MULTIMEDIA PROCESSING ON MANY-CORE TECHNOLOGIES USING DISTRIBUTED MULTIMEDIA MIDDLEWARE

Michael Repplinger\textsuperscript{1,2}, Martin Beyer\textsuperscript{1}, and Philipp Slusallek\textsuperscript{1,2}

\textsuperscript{1}Computer Graphics Lab, Saarland University, Saarbrücken, Germany
\textsuperscript{2}German Research Center for Artificial Intelligence (DFKI), Agents & Simulated Reality, Saarbrücken, Germany
email: michael.repplinger@dfki.de, beyer@graphics.cs.uni-sb.de, philipp.slusallek@dfki.de

ABSTRACT

Today, there are increasingly more powerful many-core processors (MCPs) that are integrated into stationary and mobile devices, freely programmable and well suited for multimedia processing. However, there are still many obstacles and problems when programming multimedia applications for MCPs. In this paper we show that all these obstacles and problems can be solved by treating a device with an integrated MCP in the same way as a distributed system and using distributed multimedia middleware even for developing locally running applications. This approach completely hides all specific aspects of an MCP while the application can fully exploit the processing power of MCPs in local and even remote devices.

KEY WORDS

Parallel and Distributed Processing, Distributed Multimedia Middleware, Many-Core, CUDA

1 Introduction

Available distributed multimedia middleware solutions such as the Network-Integrated Multimedia Middleware (NMM) \cite{7} consider the network as an integral part of their architecture and allow transparent use and control of stationary and mobile devices across the network. To hide technology and network specific aspects from the application, they apply the concept of a \textit{distributed flow graph}, providing a strict separation of media processing and media transmission as can be seen in Figure 1.

The \textit{nodes} of a distributed flow graph represent processing units and hide all aspects of the underlying technology used for media processing. \textit{Edges} represent connections between two nodes and hide all specific aspects of data transmission within a \textit{transport strategy} (e.g., pointer forwarding for local and TCP for network connections). Thus, media streams can flow from distributed source to sink nodes, being processed by each node in-between.

However, upcoming many-core technologies and processors like Nvidia’s \textit{Compute Unified Device Architecture} (CUDA) \cite{8} on top of GPUs or IBM’s \textit{Cell Broadband Engine} (CBE) \cite{6} allow highly parallel processing within a single system and are well suited to be used for multimedia processing.

In general, an MCP can only execute algorithms for data processing, we call \textit{kernels}, while the corresponding control logic still has to be executed on the CPU. The main problem for a software developer is that a kernel runs in a different address space than the application itself. To exchange data between the application and the kernel, specialized communication mechanisms (e.g., DMA data transfer), memory areas, and special scheduling strategies have to be used. This seriously complicates integrating MCPs into new and existing multimedia applications. However, these problems are quite similar to the problems solved by a distributed multimedia middleware.

The general idea presented in this paper is to solve these problems by treating collocated CPUs and MCPs like a distributed system. As an example, we use CUDA. Employing distributed multimedia middleware makes it possible to attain the following three goals:

1. hide CUDA specific aspects from the application,
2. enable the \textit{efficient} combination of nodes using CPU and GPU for media processing, i.e., avoid unrequired memory copies and use all GPUs within a system,
3. and transparently use remote GPUs for processing.

In Section 2 we discuss CUDA specific issues and evaluate the requirements for integrating CUDA into a distributed multimedia middleware to reach the above goals. Section 3 discusses related work and Section 4 presents the integration of CUDA into NMM. Section 5 shows how to extend a distributed multimedia middleware so that it can transparently use remote GPUs for media processing. Section 6 presents performance measurements showing that multimedia applications using CUDA can even improve their performance when using our approach. Section 7 concludes this paper and highlights future work.
2 Requirements

The Nvidia CUDA Toolkit [8] offers great possibilities for parallel programming on GPUs but does not provide convenient abstractions for multimedia applications. We identified the following five requirements to be supported by a distributed multimedia middleware in order to integrate CUDA and to reach the three goals introduced in Section 1.

(1) Distributed Flow Graph The concept of distributed flow graph is required to achieve a strict separation of media processing and transmission which allows presenting nodes using the CPU or GPU in a unified way to the application. CUDA kernels can then be integrated into nodes. In the following, nodes using the CPU for media processing are called CPU nodes while nodes using the GPU are called GPU nodes. A GPU node runs in the address space of the application but configures and controls kernel functions running on a GPU. Since kernels of a GPU node run in a different address space, media data received from main memory has to be copied to the GPU before it can be processed. CUDA provides DMA transfer to reduce the overhead caused by memory transfers from CPU to GPU and vice versa.

However, since DMA transfer is only required when connecting nodes that process data in different address spaces (e.g., GPU and CPU nodes), data should not be copied within a node to avoid dependencies and ensure extendability. Therefore, a strict separation of media processing and transmission is essential. This is ensured in the approach of a distributed flow graph. Here, the DMA transfer has to be integrated as a transport strategy and is completely hidden from the application and independent of nodes.

(2) Parallel Binding In general, a multimedia stream consists of multimedia data, called buffers, and control information, called events, while a strict message order has to be preserved [9]. Since a GPU can only process buffers, events have to be sent to and executed within the corresponding GPU node. This means that different transport strategies have to be used to send messages of a data stream to a GPU node, i.e., DMA transfer for media data and pointer forwarding for control events. In [9] the concept of parallel binding, as shown in Figure 2, is introduced. This allows to use different transport strategies for buffers and events, while the original message order is preserved.

Using (1) distributed flow graph together with (2) parallel binding hides CUDA specific aspects from the application and thus constitutes the completion of our first goal.

(3) Shareable Buffer Manager CUDA needs memory allocated as page-locked memory on the CPU side to use DMA communication for transporting media data, which is generally not used by CPU nodes. To enable efficient communication and avoid unnecessary memory copies between CPU and GPU nodes, a distributed multimedia middleware has to support shareable buffer managers. Shareable buffer managers are responsible for allocating and releasing specific memory blocks, e.g., page-locked memory, and – which is the important aspect here – they can be shared between multiple nodes and transport strategies. This allows a transport strategy or node to inform all preceding nodes and transport strategies to use a specific buffer manager for media processing.

(4) Scheduling Strategies To consider CUDAs special requirements regarding scheduling and to achieve an efficient integration of CUDA, support for different scheduling strategies within a distributed multimedia middleware is essential. As soon as a CUDA specific operation is executed (e.g., allocating page-locked memory on CPU side), all following operations related to this memory are bound to the same GPU. Moreover, all following CUDA operations must be initiated by the same application thread. Due to the complexity of using multiple application threads together with CUDA, Nvidia proposes to use a single application thread in its programming guide [8]. This is not acceptable within a distributed multimedia middleware, especially because one application thread cannot be used to access and control different GPUs. Thus, all components of a distributed multimedia middleware have to be enabled to request a specific application thread for processing.

Support for (3) sharable buffer manager together with support for different (4) scheduling strategies enables efficient communication between CPU and GPU nodes which constitutes the completion of our second goal.

(5) Pipeline of Transport Strategies A pipeline of transport strategies is required to use remote GPU nodes. A GPU only has access to the main memory of the collocated CPU and is not able to send buffers directly to a network interface. If a buffer located on a GPU has to be sent to a remote system, it has to be copied to the main memory first. In a second step, the buffer can be sent to a remote system using standard network protocols like TCP. This requires to set up a pipeline of different transport strategies within a parallel binding in order to reuse existing transport strategies. Support for (5) pipeline of transport strategies
enables transparent use of distributed CPU and GPU nodes and constitutes the completion of our third and last goal.

3 Related Work

Due to the wide acceptance of CUDA, there already exist some CUDA specific extensions, but none of them achieve all of the three goals introduced in Section 1. Typical libraries on top of CUDA, like GpuCV [1] and OpenVidia [4], completely hide the underlying GPU architecture. Since they act as black box solutions it is difficult to combine them with existing multimedia applications that use the CPU. However, the CUDA kernels of such libraries can be reused in the presented approach. Hartley et al. [5] integrated CUDA in a grid computing framework which is built on top of the DataCutter middleware, to speed up compute intensive tasks. Since the DataCutter middleware focuses on processing a large number of completely independent tasks, it is not suitable for multimedia processing.

Available distributed multimedia middleware solutions like NMM [7], NIST II [3] and Infopipe [2] are especially designed for distributed multimedia processing, but only NMM and NIST II support the concept of a distributed flow graph. However, the concept of parallel binding (requirement (2)) as well as shareable buffer managers (requirement (3)) is only supported by NMM and not by NIST II. Moreover, in NMM each node can choose its own scheduling approach, and scheduling information (e.g., thread IDs) can be exchanged between all components of the flow graph either as part of buffers or as special events. This allows to combine different scheduling strategies within NMM (requirement (4)). The combination of different transport strategies within parallel binding (requirement (5)) is not support by NMM. However, to the best of our knowledge, none of the available distributed multimedia middleware solutions support this functionality.

Altogether, there is no framework on top of CUDA that attains all three goals stated in Section 1 and no multimedia middleware that fulfills the five requirements stated in 2.

4 Integration of CUDA

We use a three layer approach for integrating CUDA into NMM as can be seen in Figure 3. All three layers can be accessed from the application, but only the first layer, which includes the distributed flow graph, can be seen by default. Here, all CUDA kernels are encapsulated into specific GPU nodes, so that they can be used within a distributed flow graph for media processing. Since processing of a specific buffer requires that all following operations have to be executed on the same GPU, our integration of CUDA utilizes this aspect and ensures that all following GPU nodes use the same GPU. Therefore, GPU nodes are interconnected using the LocalStrategy which simply forwards pointer of buffers.

However, the concept of parallel binding is required for connecting CPU and GPU nodes. Here, incoming events are still forwarded to a LocalStrategy because a GPU node processes events in the same address space as a CPU node. Incoming buffers are sent to a CPUGPUstrategy or GPUCPUstrategy to copy media data from main memory to GPU memory or vice versa using CUDA's asynchronous DMA transfer. NMM also provides a connection service that is extended to automatically choose these transport strategies for transmitting buffers between GPU and CPU nodes. This shows that the approach of a distributed flow graph hides all specific aspects of CUDA to the application and thus attains the first goal introduced in Section 1.

The second layer enables efficient memory management. Page-locked memory that can be directly copied to a GPU is allocated and managed by a CPUBufferManager, while a GPUBufferManager allocates and manages memory on a GPU. Since a CPUGPUstrategy requires page-locked memory to avoid unnecessary copy operations, it forwards a CPUBufferManager to all predecessor nodes. This is enabled by the concept of shareable buffer managers, described in Section 2. As can be seen in Figure 3, this is done as soon as a connection between a CPU and GPU node is established. Again, this shows the benefit of using a distributed multimedia middleware for multimedia processing on an MCP. GPU nodes can be combined with existing CPU nodes in an efficient way, i.e., without unrequred memory copies, and without changing the implementation of existing nodes.

The lowest layer is responsible for managing and scheduling of available GPUs within a system. Since different GPUs cannot be accessed by using the same application thread, the CUDAManager maintains an individual thread for each GPU, called GPU thread. If a component executes a CUDA operation within one of its methods (e.g., executes a kernel), it requests the CUDAManager to invoke this method by using a specific GPU thread. Execut-
Figure 4. This figure shows buffer processing using a combination of CPU and GPU nodes.

Figure 4. This figure shows buffer processing using a combination of CPU and GPU nodes.

4.1 Using Multiple GPUs

The last step to attain the second goal introduced in Section 1 is to automatically use multiple GPUs within the same system. The most important influence on scheduling, especially when using multiple GPUs, is that page-locked memory is bound to a specific GPU. This means that all following processing steps are bound to a specific GPU but GPU nodes are not explicitly bound to a specific GPU. As soon as multiple GPUs are available, the processing of the next media buffer can be initialized on a different GPU.

However, this is only possible if a kernel is stateless and does not depend on information about already processed buffers, e.g., a kernel that changes the resolution of each incoming video buffer. In contrast to this, a stateful kernel, e.g., for encoding or decoding video, stores state information on a GPU and cannot automatically be distributed to multiple GPUs.

When using a stateful kernel inside a GPU node, the corresponding CPUToGPUStrategy uses a CPUBufferManager of a specific GPU to limit media processing to a single GPU. But if a stateless kernel inside a GPU node is used, the corresponding CPUToGPUStrategy forwards a CompositeBufferManager to all preceding nodes. The CompositeBufferManager includes CPUBufferManager for all GPUs, and when a buffer is requested it asks the CUDAManager which GPU should be used and returns a page-locked buffer for the corresponding GPU. So far we implemented a simple round robin mechanism that is used inside the CUDAManager to distribute buffers one GPU after the other. Together, this integration of CUDA into NMM attains goal 1 and 2 as stated in Section 1.

5 Pipelined Parallel Binding

To use remote GPUs for media processing, the combination of different transport strategies within parallel binding has to be supported, as described in Section 2. This is for example required, if a video stream is received from a remote source and should be directly send to the local GPU.
for decoding. Since such a functionality is not supported by NMM or any other existing multimedia middleware, we propose the concept of **pipelined parallel binding** as an extension to the parallel binding described in [9].

The general idea of this concept can be seen in Figure 5. This figure shows how to connect two remote GPU nodes. In this example, three transport strategies, i.e., a **GPUtoCPUStrategy**, a **TCPStrategy**, and a **CPUtoGPUStrategy**, are required to send buffers between two remote GPU nodes. For sending events, a single **TCPStrategy** is sufficient because events are processed on the CPU.

To automatically find suitable combinations of transport strategies we propose the following extensions to a distributed multimedia middleware. These were also integrated into NMM. First, we uniquely describe an address space by using a so called **location information** that consists of the following key value pairs: **Domain** specifies the name of the network domain of a system. **Host** specifies the hostname of a system within a network domain, **PID** specifies the process ID to distinguish between different applications running on the same host and **Core** describes the used processor (e.g., CPU, GPU, CBE, . . .).

Each node in NMM is extended by a location information for buffer and event processing. The keys **Domain**, **Host** and **PID** are automatically set by NMM as soon as a node is instantiated. Only the key **Core**, which is set to **CPU** by default, has to be changed within a node that does not use the CPU for processing events or buffers.

Furthermore, each transport strategy is extended to provide two types of location information, called **source location** and **destination location**, that describe the address spaces between which buffers and events can be transported. If a transport strategy supports arbitrary or multiple values for a specific key, it can specify a wildcard or a list of values, respectively. For example, transport strategies using a network protocol can in general be used for sending data between different hosts, domains and processes. Thus a transport strategy specifies a wildcard for these values.

Since different transport strategies can be used for exchanging data between the same address spaces, e.g., for network transport, each transport strategy is extended by a suitable weight \( w > 0 \) to automatically decide which transport strategy is used by default. However, an application can change these suitable values if required. Using this information, we developed a connection algorithm that is able to (1) find possible combinations of transport strategies that are stored in a **connection graph** and (2) to choose a default combination for automatic connection setup based on the weight of transport strategies.

If two nodes are connected, the connection algorithm creates the corresponding connection graph for buffers and events. Figure 6 shows the establishment of the connection graph for buffers between two remote GPU nodes. The location information of the GPU nodes is used as source and sink of the connection graph (Figure 6 (a)). Afterwards, source and destination location of transport strategies are added as nodes of the connection graph, if they either produce a **partial match** (e.g., Figure 6 (b) and (c)) or a **complete match** (e.g., Figure 6 (d)). The weight of a transport strategy is stored in added edges.

A **partial match** occurs if a transport strategy is found on the source location of which is **identical** with the destination location of the preceding location information. Two instances of location information are identical if and only if they have at least one identical value for each key. Wildcards are replaced by matching values. Furthermore, the destination location has to differ at least in one key with the sink of the connection graph, and the added pair of location information must not already occur on the corresponding path of the connection graph in order to avoid cycles. A **complete match** occurs if the destination location is identical with the sink of the connection graph. As soon as a complete match has been found, a valid combination of transport strategies has been found to establish a connection between two nodes.

In a second step, the connection algorithm determines the shortest path of the graph. Since the connection graph has a single source, it can be reduced to the standard graph problem **Single Source Shortest Path** (SSSP). Moreover, all edges have a weight \( w > 0 \), so we apply Dijkstra’s algorithm for finding the shortest path.

Altogether, the proposed approach of pipelined parallel binding allows to use GPU nodes for distributed processing and thus completes the third goal introduced in Section 1. Furthermore, this approach achieves a maximum of flexibility and re-usability because only the MCP specific transport strategies have to be added. These are automatically used and combined with existing transport strategies by the above described connection algorithm.
Table 1. Performance of the NMM-CUDA integration versus a single threaded reference implementation. Throughput is measured in buffers per second [Buf/s].

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>6572</td>
<td>6771 (103 %)</td>
<td>10762 (118 %)</td>
</tr>
<tr>
<td>300</td>
<td>996</td>
<td>1175 (118 %)</td>
<td>2317 (228.6 %)</td>
</tr>
<tr>
<td>1000</td>
<td>513</td>
<td>583 (114 %)</td>
<td>1173 (228.7 %)</td>
</tr>
<tr>
<td>2000</td>
<td>259</td>
<td>297 (114.7 %)</td>
<td>592 (228.6 %)</td>
</tr>
<tr>
<td>3000</td>
<td>174</td>
<td>203 (116.7 %)</td>
<td>407 (233.9 %)</td>
</tr>
</tbody>
</table>

6 Performance Measurements

For all performance measurements we use a PC with an Intel Core2 Duo 3.33 GHz processor, 4 GB RAM and 2 Nvidia GeForce 9600 GT graphics boards with 512 MB RAM, running 64 Bit Linux and CUDA Toolkit 2.0. In order to measure the overhead of the presented approach, we compare the maximum throughput of buffers processed on a GPU using a reference program that copies data from CPU to GPU, executes a kernel and finally copies data back to main memory using a single application thread with a corresponding flow graph that consists of two CPU nodes and one GPU node in between using the same kernel. Based on the throughput of buffers per second that can be passed, we compare the reference implementation and the corresponding NMM flow graph. Since NMM inherently uses multiple application threads, which is not possible by the reference application without using a framework like NMM, these measurements also include the influence of using multiple threads.

For all measurements, we used a stateless kernel that adds a value to each byte of the buffer. The resulting throughput with different buffer sizes can be seen in Table 1. The achieved throughput of our integration is up to 16.7% higher compared to the reference application. These measurements show that there is no overhead when using NMM as distributed multimedia middleware together with our CUDA integration, even for purely locally running applications. Moreover, the presented approach inherently uses multiple application threads for accessing the GPU, which leads to a better exploitation of the used GPU. Since CUDA operations are executed asynchronously, the CPU only spends a negligible amount of time to issue the kernel calls which can then be processed on the GPU. Finally, adding a second GPU can double the buffer throughput for larger buffer sizes, if a stateless kernel is used.

7 Conclusion and Future Work

In this paper we demonstrated that a distributed multimedia middleware like NMM is suitable for multimedia processing on a GPU. Moreover, it fulfills three essential goals: 1. to hide CUDA-specific aspects from the application, 2. to enable the efficient combination of components using CPU and GPU for media processing, and 3. the transparent use of remote GPUs for media processing. From our point of view, a distributed multimedia middleware such as NMM is essential to fully exploit the processing power of many-core technologies, while still offering a suitable abstraction for developers. Thus, future work will focus on integrating emerging many-core technologies to conclude on which functionality should additionally be provided by a distributed multimedia middleware.

References