth goal.
5. The usage of all locally available GPUs could only be achieved for stateless nodes (see section 6.4.1.1). A strategy manages the distribution of buffers to the GPUs within a single machine. The necessary extensions to support stateful nodes will be discussed in the next section. Admittedly, the performance gain when using multiple GPUs is impeded by the `GPToCPUStrategy`. Providing a stable version of the asynchronous `GPToCPUStrategyAsync` described in section 7.5 would solve this issue. Applications built on top of NMM can specify a remote host that should be used for media processing using CUDA. So the goal to use remote GPUs for media processing could be achieved. The direct connection between GPU nodes of different hosts by a pipeline of transport strategies is cur-

rently not supported, however, placing an identity CPU node in between is sufficient.

6. Benchmarks presented in chapter 9 showed that using a distributed middleware even for a purely local application does not add any overhead for reasonable large problem sizes. Due to the automatic use of the features provided by CUDA, applications can even improve their performance compared to a plain CUDA program.

10.3 Future Work

This section discusses features to be added.

10.3.1 Additional Memory Spaces

Currently `CudaBuffers` only offer native support for linear memory on the GPU which is managed by a `GPUBufferManager`. If a kernel function needs data bound to a 2D texture for example, the node has to manually copy this data to a 2D memory region (see section 8.3.2). This memory copy certainly adds some overhead and is a subject that needs further improvements.

The next logical step is to extend the `CudaBuffer` to support arbitrary memory spaces on the GPU such as multidimensional CUDA arrays which can be bound to texture references. For example, CUDA arrays rely on a rather complex configuration, called *descriptors*. Actually, the descriptors correspond to the NMM concept of *formats* (Section 5.4) for the input and output jacks of a node. Since NMM supports subformats to be added to the formats, it is possible to convert the information contained in the descriptors into such a subformat. Then, only nodes with matching descriptors could be connected. Furthermore, suitable buffer managers have to be added and transport strategies have to be extended because CUDA arrays cannot be allocated and copied in the same way as linear GPU memory.

10.3.2 OpenGL Integration

Since CUDA can use OpenGL *Pixel Buffer Objects (PBO)* to store the output of a kernel function, a possible extension to NMM would be an *OpenGL Display Node* which can be used to directly display the results generated by GPU nodes without the necessity to copy the results back to CPU memory.

When writing the `SimpleRayTracer` node it turned out that PBOs can almost be treated as linear GPU memory. A specialized buffer manager should be a sufficient extension to the CUDA-NMM framework to support PBOs while existing transport strategies can be reused. Since such a node acts as a sink node and no `GPUtoCPUstrategy` is used, it has to implement some functionality of the transport strategy. In particular, it has to ensure synchronization and to release the `CudaBuffers` stored in the stream.

10.3.3 Stateful GPU Nodes

Recall from section 6.4.1.1 that stateful nodes need information about previously processed buffers. As already mentioned, this requires that a stateful GPU node

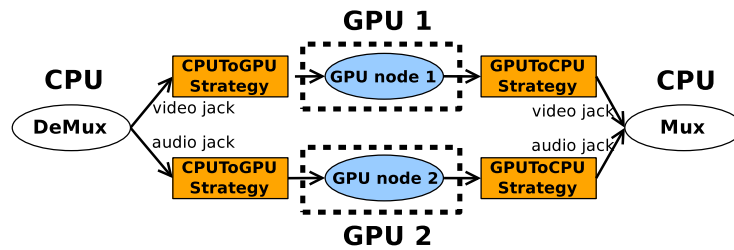


Figure 10.1: Stateful Nodes: (De-)Multiplexing is used to split data into independent video and audio streams. Although the GPU nodes are stateful, both GPUs in a system can be used to process independent data streams.

is limited to operate on a single GPU in order to avoid memory copies between different GPUs. Despite the workaround to limit the number of GPUs used in a system to one, the support for stateful nodes is missing completely. Extensions to the framework are necessary to limit stateful nodes to operate on a single GPU. Therefore, the preceding node has to be configured to use a particular context for the buffers that are forwarded to the stateful GPU node.

Then, a combination of demultiplexer and multiplexer can be used to split, for example, a video stream in its video and audio streams as can be seen in figure 10.1. Although the GPU nodes that process these streams might be stateful, two GPUs within a system can be used to process independent buffers in different branches of the flow graph.

10.3.4 Extensions to Transport Mechanism

The `UserInfo` of the GPU nodes and the transport strategies have to be implemented in order to automatically propose suitable transport strategies to connect GPU and CPU node. Currently, this has to be done manually.

Additionally, a stable version of the asynchronous `GPUToCPUStrategyAsync` has to be provided in order to efficiently copy back the results when using multiple GPUs (see section 7.5).

10.3.5 Pipelined Parallel Binding

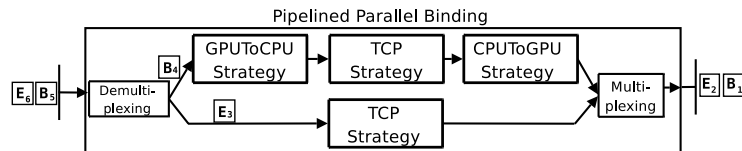


Figure 10.2: Sketch of a pipelined parallel binding to directly connect GPU nodes of different hosts. Buffers are transported by a pipeline of transport strategies while a single `TCPStrategy` is sufficient for event transmission. [47]

To directly connect GPU nodes running on different hosts, a pipeline of transport strategies is required. A `GPUToCPUStrategy` can then copy data to

CPU memory, a `TCPStrategy` can transfer buffers over a network connection and a `CPUToGPUStrategy` on the target host can copy the data to GPU memory. This is currently not possible without a CPU node in between.

Furthermore, NMM offers the concept of a parallel binding of transport strategies. The transport strategies implemented in the CUDA-NMM framework handle both, buffer and event transmission. Although this is no major drawback in the current state of the implementation, a separation of buffer and event transport would remove redundancies in the implementation and increase reusability.

Combining the concepts of a pipeline of transport strategies and parallel binding would realize a *pipelined parallel binding* to directly connect remote GPU nodes as depicted in figure 10.2.

List of Figures

2.1	Comparison of floating point performance of CPU and GPU . . .	8
3.1	Tesla GPU Architecture	12
3.2	Heterogeneous Programming Model of CUDA	14
3.3	CUDA Thread Model	16
3.4	CUDA Memory Hierarchy	17
3.5	CUDA Asynchronous Operations	19
3.6	CUDA Streams	20
3.7	CUDA Events	21
5.1	Client/Server versus Middleware approach	30
5.2	NMM Multimedia Flow Graph	30
5.3	Distributed NMM Flow Graph	32
5.4	Parallel Binding of Transport Strategies	33
5.5	States of NMM Nodes	33
6.1	Overview of an NMM flow graph with GPU nodes	35
6.2	Scheduling Strategy: Using a single CUDA stream	36
6.3	Scheduling Strategy: Assignment of one stream per buffer	37
6.4	Scheduling Strategy: Assignment of one stream per object	37
6.5	Benchmark of CUDA streams on compute capability 1.1	38
6.6	Benchmark of CUDA streams on compute capability 1.0	39
6.7	Design of the CUDA-NMM framework	40
7.1	Processing on CPU and GPU	45
7.2	CudaCompositeBufferManager	47
7.3	GenericCudaNode base class	47
7.4	Requesting a Worker Thread	48
10.1	Stateful Nodes	67
10.2	Pipelined Parallel Binding	67

List of Tables

3.1	Memory Spaces on a Tesla GPU	13
5.1	Different Types of Nodes in NMM	31
9.1	Benchmark: Identity Function	60
9.2	Benchmark: YV12 Brightness Function	61
9.3	Benchmark: Asynchronous Transport Mechanism	61

Bibliography

- [1] Motama GmbH. <http://www.motama.com>.
- [2] Network-Integrated Multimedia Middleware (NMM). open-source version. <http://www.motama.com/nmm.html>.
- [3] NMM Project at Computer Graphics Lab, Saarland University. <http://graphics.cs.uni-sb.de/NMM/>.
- [4] CUDA Coding Contest, 2008-10-16. <http://cudacontest.nvidia.com/>.
- [5] Wikipedia Article on Data Parallelism, 2009-04-05. http://en.wikipedia.org/wiki/Data_parallel.
- [6] Wikipedia Article on Stream Processing, 2009-04-05. http://en.wikipedia.org/wiki/Stream_processing.
- [7] Wikipedia Article on Task Parallelism, 2009-04-05. http://en.wikipedia.org/wiki/Task_parallelism.
- [8] ATI Stream Technology, 2009-04-29. <http://www.amd.com/stream>.
- [9] DGAVCDecNV, 2009-04-29. <http://neuron2.net/dgavcdecnv/dgavcdecnv.html>.
- [10] GPU4Vision, 2009-04-29. <http://www.gpu4vision.org>.
- [11] NMM Online Documentation, 2009-04-29. <http://www.motama.com/nmmdocumentation.html>.
- [12] OpenCL Overview, 2009-04-29. <http://www.khronos.org/opencvl>.
- [13] The OpenGL Shading Language, 2009-04-29. <http://www.opengl.org/documentation/glsl>.
- [14] GPGPU.org Developer Resources, 2009-05-02. <http://www.gpgpu.org>.
- [15] Cyberlink PowerDirector 7, 2009-05-07. http://www.cyberlink.com/products/powerdirector/overview_en_US.html.
- [16] distributed.net, 2009-05-07. <http://www.distributed.net/>.
- [17] Yannick Allusse, Patrick Horain, Ankit Agarwal, and Cindula Saipriyadarshan. GpuCV: An opensource GPU-accelerated framework for image processing and computer vision. In *MM '08: Proceeding of the 16th ACM international conference on Multimedia*, pages 1089–1092, New York, NY, USA, 2008. ACM.

- [18] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.
- [19] Blaise Barney. Introduction to Parallel Computing, 2009-04-29. https://computing.llnl.gov/tutorials/parallel_comp/.
- [20] Michael D. Beynon, Tahsin Kurc, Umit Catalyurek, Chialin Chang, Alan Sussman, and Joel Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Comput.*, 27(11):1457–1478, 2001.
- [21] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM Press.
- [22] Elemental Technologies. Badaboom Media Converter, 2009-04-29. <http://www.badaboomit.com/>.
- [23] Kayvon Fatahalian and Mike Houston. GPUs: A Closer Look. *ACM Queue: Tomorrow's Computing Today*, 6(2):18–28, mar 2008.
- [24] Michael Flynn. Some computer organizations and their effectiveness. *IEEETC:JOURNAL*, 21(9):948–960, 1972.
- [25] James Fung and Steve Mann. OpenVIDIA: Parallel GPU computer vision. In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, pages 849–852, New York, NY, USA, 2005. ACM.
- [26] James Fung and Steve Mann. Using graphics devices in reverse: GPU-based Image Processing and Computer Vision. In *ICME*, pages 9–12. IEEE, 2008.
- [27] G. Giupponi, MJ Harvey, and G. De Fabritiis. The impact of accelerator processors for high-throughput molecular modeling and simulation. *Drug Discovery Today*, 13(23-24):1052–1058, 2008.
- [28] Hartley, Timothy D. R., Catalyurek Umit, Ruiz Antonio, Igual Francisco, Mayo Rafael, and Ujaldon Manuel. Biomedical Image Analysis on a Co-operative Cluster of GPUs and Multicores. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 15–25, New York, NY, USA, 2008. ACM.
- [29] Intel. Choose the Right Threading Model (Task-Parallel or Data-Parallel Threading). 2009-05-20. <http://software.intel.com/en-us/articles/choose-the-right-threading-model-task-parallel-or-data-parallel-threading/>.
- [30] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.
- [31] Marco Lohse. *Network-Integrated Multimedia Middleware, Services, and Applications*. VDM Verlag, 2007.

- [32] A. Lovesey. A Comparison of Real Time Graphical Shading Languages. Technical report.
- [33] William R. Mark, Steven R. Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 896–907, New York, NY, USA, 2003. ACM Press.
- [34] Kostas Michalopoulos. A simple CUDA enabled Ray Tracer, 2009-04-14. <http://www.badsectoracula.com/projects/misc/tests/graphics/cuda/raytracer/>.
- [35] Matthew Monteyne. RapidMind Multi-Core Development Platform. Technical report, Rapidmind Inc., feb 2008.
- [36] MotionDSP. vReveal, 2009-04-29. <http://www.vreveal.com/gpu>.
- [37] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.
- [38] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*, 2008. http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf.
- [39] NVIDIA. *NVIDIA CUDA Reference Manual 2.0*, 2008. http://developer.download.nvidia.com/compute/cuda/2_0/docs/CudaReferenceManual_2.0.pdf.
- [40] NVIDIA. NVIDIA CUDA SDK 2.0, 2009-04-29. http://www.nvidia.com/object/cuda_get.html.
- [41] NVIDIA. Tesla Unified Graphics and Computing Architecture, 2009-04-29. <http://www.nvidia.com/tesla>.
- [42] NVIDIA. NVIDIA CUDA Zone, 2009-05-04. <http://www.nvidia.com/cuda>.
- [43] NVIDIA. NVIDIA CUDA Visual Profiler 1.0, 2009-05-25. http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/profiler/CudaVisualProfiler_linux_1.0_13June08.tar.gz.
- [44] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In Christophe Schlick and Werner Purgathofer, editors, *STAR Proceedings of Eurographics 2004*, pages 21–51, Grenoble, France, sep 2004. Eurographics Association.
- [45] Pegasys. TMPEGEnc Xpress 4.0, 2009-04-29. <http://tmpgenc.pegasys-inc.com/en/product/te4xp.html>.
- [46] Bart Pieters, Dieter Van Rijsselbergen, Wesley De Neve, and Rik Van de Walle. Performance Evaluation of H.264/AVC Decoding and Visualization using the GPU. volume 6696. SPIE, 2007.

- [47] Michael Repplinger, Martin Beyer, and Philipp Slussalek. Multimedia Processing on Many-core Technology using Distributed Multimedia Middleware. Technical Report 2009-01, Saarland University, Saarbrücken, Germany, 2009.
- [48] Michael Repplinger, Florian Winter, Marco Lohse, and Philipp Slusallek. Parallel Bindings in Distributed Multimedia Systems. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2005)*, pages 714–720. IEEE Computer Society, 2005.
- [49] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (23th PPOPP'2008)*, pages 73–82, Salt Lake City, UT, feb 2008. ACM SIGPLAN.
- [50] N. Satish, M. Harris, and M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. *Submitted to SuperComputing*, 2008.
- [51] Carsten Schmitt. A Multimedial Analysis of the Tradeoff between Performance and Modularity based on a Middleware Extension for a System with Hard Technical Constraints. Master's thesis, RWTH Aachen, 2008.
- [52] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.
- [53] John Stratton, Sam Stone, and Wen mei Hwu. MCUDA: An Efficient Implementation of CUDA Kernels on Multi-cores. Technical Report IMPACT-08-01, University of Illinois at Urbana-Champaign, March 2008.
- [54] M. Suess and C. Leopold. Implementing Data-Parallel Patterns for Shared Memory with OpenMP. In *Proceedings of the International Conference on Parallel Computing (PARCO 2007)*, Juelich, Germany, 2007.
- [55] Wladimir J. van der Laan, Andrei C. Jalba, and Jos B.T.M. Roerdink. Accelerating Wavelet Lifting on Graphics Hardware using CUDA. Technical Report 2009-01, Institute for Mathematics and Computing Science University of Groningen, 2009.
- [56] Eric Young and Frank Jargstorff. Image Processing & Video Algorithms with CUDA, 2009-05-07. http://www.nvidia.com/content/nvision2008/tech_presentations/Game_Developer_Track/NVISION08-Image_Processing_and_Video_with_CUDA.pdf.