An Efficient Stream Buffer Mechanism for Dataflow Execution on Heterogeneous Platforms with GPUs

Ana Balevic
Leiden Institute of Advanced Computer Science
University of Leiden
Leiden, The Netherlands
balevic@liacs.nl

Bart Kienhuis
Leiden Institute of Advanced Computer Science
University of Leiden
Leiden, The Netherlands
kienhuis@liacs.nl

ABSTRACT

The move towards heterogeneous parallel computing is underway as witnessed by the emergence of novel computing platforms combining architecturally diverse components such as CPUs, GPUs and special function units. We approach mapping of streaming applications onto heterogeneous architectures using a Process Network (PN) model of computation. In this paper, we present an approach for exploiting coarse-grain pipeline parallelism exposed by a dataflow graph and describe its mapping onto CPU-GPU architecture. First experimental results conducted on a Tesla C2050 GPU indicate that use of a dataflow model on heterogeneous platforms not only enables exploiting different forms of parallelism (such as task, pipeline and data parallelism), but also has a potential to become an effective solution for reducing I/O overheads.

1. INTRODUCTION

The ongoing move towards parallel computing is well underway driven by the capabilities and limitations of modern semiconductor manufacturing. Emerging embedded and HPC platforms combine architecturally diverse components, such as CPUs, GPUs, and FPGAs. On the one hand, the heterogeneous architectures offer enormous potential to exploit different forms of parallelism, such as data and task parallelism. On the other hand, they present numerous challenges for application developers. To take advantage of available processing power and parallelism, they must be carefully partitioned for parallel processing, mapped with in-depth architecture considerations, and written in a scalable fashion.

Recently, there has been a large amount of activity in the area of automatic-parallelization, resulting in significant advances in generation of highly efficient, data parallel codes for individual kernels [3, 4]. As an evolutionary step toward full program acceleration, we are interested in the mapping of applications composed of several functions with different compute requirements onto heterogeneous platforms.

We approach this mapping by exploiting the Kahn Process Network (KPN) model of computation, which allows structuring of an application as a network of concurrent processes communicating over channels. As an illustration, we show a KPN model of a streaming video application in Figure 1, where nodes represent processes and edges represent channels. This process network was automatically obtained from the sequential application code using the Compaan compiler [7]. Due to clear separation of communication and computation, Kahn Process Networks (KPN) and its dataflow variants [8] gained large acceptance for representation of streaming applications.

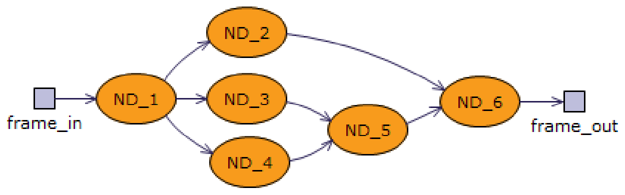


Figure 1: A Polyhedral Process Network (PPN) Model of a Streaming Multimedia Application.

Many real-world applications which are traditionally accelerated on FPGAs or GPUs can be represented in a dataflow model. Figure 2 shows an example hardware platform involved in acceleration of X-ray processing in medical domain. The structure of this processing solution involves data acquisition on an external device, pre-processing on the host, acceleration on the GPU, and visualization. The arrow in Figure 2 shows the path necessary for data to traverse in order to be processed on the GPU, exposing the data movement as one of the main challenges for efficient application execution on a heterogeneous platform.

In this paper, we address the mitigation of data I/O on the performance of applications accelerated on heterogeneous platforms. Our approach is based on the principle of *asynchronous execution*, which provides for high flexibility and scalability due to decoupling of computation via dataflow. We also show how to exploit *coarse-grain pipeline parallelism* in a PN model by leveraging advanced mechanisms for CUDA stream processing and concurrent execution. These mechanisms are fully supported on the latest generation GPUs, such as Tesla C2050 GPU. We also introduce a *Stream Buffer* mechanism which enables pipelined execution between the CPU and the GPU. As a proof of concept, we

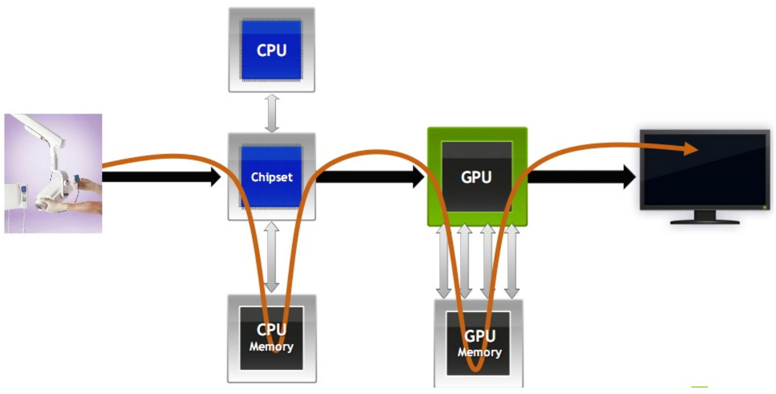


Figure 2: High Performance Heterogeneous Systems for Streaming Applications

implemented a multi-threaded prototype framework, where we successfully integrated POSIX Threads and CUDA 4.0 into a single application running on a work station featuring a 4-core AMD Phenom II CPU and a Tesla C2050 GPU.

The paper is structured as follows: In Section 2, we give an overview of the related work. In Section 3, we present the process network model, its mapping onto a heterogeneous platform, and the stream buffer concept. In Section 4, we present the first experimental results and in Section 5, our conclusions.

2. RELATED WORK

The architectural diversity of emerging hybrid platforms has been reflected in a large number of tools, programming models, and APIs for programming these architectures. These APIs are rarely designed for compatibility, and require a significant effort when developing high-performance cross-platform applications by hand.

Automatic parallelization of sequential programs for parallel architectures has acquired much research attention lately. The PLuTo framework [2, 3] explores the hyperplane tiling approach for coarse-grain data parallelization and locality optimization on multi-core CPUs and first-generation GPUs. In the CHiLL framework, the authors generate a parallel solution using classical compiler optimization techniques for locality and data parallelism and perform a parameter space search to optimize the generated code [4], with resulting codes matching performance of hand-tuned libraries.

Apart from speeding-up the computation, the overall application execution time can be seriously affected by data I/O. The I/O transfers are one of the major bottlenecks and can possibly diminish benefits from the acceleration itself [6], thus making orchestration of data transfers a highly relevant problem. To mitigate this issue, several advanced streaming solutions have been proposed by NVIDIA [9]. Following this approach, programmers use a predefined code pattern which enables overlap of computation and communication. Although powerful, this approach requires a custom-made solution for each application and may require careful load balancing to achieve efficient overlap.

An alternative approach to obtain computation and communication overlap is to exploit dataflow models of computation. For example, the dataflow language StreamIt demonstrated efficient execution on a multicore platform [5]. For a specific class of SANLP applications, the PPN model can be obtained automatically using the Compaan tool [7]. The PPN model represents naturally task and pipeline parallelism available in the source code. In [1], we extend this approach with support for data parallelism.

3. DATAFLOW EXECUTION ON HYBRID PLATFORMS WITH ACCELERATORS

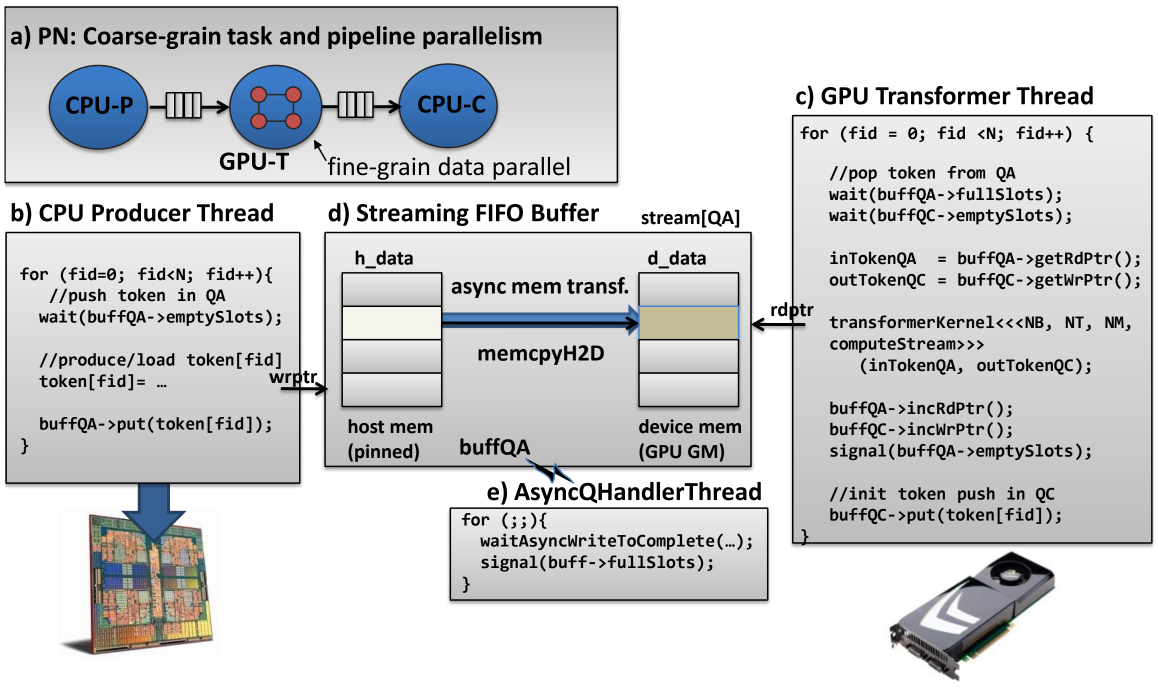
A KPN model describes an application as a network of concurrent autonomous processes that communicate over unbounded FIFO channels. A subclass of KPNs are Polyhedral Process Networks (PPNs) [10] that can be automatically derived from a class of sequential nested loop programs using modern compiler techniques [7].

In a PPN, a process is generated for each nested loop body statement of the application being investigated. The iteration space of the statement is used to construct the node domain of a process. A process is connected to other processes via channels. Channels are used for passing data tokens between processes. The tokens correspond to the input and the output arguments of the function executed by a process. The PPN processes are blocked on input ports until data becomes available. Once fired, a PPN process proceeds to execute the next iteration point from its node domain.

A PPN of a typical streaming video application, such as the one given in Figure 1, consists of multiple kernels connected in a dataflow fashion. All communication between processes is specified by the dataflow graph and all processes in Figure 1 execute autonomously. This property enables mapping of PPNs nodes onto different architectural components and exploration of diverse mappings.

3.1 Asynchronous Execution Model

For mapping onto a heterogeneous CPU+GPU system, we built a prototype framework following the asynchronous execution model of a PN in which we use the pthread library for managing the work of a multi-core CPU and CUDA 4.0 API for accessing the GPU. By default, we map each PN node (i.e. task) to a single thread of execution. For example, the PN model shown in Figure 3a consists of three processes connected in a simple pipeline. We make the following mapping assumption: the *producer* process is executed by the CPU-P thread running on the CPU, the *transformer* process is executed by the GPU-T thread running on the GPU, and the *consumer* process is executed by the CPU-C thread also running on the CPU. Each PN thread executes



```
for (fid=0; fid<N; fid++){
   //push token in QA
  wait(buffQA->emptySlots);

  //produce/load token[fid]
  token[fid]= …

  buffQA->put(token[fid]);
}
```

```
for (fid = 0; fid <N; fid++) {

   //pop token from QA
   wait(buffQA->fullSlots);
   wait(buffQC->emptySlots);

   inTokenQA  = buffQA->getRdPtr();
   outTokenQC = buffQC->getWrPtr();

   transformerKernel<<<NB, NT, NM,
   computeStream>>>
      (inTokenQA, outTokenQC);

   buffQA->incRdPtr();
   buffQC->incWrPtr();
   signal(buffQA->emptySlots);

   //init token push in QC
   buffQC->put(token[fid]);
}
```

```
for (;;){
  waitAsyncWriteToComplete(…);
  signal(buff->fullSlots);
}
```

Figure 3: Data-Driven Execution: Asynchronous Processing + Stream Buffer Communication.

asynchronously and loops over a series of tokens, with each token representing a single frame in the running example. Execution of a PN process is structured into the following phases:

- *stream data in - blocking read*
- *computation - function execution*
- *stream data out - blocking write*

The computation phase corresponds to the execution of a function enclosed in the loop nest from which the PN process was generated. The function can be executed "as is", following the lexicographical order in the application source code, or it can be further transformed using polyhedral techniques [1–3] for optimization purposes (e.g. locality) or to support execution on a specific target architecture, such as GPU.

In the running example, the producer and consumer functions are left in their original form, while we transformed the *transformer* function for fine-grain data parallel execution on the GPU. The iteration domain of the PN transformer node is simply mapped 1-1 onto the N-Dimensional range (execution configuration) of the CUDA kernel, resulting in each iteration point being assigned to a different CUDA thread. The computation function in the GPU-T thread is executed on the GPU as *transformerKernel* by a large number of lightweight CUDA threads in parallel. The CUDA threads are structured into NB thread blocks composed of NT threads. Kernels launched by different PN processes are issued into different CUDA streams and can be executed in parallel if the target device supports concurrent kernel execution as is the case on Fermi architecture GPUs.

3.2 Design of a Stream Buffer

To address the data I/O between GPU and host, we use dataflow to exploit pipeline parallelism on a heterogeneous platform. We designed a *Stream Buffer (SB)* mechanism which allows data transfers between host and device to occur concurrently with computations.

The Stream Buffer mechanism is used to implement channels in the PPN model. In order to minimize the data movement, we designed the SB on the basis of a Circular Buffer (CB). The salient feature of the CB is that it is not necessary to move data around on a read/write operation, but only pointers. This feature makes it especially interesting for FIFO buffer implementations on shared memory systems where the cost of data movement can easily outweigh the computation time.

We designed a SB for a *single producer - single consumer (SPSC)* case, as all communication in a PPN is point-to-point. The SPSC property allows us to design a mechanism in which producer and consumer can advance at their own pace and read/write data concurrently as long as there are full/empty buffer slots available. A PPN producer accesses a SB using a blocking write mechanism, i.e. the *put* operations will block the producer on the semaphore buff->emptyCount until there is an empty slot in the buffer. Similarly, a PPN consumer will block on the semaphore buff->fullCount until there is a token available.

The Stream Buffer implementation is chosen on the basis of the hardware architectures on which the threads implementing PPN producer and consumer processes execute. Figure 3 shows a design of a host-to-device (H2D) Stream Buffer, where the producer runs on a CPU and the consumer runs on the GPU. We implemented the SB using a distributed memory approach with double buffering. For example, for buffQA shown in Figure 3d, we allocate slots in the host memory (h_data) and slots in the global memory of the GPU (d_data). The host buffer slots are allocated in non-pageable (pinned) host memory, as required for streaming in CUDA.

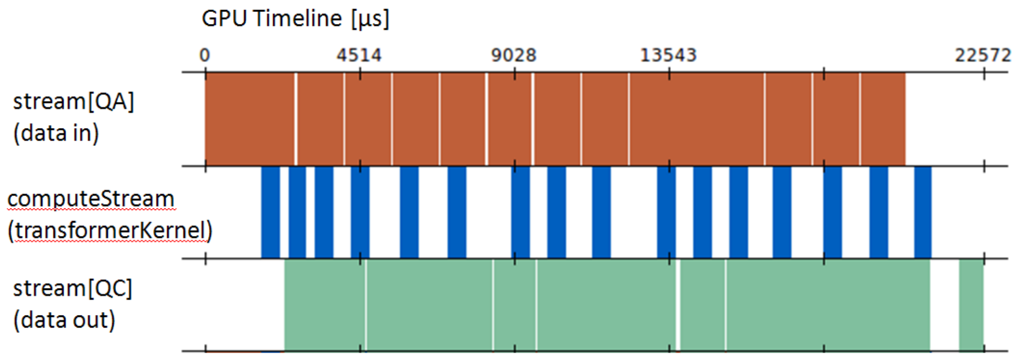


Figure 4: A Dataflow Approach: Overlapping of Kernel Execution and Memory Transfers on Tesla C2050 GPU.

The communication between host and device buffer slots is implemented via asynchronous memory transfer over a PCI Express bus using the two DMA engines on the GPU. The synchronization between the PPN producer driving the CPU operations and the PPN consumer driving the GPU operations requires additional mechanisms for CPU-GPU synchronization. For this purpose, we associate an *AsyncQWriteHandler* thread to each asynchronous Stream Buffer. The AsyncQWriteHandler monitors the asynchronous memory transfer between host and device and signals data availability to the PPN consumer. This is illustrated for buffQA in Figure 3e: as soon as a memcpyH2DAsync is completed, the AsyncQWriteHandler for buffQA signals that a new buffer slot is written to the GPU-T thread (Figure 3c), which then launches the CUDA kernel.

4. PRELIMINARY RESULTS

To validate the benefits of the coarse-grain task and pipeline parallel approach, we developed a multi-threaded PPN prototype using a Pthreads library. We integrated Pthreads and CUDA into a single project that runs on a heterogeneous platform and designed a minimal GPU-proxy that loads the work to GPU and launch asynchronous memory transfers directly from the Pthread code. Performance was measured on a machine with an AMD Phenom II X4 965 Processor CPU and 1xNVIDIA Tesla C2050 GPU, PCIe 2.0, with 448 cores and 2 asynchronous copy engines. The Tesla C2050 GPU supports overlapping of GPU kernel execution and two host-device data transfers (in different directions). We run a synthetic producer-transformer-consumer streaming application, for 100 frames, frame size 1M data elements, 4-slot FIFO stream buffers, as described in the previous section. The producer and consumer PN nodes run on the CPU, and the transformer node was assigned to the GPU.

The GPU execution timeline in Figure 4 obtained by CUDA Profiler shows data transfers and kernel executions on the Tesla C2050 GPU in the streaming mode. The producer thread generates tokens into stream buffer QA. The buffQA tokens that are transfered from host to GPU are enqueued in stream[QA]. The computeStream is used for computation, and shows kernel launches for transformation of input QA tokens into output QC tokens in the transformer thread. The stream[QB] is used to download the output tokens into CPU-side buffQC. As it can be seen in Figure 4, using asynchronous execution model and stream support, we create an I/O transfer and computation pipeline: while frame $fid = k$ is uploaded to the GPU from host memory, previous frame $fid = k - 1$ is processed by the kernel running on the GPU, and frame $k - 2$ is downloaded to the host using the second DMA engine of Tesla C2050.

Preliminary results show that synchronization overheads are rather low and that a good overlap of memory transfers and computation can be achieved using this approach. As next steps, we plan to experiment with execution of multiple threads on the GPU, load balancing and OpenCL command queue model.

5. CONCLUSION

We present an approach based on a Polyhedral Process Network model of computation for automatic mapping of sequential applications onto heterogeneous architectures. Our approach exploits coarse-grain task and pipeline parallelism in streaming applications, as well as fine grain data parallelism within tasks. We designed a stream buffering scheme for a heterogeneous platform that supports efficient pipelined execution. We implemented a multi-threaded framework prototype using Pthreads library and CUDA4.0. We leveraged the CUDA stream concept to enable asynchronous execution on the GPU side and overlap communications and computation. Preliminary results show that a good overlap of memory transfers and computations can be achieved using the PPN model of computation. This indicates that dataflow models have large potential for efficient and automatic mapping of applications onto heterogeneous architectures.

6. ACKNOWLEDGMENTS

We would like to thank **Cliff Woolley** for his invaluable help and expertise on advanced CUDA topics. We also very much appreciated the support of NVIDIA Corporation in donating a Tesla C2050 GPU for our research.

7. REFERENCES

[1] A. Balevic and B. Kienhuis. A Data Parallel View on Polyhedral Process Networks. *SCOPES'11*.

[2] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine

programs. In *Proc. of Compiler Construction (CC 2010)*. Springer, 2010.

[3] U. Bondhugula et al. PLuTo: a practical and fully automatic polyhedral program optimization system. In *Proc. of PLDI'08, Tucson, AZ, 2008.*

[4] C. Chen, J. Chame, and M. Hall. CHiLL: A framework for composing high-level loop transformations. *U. of Southern California, Tech. Rep*, pages 08–897, 2008.

[5] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–162. ACM, 2006.

[6] C. Gregg and K. Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144. IEEE.

[7] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *Proc. of CODES'00*, pages 13–17. ACM, 2000.

[8] E. A. Lee and T. M. Parks. Dataflow Process Networks. *Proc. of the IEEE*, 83(5):773–801, 2002.

[9] NVIDIA Corp. Maximizing GPU Efficiency in Extreme Throughput Applications. Technical report, Sept. 2009.

[10] S. Verdoolaege. Polyhedral Process Networks. *Handbook of Signal Processing Systems*, pages 931–965, 2010.