The handle http://hdl.handle.net/1887/21017 holds various files of this Leiden University dissertation

Author: Balevic, Ana
Title: Exploiting multi-level parallelism in streaming applications for heterogeneous platforms with GPUs
Issue Date: 2013-06-26
Chapter 5

PPN Execution on Heterogeneous Platforms with GPUs

5.1 Introduction

In Chapter 4, we have shown how to construct a two-level program featuring task, pipeline and data parallelism. The top-level of the resulting program contains coarse-grain autonomous tasks communicating via channels that are generated as FIFO buffers from the PPN specification. Statements executed by computationally intensive tasks are further transformed for data parallelism in order to be offloaded for execution on an accelerator. In this chapter, we introduce novel techniques for execution of PPN process’ statements on an accelerator while improving the efficiency of host-accelerator communication.

Let us again consider the two-level M-JPEG PPN illustrated in Figure 4.2. The nodes of the top-level PPN, which are denoted in this figure as $S'$, $T'$, $Q$ and $VOUT$, are implemented as parallel tasks and mapped for execution on different threads running on a multicore CPU of the host platform. Communication between the four tasks is organized exclusively using FIFO buffers. At each iteration, the $S'$ process writes a token into its output buffer. The $T'$ process reads the token from the buffer, and executes the next process iteration which executes the statement, that contains a call to the mainDCT function. Our goal at this stage is simple - and that is to offload the execution of mainDCT onto an accelerator.

After generating data parallel accelerator code for the mainDCT (either manually or e.g., using the approach presented in Chapter 3), it is necessary to provide the host-side accelerator management code for kernel offloading onto a GPU. To execute the node $T$ (DCT) on the GPU, two issues need be resolved:
5.1. INTRODUCTION

- (1) How to manage the offloading of PPN process iterations onto an accelerator (such as GPU)
- (2) How to reduce host-accelerator communication overheads, i.e. how to improve the communication efficiency

The GPU accelerator is typically seen as a co-processor managed by the host. The traditional model for kernel offloading consists of three phases:

- **copy-in** - data transfer from host memory (main memory) to device memory (GPU global memory),
- **kernel** - execution of process iteration (kernel) on the GPU
- **copy-out** - data transfer from device memory to host memory

In compiler-assisted parallelization frameworks, the kernel offloading is traditionally realized with a *drop-in code replacement*. The drop-in code replacement refers to the substitution of the sequential code on the host-side with the kernel offloading mechanism. We refer to this mechanism as *Synchronous Offloading of a Kernel* (SOK). The SOK mechanism is illustrated for function call $Y$ in Figure 5.1.

![Figure 5.1: Blocking kernel offloading using the drop-in code replacement.](image-url)

The body of the PPN process $T'$ is shown in Figure 5.1(a). At each iteration $(f, is)$, $T'$ executes statement $DS$. The statement $DS$ invokes the function $Y$ that implements the DCT processing. It takes as input argument a variable of type $row$ that is an array of $HNumBlocks$ block elements, transforms it, and returns the resulting $row$. To execute the DCT on the GPU, the call to function $Y$ is simply replaced by the drop-in code replacement shown in Figure 5.1(b). Figure 5.1(b) shows the three steps of synchronous kernel offloading. First, we transfer the input data from the variable $row$ in the host memory (main memory) to the variable $gBlocksIn$ in the GPU memory using a CUDA API call `cudaMemcpy` indicating the direction as...
5.2. APPROACH

cudaMemcpyHostToDevice. For brevity, we denote cudaMemcpy calls from host to device as memcpyH2D, and cudaMemcpy calls from device to host as memcpyD2H. Second, we launch the kernel that implements the DCT processing. The kernel reads the data from the gBlocksIn array and writes it to the gBlocksOut array, which is also in the GPU global memory. Once the kernel runs to completion, we transfer the results from the GPU memory gBlocksOut to the host memory rowToken. Once all three actions are completed, the control is returned to the process T’, which only then proceeds with the execution of the next process iteration. We illustrate the GPU execution timeline using the SOK mechanism in Figure 5.2(a).

Figure 5.2: (a) Have: Synchronous execution of CUDA operations (no overlaps), (b) Want: Asynchronous execution (overlapped).

The main limitation of the SOK mechanism is that this type of kernel offloading is blocking. The PPN process can not start the data transfer from host to device and the computation of the next iteration, until all three phases of the current iteration have completed. In case of streaming applications, this inhibits the pipelining of data transfers and processing on the GPU. The impact of data transfers can be mitigated by overlapping data transfers and computation (see Figure 5.2(b)), as demonstrated in several application case studies [14, 143]. This can be achieved following NVIDIA’s technical note [100] that contains an example code pattern that enables overlapping of data transfers and kernel execution. State of the art frameworks for run-time task scheduling, such as StarPU [10], use asynchronous data transfers for communication with the GPU. Farago and Nikolov [52] experimented with the SOK mechanism in the context of PPNs, which results in the sequential timeline given in Figure 5.2(a).

The question that we address in this chapter is how to efficiently offload kernel execution from a PPN node to an accelerator and achieve overlapping of computation and communication phases as illustrated in Figure 5.2(b).

5.2 Approach

To enable overlapping of computation and communication phases of consecutive PPN process iterations illustrated in Figure 5.2(b), we propose a model-based Asynchronous Offloading of a Kernel (AOK) mechanism for kernel offloading to acceler-
5.2. APPROACH

ators, which leverages the asynchronous, data-driven nature of the PPN model combined with asynchronous data transfers to the accelerator.

As a step to asynchronous communication with the accelerator, we differentiate between two types of channels: the channel between processes executing their process iterations on the same device (e.g. a host CPU), and the channels between two processes executing process iterations on different devices (e.g. a host CPU and a GPU accelerator). For simplicity, we refer to the first type of channels as a Shared Memory Channel (SMC) (or channel type A) and the second type as Distributed Memory Channel (DMC), or simply channel type B. When the SOK mechanism is used, the PPN process makes use only of SMC channels to connect to other processes, as illustrated in Figure 5.3(a). In this figure, the data transfers between the host and GPU are denoted as dataxfer. For processing on the GPU, the process \( P \) first puts data into the SMC channel \( Ch_1 \), the process \( T' \) reads the token from the SMC channel \( Ch_1 \), and then invokes the kernel offloading mechanism shown in Figure 5.1(b). The SOK mechanism invokes three CUDA operations: (1) synchronous data transfer from host to GPU, (2) kernel execution, and (3) synchronous data transfer from GPU to host. The timeline of the CUDA operation execution for the three consecutive process iterations is shown in Figure 5.3(c). Only after the all three CUDA operations have completed the control is returned into process \( T' \), which then puts the results of GPU execution into the SMC channel \( Ch_2 \).

The core idea of our approach for asynchronous kernel offloading is to directly send the data produced by \( P \) to the GPU without transferring it first to the host memory of process \( T' \), by taking advantage of different channel types. Following the AOK approach, process \( P \) does not block waiting for the data transfer to complete. Instead, process \( P \) is free to proceed with the next process iteration and start the next data transfers to the GPU. As soon as the GPU receives the input data from process \( P' \), it launches the DCT kernel, and once the kernel is completed, we directly transfer the results from GPU memory to the process \( C \). The arrows between processes...
5.2. APPROACH

$P$ and $T'$, and $T'$ and $C$ in Figure 5.3 indicate synchronization that takes place between processes, and arrows between processes $P$ and $T$ and $T'$ and $C$ indicate the host-accelerator data transfers. So, instead of having $P$ transfer data from the host memory buffer to the host memory buffer of $T'$, and then having $T'$ transfer the data from the host memory buffer into GPU memory using a synchronous data transfer call, $P$ transfers data directly from its host memory buffer into the GPU memory buffer access by $T$, as indicated by dataxfer marks in Figure 5.3. To realize this pattern we take advantage of asynchronous DMA transfers between the host and GPU accelerator. This sort of asynchronous execution enables pipelining of GPU operations (e.g. kernel execution from the first iteration (denoted as $i_1$) can be overlapped with the host-accelerator data transfer from the second iteration (denoted as $i_2$)), thus facilitating the desired overlapping of communication and computation. The timeline of the CUDA operation execution for three consecutive process iterations is shown in Figure 5.3(d).

Our approach is inspired by the technical note from NVIDIA [100] that shows how to use the concept of a CUDA stream to realize overlapping of communication and computation. A CUDA stream describes a sequences of GPU operations (kernel executions, data transfers) that execute in-order. The operations from different streams execute in parallel. This enables overlapping of a kernel execution in stream $s_1$ with a data transfer in stream $s_2$.

In addition, we leverage double-DMA capabilities of Fermi-architecture GPUs to achieve the overlapping of not only data transfers to the GPU and kernel execution, but also to additionally overlap GPU data transfers in different streams. This allows us to upload an input token from iteration $i_3$ to the GPU (data xfer in phase 1), while we are at the same time executing the kernel for iteration $i_2$, and downloading the results of iteration $i_1$ to the host memory (data xfer in phase 3), as shown in Figure 5.3(d). Coupled with the PPN properties of data-driven asynchronous execution, the technical capabilities of modern GPUs enable us to design a model-based approach for overlapping communication and computation on GPU based systems.

The chapter is structured as follows. In Section 5.3.1, we introduce a classification of PPN channels that enables model-driven channel design and mapping. In Section 5.3.2, we give an overview of traditional channel design, and explain extensions that improve its efficiency by reducing the amount of data transferred at each channel access. In Section 5.4, we cover the AOK mechanism. First, we present our stream buffer design for type B channels that enables model-driven overlapping of computation and communication in Section 5.4.1. Second, we show how to apply the stream buffer design to enable AOK in a PPN in Section 5.4.2. Finally, we show experimental results in Section 5.5, and discuss our findings.
5.3. Model-Driven Communication Design

5.3.1 Classification of PPN Channels

Let’s consider a heterogeneous platform containing several discrete devices (architectural components) each with its own private memory. For efficient design and mapping of communication between PPN processes, it is necessary to consider whether the processes communicate through the physically same piece of memory, or not. For example, two processes entirely executing on the host CPU both access the main memory. However, if one process executes on the CPU and the other offloads its iterations to the GPU, these two processes access two physically different memories: the main memory, and the GPU device memory. According to the memory accessed by the two processes, we propose the following classification of PPN channels into two main categories:

- **A. Shared Memory Channel (SMC)**
- **B. Distributed Memory Channel (DMC)**

A Shared Memory Channel (SMC) connects two PPN processes that execute on the same device, and thus access the same memory. If both processes connected by the channel execute on the CPU, such channel is further classified as host-to-host (H2H) SMC. If both processes execute on an accelerator device, such as GPU, such channel is further classified as device-to-device (D2D) SMC.

A Distributed Memory Channel (DMC) connects two PPN processes that execute their process iterations on two devices, each with their own memory. For example, the VIN process executes exclusively on the host CPU, while the DCT process executes kernels on the GPU. In this case, we have a distributed memory system, i.e. both host and accelerator have their own memory space. If the channel’s producer process executes on the host and its consumer process executes on the GPU, we further classify this channel as host-to-device (H2D) DMC. Analogously, if the channel’s producer process executes on the GPU and its consumer process executes on the host, we further classify this channel as device-to-host (D2H) DMC. While SMC channels can be simply realized using some of the available FIFO libraries, host-GPU DMC channels require closer consideration as the communication and synchronization must be carefully handled.

5.3.2 SMC Design

The PPN channels are typically designed as first-in first-out (FIFO) buffers. There is large number of FIFO buffer implementations available. For illustration, we will make a brief overview of a general FIFO buffer design based on a well-known concept
of a circular buffer (ring buffer) and explain a simple extension to this design, that results in minimized data movement over the bus.

A circular buffer is a fixed-size bounded buffer allocated in a single piece of memory. The circular buffer of size $m$ has $m$ slots, which are written and read in a circular fashion. The producer process writes tokens one by one into the buffer. After all slots have been filled up, the producer starts writing from the beginning of the buffer. If there are no empty slots left, we block the producer process until the consumer process reads the next token and thus another slot becomes available. Thus we realize SMC channels by using circular buffers and blocking read/write accesses.

The point-to-point communication in a PPN results in PPN channels having the single producer - single consumer (SPSC) property. Due to this SPSC property, the channel’s producer and consumer PPN nodes can read and write data concurrently from the channel as long as there is a sufficient number of full/empty buffer slots available.

The current realisation of PPN processes requires physical transfer of data tokens from channels into a local memory upon read, and from the local memory into the channel upon write. However, when two processes are connected with SMC, these data transfers are unnecessary. Instead of physically copying a data token into a local variable of the process, it is sufficient to acquire a pointer to the buffer slot holding the data token. For example, instead of copying a complete token (e.g. 1KB block in M-JPEG or larger) into a local variable and then processing the local variable as input argument to the process, it is sufficient to take the pointer to the token and pass it as the input argument to the process. We call this type of access direct access to memory. When direct access to memory is used, additional modifications to the PPN R/E/W execution protocol are required. For this we propose use of an Acquire Direct Access to Memory (ADAM) protocol. In the ADAM protocol, a process first acquires pointers to the tokens that are the actual input and output arguments. Then, it evaluates the process function, and finally, it releases both pointers. The use of the ADAM protocol considerably reduces the amount of data traffic and has significant performance impacts on memory bounded applications.

### 5.3.3 Synchronous DMC Design

Each PPN process connected to a DMC channel accesses a physically separate portion of memory on the system. To realize the communication between the producer PPN process and the consumer PPN process, we need to transfer the data between the memories of the producer and the consumer.

To realize DMC channels we use a distributed double buffering scheme. Let us illustrate this on the host-to-device DMC in Figure 5.4. In this figure, we see a P/C pair of processes. The producer process of the P/C pair, denoted as $P$, executes
5.3. MODEL-DRIVEN COMMUNICATION DESIGN

iterations on the host CPU, a the consumer process of the P/C pair, denoted as $T'$, executes iterations on the GPU. The data produced by $P$ can be used as the input argument at $T'$ only after this data is transferred from the host memory to the GPU memory.

![Diagram](image)

Figure 5.4: SMC vs DMC

To realize this, we first introduce a distributed double buffering scheme. We create one circular buffer in the host memory labeled as buff1h and one circular buffer of the same size in the GPU memory labeled as buff1d. For each circular buffer, we use the SMC design described in Section 5.3.2. The producer writes the data into the buff1h, and the transformer $T$ reads the data from the buff1d.

The data that $P$ writes into the host-side buffer has to be transferred into the GPU-side buffer. There needs to be a mechanism for buffer-to-buffer communication in place. We realize buffer-to-buffer communication by initiating a data transfer (cudaMemcpy) from host to device. The producer initiates the transfer to the GPU, since it is capable of starting the transfer as soon as it has produced the data.

Writing to the DMC from the producer is composed of several operations. First, the producer tries to acquire a slot for writing in host-side memory buffer (buff1). Second, the producer puts the data into host-side memory buffer (buff1). Third, the producer invokes the cudaMemcpy call to transfer data from the host-side buffer into device-side buffer buff2. The cudaMemcpy operation is synchronous, which means that the control returns to the producer only after the GPU transfer has completed. Once the cudaMemcpy returns, the producer signals to the process $T'$ controlling the GPU that the input data is available. The GPU then issues the kernel. In this (synchronous) mode of operation, all GPU operations are issued in the same CUDA stream by default.

The basic DMC model uses synchronous data transfers between host and GPU buffer slots, which results in blocking of the producer process until the transfer has completed. As a consequence, the host-accelerator data transfers cannot be overlapped with the computation.
5.4 Asynchronous Offloading of Kernels (AOK)

The AOK mechanism requires asynchronous host-accelerator communication. We first present an efficient design for asynchronous host-accelerator (i.e. CPU-GPU) communication in Section 5.4.1 and then we show how to leverage this design to realize AOK in PPN in Section 5.4.2.

5.4.1 Asynchronous Stream Buffer Design

To improve the communication efficiency between host and accelerator, we want to overlap computation and data transfers. In this section we extend the DMC design to take advantage of asynchronous data transfers and host-accelerator DMA. The result of this extension is what we call the stream buffer design.

Our goal is to overlap different GPU operations. We distinguish three main categories: data transfers from host to accelerator (type 1 in Figure 5.5), kernel execution (type 2 in Figure 5.5), and data transfers from accelerator to host (type 3 in Figure 5.5). In CUDA, different GPU operations can execute concurrently provided that they are issued in different CUDA streams. In addition, the following requirements must be satisfied: data transfers are used in combination with pinned memory \(^1\) on the host and only asynchronous data transfers are used.

Let us first introduce CUDA streams for the execution of operations in the three categories. For each type of data transfer we introduce a separate stream, namely an upload stream denoted as \(s_{H2D}\) for data transfers from host to accelerator, and a download stream denoted as \(s_{D2H}\) for data transfers from accelerator to host. Execution of GPU operations in a dedicated stream increases the concurrency from the sequential execution in the default stream \(s_0\) illustrated in Figure 5.5(a) to the concurrent execution of operations in the three streams as depicted in Figure 5.5(b).

\[\text{default stream } s_0\]
\[\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 2 & 2 & 3 & 3 \\
\end{array}\]
\[\text{upload stream } s_{H2D}\]
\[\begin{array}{cccccccc}
1 & 1 & 1 & 1 \\
\end{array}\]
\[\text{kernel stream } s_k\]
\[\begin{array}{cccccccc}
1 & 2 & 2 & 2 \\
\end{array}\]
\[\text{download stream } s_{D2H}\]
\[\begin{array}{cccccccc}
3 & 3 & 3 & 3 \\
\end{array}\]

Figure 5.5: Single stream vs Dedicated streams.

The stream buffer design in Figure 5.4(b) issues host-to-accelerator data transfers into the upload stream \(s_{H2D}\). To execute independent kernels concurrently, we also introduce a separate kernel execution streams for different kernels. In the M-JPEG example, we offload only the DCT kernel, hence it is sufficient to have a single kernel

\(^1\)Pinned memory is non-pageable portion of system memory, i.e. it cannot be swapped out to disk by the operating system.
5.4. ASYNCHRONOUS OFFLOADING OF KERNELS (AOK)

execution stream, which we denoted as $s_K$. The accelerator-to-host data transfers are issued into the download stream $s_{D2H}$.

Let us now explain how we combine the concept of stream-based execution with asynchronous transfers to achieve the desired overlapping. Use of CUDA streams is only possible with asynchronous data transfers. CUDA provides an asynchronous API for its device-to-host and host-to-device $\text{cudaMemcpy}$ operations. The asynchronous data transfers are realized using $\text{cudaMemcpyAsync}$. The invocation of an asynchronous CUDA call immediately returns control to the caller (e.g. producer process), without waiting for the GPU operation to complete. This means that the producer process can immediately proceed with processing the next iteration, in which it produces the next result and issues the next host-to-device data transfer. The host-to-device transfers are executed in order, resulting in a sequence of data host-to-device transfers (type 1) in stream $s_{H2D}$ shown in Figure 5.5(b).

![Figure 5.6: Stream Buffer Design with Asynchronous Data Transfers: Interaction Diagram.](image)

When using synchronous data transfers, the producer has to wait until a data transfer to the GPU completes. After the data transfer has finished, the producer process signals to the consumer process that input data is available, and that it can launch its GPU kernel. As the end time of asynchronous host-device data transfers is not known, it is not possible to use the same signaling scheme in the asynchronous execution mode. Thus, the use of asynchronous data transfers imposes additional synchronization requirements.

These additional synchronization requirements are captured in the producer-initiates consumer-completes (PICC) protocol for asynchronous communication over DMC-
5.4. ASYNCHRONOUS OFFLOADING OF KERNELS (AOK)

type channels between PPN processes that run on host and accelerator. The communication pattern using the PTT protocol is illustrated by the interaction diagram in Figure 5.6. The PTT protocol starts when the producer process completes its execute phase and needs to write the results into the channel, i.e. when it invokes the put operation on the buffer. As the first step of the put operation, the producer process initiates an asynchronous data transfer (memcpyH2DAsync) to the GPU. To support the use of asynchronous data transfers, we introduce an additional GPU-side signaling mechanism. The role of the GPU-side signaling mechanism is to record a DATARDY event on the GPU after the asynchronous data transfer memcpyH2DAsync initiated by the producer P has completed. We realize this by issuing a GPU-side DATARDY event in the upload stream sH2D immediately after the asynchronous data transfer call, as the second step of the put operation. Since the operations in the same CUDA stream must (1) run to completion and (2) execute in order, the GPU captures the DATARDY event only after the data transfer completes, even though the data transfer call is asynchronous. The consumer needs to wait on the DATARDY event on the GPU to occur, which is illustrated by waitEvent(DATARDY) phase in the consumer side of the interaction diagram. Observing a DATARDY event, indicates to the consumer that the data transfer is complete. Hence, the consumer completes the PTT protocol. As soon as DATARDY event is captured, the consumer processes continues its execution. If the kernel requires data from multiple producers, the consumer process waits on multiple DATARDY events. If DATARDY events for all input arguments have been captured, the consumer process C exits the blocking read state, and enters its execution phase, in which it issues its GPU kernel in the compute stream.

In the CUDA programming model, waiting on an GPU event that has not yet been issued is not defined. To insure the correct execution of the PTT protocol with the current CUDA API, it is necessary to include additional host-side synchronization that indicates to the consumer when it is safe to start waiting on the GPU-side DATARDY event. We realized this in practice with a CPU-side semaphore producedCount in Figure 5.6. The producer increases its produceCount semaphore immediately after it issues the asynchronous data transfer. The consumer waits on the produceCount semaphore until the producer sets the new value. This indicates to the consumer process C that the data transfer has started and that it should start waiting on the DATARDY event indicating that the transfer has finished.

5.4.2 Application in PPN Execution

In this section, we instantiate PPN channels between the CPU and the GPU (DMC type) using a stream buffer design presented in Section 5.4.1. The use of the stream buffer design enables asynchronous host-accelerator communication in a PPN, and allows us to use AOK in a PPN instead of the SOK mechanism. Let us now illustrate
### 5.4. Asynchronous Offloading of Kernels (AOK)

AOK in a PPN on a three-node pipeline illustrated in Figure 5.7. We will discuss the flow of data from the CPU producer process $P$ via the GPU transformer process $T'$ to the CPU consumer process $C$.

![PPN Node P (code snippet):](image1)

![PPN Node T' (code snippet):](image2)

Figure 5.7: AOK using stream buffers for communication.

Each PPN process is executed by an independent thread of execution, and proceeds asynchronously in a data-driven fashion. The processes $P$ and $C$ execute on the CPU, while process $T$ executes the DCT kernel on the GPU. Let us explain how the nodes execute using the asynchronous mechanisms for data transfers and kernel offloading. After generating data, process $P$ puts data into the stream buffer $SB_1$ that implements the host-to-device DMC channel. Once a token is in the host-side buffer $buff1h$, $P$ first queues up the asynchronous host-device data transfer into the upload stream $SH2D$ denoted in code as $stream\_up$. Second, $P$ starts the two phase CPU-GPU synchronization according to the PICC protocol presented in Section 5.4.1. A $DATARDY$ event is queued up by $P$ after the data transfer in the upload stream $stream\_up$ to signal the end of data transfer. Second, the $P$ signals via host-side semaphore to the process $T'$ that the data transfer has been initiated, and proceeds to execute the next process iteration.

Process $T'$ waits on the host-side semaphore before it starts the GPU-side synchronization. As an important difference to the SOK mechanism, the $T'$ must also acquire an empty buffer slot for storing its results before it can proceed with the execution. We realize this by blocking the process $T'$ that makes use of AOK until all input data and an empty slot in the output channel are available in the GPU memory. Thus, the blocking parts of the blocking on read and the blocking on write phases of the process execution are now combined into a single phase at the start of each process iteration.
As a consequence, only this step of the process execution with AOK is blocking. All subsequent steps, i.e. reading data, processing data, writing data, and starting data transfer to the consumer, are non-blocking.

After process $T'$ observes the GPU event, a new data transfer can take place on the PCIe bus while the GPU executes the DCT kernel of process $T'$. When $T'$ completes the kernel execution on the GPU, it starts a put operation into the stream buffer $SB_2$. As part of the put operations, $T'$ issues and asynchronous data transfer in download stream $s_{D2H}$, and enqueues its DATARDY event. Once the consumer $C$ captures the DATARDY event, it proceeds with consuming the data produced by the GPU. In the meanwhile, the GPU is already executing the DCT kernel on the next token.

This mode of operation is significantly different from synchronous kernel offloading, as (1) we avoid unnecessary data transfers from the host memory buffer of $P$ into host memory buffer of $T'$, (2) the data transfers and the kernel execution are asynchronous. By combining the stream buffer mechanism with asynchronous execution in a PPN, the computations on the CPU, on the GPU, and a data transfers from CPU to GPU are overlapped. On a high-end GPU, such as a Tesla C2050 with two DMA engines it is also possible to overlap data transfers in different directions. Given a streaming video application with tokens representing frames, the following activities occur simultaneously:

- Transformer process $T$ executes GPU kernel on frame $k$
- The first DMA engine uploads frame $k + 1$ from host buffer to GPU
- Producer process $P$ executes produce function to obtain frame $k + 2$
- The second DMA engine downloads frame $k − 1$ from GPU to host
- Consumer process $C$ executes consume function on frame $k − 2$

Thus, the data-driven asynchronous execution model with stream buffer design enables overlapping of PPN computation on the host (CPU), data transfers to the GPU, as well as kernel execution on the GPU, resulting in simultaneous utilization of all platform devices.

## 5.5 Experimental Results

To determine the efficiency of the stream buffer design and its application for AOK in PPNs, we perform a number of experiments. Let us first describe the experimental setup.
5.5. EXPERIMENTAL RESULTS

5.5.1 Platform Specification

The test platform contains an Intel Core i7-920 Nehalem architecture 2.66GHz processor, Intel Motherboard, and a high-end NVIDIA Tesla C2050 GPU. The Intel Core i7 (Nehalem) is a multi core, Hyper-threading technology (HT) enabled design [44]. Each socket supports one to eight cores, which share the level 3, a local integrated memory controller (IMC) and an Intel QuickPath Interconnect (QPI). The interface to the GPU is via PCIe 2.0 x16 bus. PCIe 1.x is often quoted to support a data rate of 250 MB/s in each direction, per lane. This figure is a calculation from the physical signaling rate (2.5 Gbaud) divided by the encoding overhead (10 bits per byte). This means a sixteen lane (16-lane) PCIe card would then be theoretically capable of 16 × 250 MB/s = 4 GB/s in each direction. For PCI 2.x with data rate of 500MB/s in each direction, the theoretical bandwidth is up to 8GB/s in each direction, which gives us a theoretic maximum of 16GB/s for bi-directional communication. In practice, these numbers are smaller, since they depend also on manufacturer’s design and the profile of the traffic, which is a function of the high-level (software) application and intermediate protocol levels.

5.5.2 Platform Micro-Benchmarks

To obtain estimates of the actual platform performance we conducted a set of micro-benchmarks. First, we measured the host memory performance. Figure 5.8 shows the measured bandwidth of main memory accesses achieved on the host side. We did four experiments. We measured the performance of a data copy from a for loop (memcpy(for − loop)) which is typically used in PPN implementations, and the performance of the GNU C library (memcpy(glibc)). In addition, we include the performance achieved separately by 32-bit read accesses rd32 − bit and 32-bit write accesses wr32 − bit. The experiment was conducted using a single thread of execution and the results were averaged over 100 iterations. Special care has been taken to avoid measuring accesses to CPU caches instead to the memory subsystem. The cache contents were manually "trashed" before conducting each measured memory access. The trashing ensures that the cache does not hold values copied in the previous benchmark iteration, and that we obtain out-of-cache memory access numbers. The results show that the GNU C memcpy(glibc) performs significantly better than when we perform add read and write accesses directly in the for loop. Since our PPN implementation uses manual read/write accesses, it is relevant to be aware of this performance gap. In addition, we found out that all four methods achieve less than 4 GB/s for tokens smaller than 16 KB. The token size in the default M-JPEG PPN is only 1 KB, which results in data access performance below 1 GB/sec. This points to the importance of being able to adjust and tune the token granularity as proposed in Chapter 4.
5.5. EXPERIMENTAL RESULTS

Figure 5.8: Throughput of Host Memory Accesses. Experiment details: Single Thread, Average over 100 iterations.

Second, we measured the scalability of parallel accesses to memory. Contention for shared resources, such as host memory bandwidth, increases with the number of threads accessing the memory. This contention can reach a point that adding more threads does not increase performance. In each experiment, we measured the time required to transfer a fixed total amount of data (64MB) from one location in host memory to another using different number of threads. Each thread gets one chunk of 64MB for transfer. As illustrated in Figure 5.9, the memory bandwidth is already saturated with 2 only threads. Further increasing the number of threads actually reduces performance. This shows that the performance of host memory subsystem is low and sustains only two memory-intensive parallel tasks.

Third, we measured performance of host-device data transfers over PCIe bus. The performance numbers in Figure 5.10 show the throughput achieved over PCIe bus as a function of data transfer size. We measured the performance of both synchronous and asynchronous host-device data transfers. The synchronous transfers via cudaMemcpy API call access pageable memory on the host, while asynchronous transfer via cudaMemcpyAsync API call directly access non-pageable (pinned) memory buffer for direct memory access (DMA) transfer. As Figure 5.10 shows,
5.5. EXPERIMENTAL RESULTS

![Figure 5.9: Scalability of Host Memory Accesses. Experiment details: Total data size 64MB (fixed), each thread copies an equal non-overlapping portion of data. Memory subsystem saturated with only two threads accessing memory concurrently.](image)

chronous transfers from pinned memory (htod – async – pinned and dtuh – async – pinned) achieve a steady throughput between 5000 – 6000 MB/s for data transfers larger than 1MB. The performance of synchronous data transfers from pageable memory (htod – sync – pageable and dtuh – sync – pageable) turned out to be lower in practice and also less predictable. For example, dtuh – sync – pageable fluctuates around 1500 MB/s, while the performance of (htod – sync – pageable suddenly jumps for transfers larger than 60 KB. The significantly better performance of asynchronous transfers can be explained by the use of pinned memory and use of an DMA engine. The pages in pinned memory can not be swapped out of RAM by the operating system, which means that there is no virtual memory management overhead involved in waiting for the operating system to load the pages from the disk. In addition, the use of a DMA engine eliminates the need to involve the CPU in the data transfer. The DMA engine puts directly the data from the pinned memory on the PCIe bus.
5.5. EXPERIMENTAL RESULTS

![Graph showing throughput vs data size for different memory types]

Figure 5.10: Throughput of Host-Accelerator Data Transfers. Comparison of performance for synchronous transfers from pageable memory and asynchronous data transfers requiring pinned memory.

5.5.3 Stream Buffer

We show the maximal communication performance improvement achieved using stream buffer design over synchronous data transfers in Figure 5.11. In this experiment, we transferred data in fixed-size packets over PCIe bus using synchronous host-device transfers from pageable memory, denoted as default (sync xfers) trace, and asynchronous host-device transfers from pinned memory, denoted as streaming (sb – asyncq) trace. The experiment has been repeated for a large number of iterations. We measured the time to complete a round-trip: we transfer the tokens from the host memory to the GPU memory and back. We show the round-trip throughput (MB/s) as a function of data size. The maximal theoretical round-trip throughput with complete overlapping of data transfers in both directions to $2 \times 8 = 16$ GB/s for a PCIe 2.0 bus. In practice, the host-device data transfers achieve 6 GB/s for large data sizes as illustrated in Figure 5.10. This brings the maximum achievable round-trip throughput with overlapping of data transfers to $2 \times 6 = 12$ GB/s. The largest round-trip that we measured was around 8 GB/s, as shown in the last bar of Figure 5.11.
5.5. EXPERIMENTAL RESULTS

Figure 5.11: Increase of Round-Trip Throughput Using The Asynchronous Stream Buffer Design.

Table 5.5.3 shows the average achieved bandwidth on PCIe bus in each direction in case of default (synchronous) data transfers and streaming (asynchronous) data transfers using stream buffer design. The last column gives percentage of time that the transfers were overlapped in case of streaming asynchronous data transfers for different data sizes. As it can be seen for very small data transfers (1024B) there is no overlap due to large runtime overheads. However, for larger, and thus longer, data transfers, we observed more than 90% of time overlap between host-to-device and device-to-host data transfers.

The percentage of the overlapped time is shown in the last column of Table 5.5.3. We also measured the average PCIe transfer bandwidth for both synchronous and asynchronous designs. The results are given in Table 5.5.3. In line with micro benchmarks, we observed higher bandwidth for host-to-device transfers in both settings. We also observed an interesting anomaly. The experiments revealed that the time to complete individual asynchronous host-to-device and device-to-host transfer increases for overlapped transfers. This indicates that the implementation of the PCIe bus on the system does not support simultaneous transfers in both directions at the full speed of PCI. As a consequence, the average bandwidth of individual host-to-
5.6. CONCLUSIONS

To minimize the impact of host-accelerator data transfers, we created a novel method for kernel offloading in PPNs. First, we realized efficient host-accelerator commu-

<table>
<thead>
<tr>
<th>Transfer Type</th>
<th>Data Size [B]</th>
<th>Average H2D PCIe Transfer BW [MB/s]</th>
<th>Average D2H PCIe Transfer BW [MB/s]</th>
<th>Overlap [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>1024</td>
<td>413.644</td>
<td>605.394</td>
<td>0</td>
</tr>
<tr>
<td>Streaming</td>
<td>1024</td>
<td>315.626</td>
<td>313.128</td>
<td>0</td>
</tr>
<tr>
<td>Default</td>
<td>131072</td>
<td>5534.44</td>
<td>5161.91</td>
<td>0</td>
</tr>
<tr>
<td>Streaming</td>
<td>131072</td>
<td>4408.32</td>
<td>4492.17</td>
<td>0</td>
</tr>
<tr>
<td>Default</td>
<td>1048576</td>
<td>6041.44</td>
<td>5531.42</td>
<td>0</td>
</tr>
<tr>
<td>Streaming</td>
<td>1048576</td>
<td>4452.45</td>
<td>4550.42</td>
<td>34</td>
</tr>
<tr>
<td>Default</td>
<td>2097152</td>
<td>6082.41</td>
<td>5527.5</td>
<td>0</td>
</tr>
<tr>
<td>Streaming</td>
<td>2097152</td>
<td>4439.63</td>
<td>4531.64</td>
<td>92</td>
</tr>
</tbody>
</table>

Table 5.1: Percent overlap with streaming device and device-to-host transfers for the streaming case is somewhat smaller, but due to the overlapping we still achieve significant performance gains.

5.5.4 PPN Execution with Asynchronous Kernel Offloading

Let us now give the performance comparison of SOK and AOK in the context of PPN execution. For the experiment, we used a PPN composed of three nodes connected in a simple P/T/C pipeline. Producer \( P \) and consumer process \( C \) execute on the host, while transformer \( T \) offloads its process iteration onto the GPU.

In Figure 5.12, we show the throughput increase when asynchronous methods are used instead of simple synchronous kernel offloading. The PPN using the synchronous kernel offloading (SOK) is represented by the \( \text{ppn} - \text{gpu} - \text{offload} \) trace. The PPN using the asynchronous kernel offloading (AOK) is represented by the \( \text{ppn} - \text{gpu} - \text{sb} - \text{adam} - \text{asynq} \) trace. The two traces in between are hybrid variants: the \( \text{ppn} - \text{gpu} - \text{sb} \) trace represents the PPN using the stream buffer design with distributed double-buffering and synchronous communication, and the \( \text{ppn} - \text{gpu} - \text{sb} - \text{adam} \) trace is its optimization that in addition uses direct memory access to buffer slots. In this experiment, we observed significant throughput increase when using the methods developed in this chapter. The AOK approach lead to up to a 4.3\( \times \) improvement over the traditional SOK approach for kernel offloading.

5.6 Conclusions

To minimize the impact of host-accelerator data transfers, we created a novel method for kernel offloading in PPNs. First, we realized efficient host-accelerator commu-
5.6. CONCLUSIONS

In Figure 5.12, we see the Throughput Increase by Using AOK vs SOK in a PPN. The diagram illustrates the performance comparison between different data sizes (1KB) and throughput (MB/s). The x-axis represents the data size in kilobytes, ranging from 1 KB to 2 KB, while the y-axis represents the throughput in MB/s, ranging from 0 to 9000 MB/s.

Compared to PPN-gpu-offload, PPN-gpu-sb and PPN-gpu-sb-adam show higher throughput for most data sizes. However, PPN-gpu-sb-adam-asyncq outperforms all other configurations, especially for larger data sizes, demonstrating the effectiveness of asynchronous data transfers.

Throughput Increase by Using AOK vs SOK in a PPN

The figure indicates that communication which supports overlapping of data transfers with computation by taking advantage of GPU streaming concepts and asynchronous data transfers. This leads to the introduction of an efficient stream buffer design for host-accelerator channels. Second, we leveraged the stream buffer design to introduce support for asynchronous kernel offloading in the PPN. The extension of the PPN execution model with support for streaming and asynchronous kernel offloading results in a model-driven approach for overlapping data transfers and computation on heterogeneous platforms with accelerators.