KPN2GPU: An Approach for Discovery and Exploitation of Fine-Grain Data Parallelism in Process Networks

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
With advances in manycore and accelerator architectures, the high performance and embedded spaces are rapidly converging. Emerging architectures feature different forms of parallelism. The Polyhedral Processes Networks (PPNs) are a proven model of choice for automated generation of pipeline and task parallel programs from sequential source code, however data parallelism is not addressed. In this paper, we present a systematic approach for identification and extraction of fine grain data parallelism from the PPN specification. The approach is implemented in a tool, called kpn2gpu, which produces fine-grain data parallel CUDA kernels for graphics processing units (GPUs). First experiments indicate that generated applications have a potential to exploit different forms of parallelism provided by the architecture and that kernels feature a highly regular structure that allows subsequent optimizations.

1. INTRODUCTION
Acceleration of data intensive, highly regular computations is rapidly gaining visibility in embedded systems used in the mobile, automotive, health care, and other fields. Both in HPC and embedded spaces, acceleration of programs by exploiting parallelism requires expert parallel programming knowledge, high time investment (ensuring correctness and maintainability) and source code modifications. Automatic compilation of sequential programs for parallel architectures is a challenging problem which attracts much research attention [1, 2, 4, 11, 13, 16]. While most automated approaches are designed around a single form of parallelism, emerging heterogeneous architectures make use of different forms of parallelism. As a step towards exploring a combination of task, data and pipeline parallelism, we extend a state of the art framework for automated generation of task and pipeline parallel programs [11] with support for data parallelism. More specifically, in this paper we address automatic identification of data parallelism, many-threaded processing and mapping on GPUs.

Our approach to automatic parallelization is based on a process networks model of computation. Due to clear separation of communication and computation, the Kahn Process Networks (KPN) and its variants gained large acceptance in embedded systems design [12]. A Polyhedral Process Networks (PPN) model has been extensively used for automatic generation of efficient pipeline and task parallel programs for a given target architecture (e.g. x86, FPGAs) and programming model (e.g. PThreads, SystemC, etc). A frequent parallelization target are nested loop programs (NLPs). For a class of static affine NLPs, the PPN specification can be automatically derived using tools such as Compaan [11]. An example of an automatically generated PPN model is given in Fig 1b for the simple sequential program in Fig 1a. Each node of a PPN is processed by a single thread of execution, which scans the iteration points of the corresponding node domain sequentially, as shown by the arrows in Fig 1c for node ND2. Although the PPN model is well suited to represent inherent concurrency of an application, it currently does not capture data parallelism.

To make use of data parallel accelerators, such as GPUs (Fig 1d), it is required to have a very large number of fine-grain threads working in a data parallel manner [15]. The Compute Unified Device Architecture (CUDA) allows programmers to write general purpose programs for graphics cores in C code. CUDA programs are designed in a thread-centric manner and impose a logical program organization onto a structured hierarchy of threads. The data parallel representation of the function in node T in Fig 1e is better suited for execution on the GPU cores. However, the differences be-

---

**Figure 1:** Sequential processing of a process network node vs. data parallel processing on GPU.

For (a) C Application:
```c
for(i=1; i<=M; i++)
for(j=1; j<=N; j++)
T: a[i,j]=transform(a[i,j]);
C: consume(a[i,j]);
```

For (b) PPN:
```
P: node ND1
T: node ND2
C: node ND3
```

For (c) ThreadT:
```
sequential execution
```

For (d) GPU Architecture:
```
P: processor
T: thread block
C: multiprocessor
```

For (e) data parallel execution
tween the single-thread PPN node processing (Fig 1c) and the many-threaded processing (Fig 1e) makes targeting data parallel accelerators, such as GPUs, challenging.

In this paper, we present a systematic approach for automatic identification of fine-grain data parallelism in the PPN specification of an application. We leverage the formal concurrency representation and techniques for dataflow analysis and scheduling in the polyhedral model. We extend the Compaan framework, which automatically converts C code into a PPN representation and offers task and pipeline parallelism, with support for automated extraction of data parallelism. As a proof of concept, we show how to identify and exploit data parallelism when mapping a PPN onto an accelerator such as GPU. The paper is structured as follows: In Section 2, we give a short overview of the PPN model of computation. In Section 3, we describe related work. Section 4 gives the problem definition. In Section 5, we present the solution approach. Section 6 shows first order results on an NVIDIA GPU. In Section 7, we present our conclusions.

2. POLYHEdRAL PROCESS NETWORKS

A Kahn Process Network (KPN) model of computation was introduced in the 70s by G. Kahn for modeling distributed systems [10]. It describes an application as a network of concurrent autonomous processes that communicate over channels. A Polyhedral Process Network (PPN) is a restricted form of a KPN, where the function executions and the specification of input and output arguments are described as polytopes obtained by polyhedral analysis of nested loop programs [1, 17].

In a PPN derived by Compaan, a process is generated for each loop nest encapsulating a statement or a function call. Hence, in Fig 1c we see three nodes in the PPN as there are three statements in the sequential program ($P$, $T$ and $C$). The iteration space of the function call is associated to a PPN process as a node domain, which is a polytope containing all iteration points in which the function is executed (also referred to as operations). In Fig 1c, we show the iteration points for the function in statement $T$ in space ($i$, $j$). A PPN process executes operations of the node domain following the sequential schedule of the original loop nest, as shown in Fig 1c. Each PPN node has a number of input and output ports, which correspond to domains of the function arguments and return values. At each iteration point execution, a process function reads/writes data over ports from/to different channels. A self-link corresponds to a channel that connects an input and an output of the same process and it captures dependencies between the statement’s loop nest, while an external link captures dataflow dependencies between different statements. The relationship between iteration points of the inputs and the outputs is specified by a mapping matrix, which is the result of exact data dependency analysis. The current state of the art allows a PPN of an application to be automatically derived for static and weakly-dynamic affine NLPs. Due to its characteristics, such as a clear separation of communication and computation, the PPN model has large acceptance in embedded systems design.

3. RELATED WORK

Many languages and APIs have been designed to simplify parallel programming. Working directly on a vendor’s APIs for a specific architecture allows implementation of highly optimized codes. However, when a single code base and acceleration for different target architectures are required, auto-parallelizing compilers may be a fast and interesting alternative.

Our approach for identification of data parallelism in PPN nodes is inspired by the experimental polyhedral framework LooPo and the work of Lengauer and Feautrier on automatic parallelization [8, 13]. While LooPo implements scheduling algorithms, it provides only basic code generation capabilities. We take this approach further by using the scheduling algorithms implemented in LooPo to identify data parallel operations in PPN node domains and apply additional transformations to PPN nodes to make mapping onto real-world architectures possible.

Some of the challenges in using polyhedral techniques to automatically derive code for GPUs are discussed in [2]. Their work also attempts to integrate state of the art polyhedral framework PLuTo in polyhedral compiler collection (PoCC). The PLuTo framework [4, 5] provides an excellent tiling approach for coarse-grain parallelization and locality-optimization. In this paper, we explore an alternative approach that derives maximal fine-grain parallelism first and allows building up towards high performance kernels using architecture-specific optimization tools, e.g. such as those in [18]. The PPN model is derived from PRDG (used in [2, 4, 5, 13]) and requires a number of additional problems to be addressed, such as adaptation of ports and links from a single-token read/write to parallel shared memory accesses. In return, the PPN model provides the opportunity to exploit not only data parallelism, but also task and pipeline parallelism.

Our mapping to shared memory architectures is reminiscent of the CRP language proposal [9]. In CRP, communication channels are also mapped onto memory arrays, however each array element has an associated one-bit flag to indicate if the element value has been written. We also follow a single-writer approach, but use events to signal data availability and organize communication in a coarser-grain manner to enable efficient mapping onto modern architectures, such as GPUs. Finally, CRP requires hand-written specification of network components by the programmer, which may make specification of exact dependencies between processes a hard manual task, while a PPN model can be obtained automatically from a sequential application [11, 16].

In Compaan [11], the PPN model is the result of data flow analysis and features all task and pipeline parallelism available in the source code, but data parallelism is not considered. The plane-cutting technique in [14] makes a first step towards exploring data parallelism in PPNs by duplicating nodes. This approach is well suited for mapping onto multicore architectures using coarse-grain threads, however full unrolling of a domain to extract fine grain data parallelism results in a node explosion, which we avoid by introducing a per-node scheduling function.
4. PROBLEM STATEMENT

The PPN model has been extensively used for automatic generation of efficient pipeline and task parallel programs for a given target architecture (e.g. x86, FPGAs) and programming model (e.g. PThreads, SystemC, etc). However, data parallelism is not explicit in the PPN model and is currently not exploited. Moreover, each PPN node is processed by a single thread that sequentially executes iteration points in the loop nest. An example is given in Fig 1: the source code of the application (Fig 1a) is transformed into a process network representation (Fig 1b) composed of three processes: producer P, transformer T and consumer C, connected via channels. Fig 1c shows a detailed view of the transformer process: the complete iteration domain of node T is processed by a single thread of execution. The thread scans the node domain one iteration point at a time, following the lexicographical order as specified in the source code. On the other hand, architectures such as the GPU depicted in Fig 1d require a large number of threads working in a data parallel manner, as shown in Fig 1e. Processing a PPN using only a single thread of execution per node does not exploit GPU resources at all. The main questions that we address in this paper are: (1) automatic identification of data parallelism in PPN networks, (2) extension of the single-thread-per-node model of PPN processing to many-threads-per-node processing, where many threads can work cooperatively or independently on the node domain in a data parallel manner, and (3) efficient mapping onto architectures featuring data parallelism.

5. SOLUTION APPROACH

Our proposal for identification of data parallelism in PPNs is shown in Fig 2. The input to the transformation process is a PPN model of a sequential application, as produced by the Compaan tool [11]. Our approach for identification of data parallelism in PPN nodes is rooted in exact dependence analysis. Data dependencies between operations (statements executed in iteration points) are the main source of causal constraints on the execution order. By analysing data dependencies in a PPN node domain, we identify the operations that are independent and can be executed in a data parallel manner. In order to make our approach scalable, we perform the analysis and transformations for data parallelism on each PPN node separately. We use the PPN network specification as a basis for generation of the complete application for a target architecture.

The transformation process in structured in the following steps: In Phase I, we extract the exact specification of data dependencies within each PPN node. In Phase II, we identify sets of concurrent operations and their partial order using scheduling techniques in the polyhedral model. The schedule and its matching allocation allow us to derive a time-space mapping(s) for a PPN node. In Phase III, we perform a mapping of the transformed PPN nodes onto a target architecture. As an illustration, we show an example mapping onto the two-level CUDA architecture for GPUs. To enable data parallel processing, we transform single token reads/writes to multiple data element loads/stores per transaction, and map PPN channels onto linear arrays. Each PPN process is transformed into a CUDA kernel. Finally, we generate a CUDA host program which orchestrates kernel execution and data exchange according to the dataflow in the PPN network.

5.1 Phase I: Extraction of Data Dependencies

A PPN node is represented by a polyhedron that contains all iteration points at which a node’s computation function is executed. In addition, each node contains a set of input and output ports, which are also specified by polyhedrons. The polyhedral model is used to represent and manipulate the loop nest structure around the program statements. The popularity of the polyhedral model for compiler analysis is based on the existence of a rich mathematical theory that supports correctness-preserving transformations and quantification of results [13]. Dataflow analysis [7] makes identification of exact data dependencies between operations (iteration points) in an iteration domain possible.

A PPN contains information on data dependencies by design - dependencies are the result of exact dataflow analysis step during the PPN construction. The specification of a channel captures the exact mapping function between producer and consumer iteration points of a given dependence. A domain and a codomain of a dependence relationship are specified as polyhedrons associated with the output port of the producer node and the input port of the consumer node. In this paper, we are interested to discover sources of fine-grain data parallelism, i.e. we need to find independent executions of the same operation. The natural candidates are repeated executions of the same node (i.e. the same function). In a PPN, self-links on a node indicate repetitions in the function execution. Each self-link describes a single intra-node dependency. At this stage, we perform a simple analysis on the PPN network and extract data dependence specification from node and channel specifications. The result of the analysis is a dependence specification for every PPN node. The per-node dependence specification contains node domain and self-links descriptions in the polyhedral model.
5.2 Phase II: Identification of Data Parallelism

The per-node dependence specification is used to identify independent operations in Phase II. Independent operations are the source of data parallelism, i.e. a form of parallelism where the same operation is executed on different data elements at the same (logical) time step. Scheduling a program is a well known strategy for assigning logical execution times to operations and one of the most powerful methods for auto-parallelization. A schedule \( t \) is a function that assigns a time step to each iteration point \((i,j)\) of an iteration domain, while preserving data dependencies. While different algorithms can be used to identify sets of concurrent operations and find their partial order [6], we demonstrate our approach using a simple, yet powerful algorithm of Feautrier.

In [8], Feautrier presented an algorithm for finding schedules as piecewise affine functions of iteration vectors. However, straightforward application of this algorithm to whole programs leads to high computational complexity (proportional to the program size) and code examples often break. By leveraging the distinguishing feature of PPNs, i.e. a clear and well-defined separation of communication and computation between processes, we mitigate the problem by applying the scheduling algorithm on each node separately. By construction, each PPN node can be described by a perfectly nested loop, i.e. its node domain is always a simply structured polytope. This enables us to split a program into smaller, well-structured, scheduling instances. Experiments show that in most cases, the Feautrier’s algorithm is able to find a solution for these structures, irrespectively of the number of nodes in a PPN graph, thus providing a scalable solution.

The result of scheduling on a node dependence specification derived from the PPN, is a time-optimal schedule for a node domain. The time-optimal schedule (minimal latency) corresponds to maximal fine-grain data parallelism. The time-optimal schedule and allocation function define a polyhedral time-space mapping, which is used to transform a (source) node domain into a target domain in \((p,t)\) space, where \(p\) denotes an abstract processing unit and \(t\) denotes a logical time step. The scheduling of nodes with different types of data dependencies is illustrated on two examples:

Figure 3: Predictor Node T: (a) Data dependencies, (b) Affine schedule, (c) Target node domain.

\(\text{Predict} (\text{Fig 3})\) computes a new pixel value in a 2D-image on the basis of its two neighbours. The computation function in the PPN node \(T\) for the predictor example has the form \(A[i][j] = \text{pred}(A[i-1][j], A[i][j-1])\), with \(\text{pred}\) being the transformation function. Data dependence flow is depicted in Fig 3(a). In Fig 3(b) dependencies are omitted, but now lines are used to connect all iteration points that could be executed at the same time step, as found by the scheduling algorithm. The scheduler identified a 1D schedule \(t(i,j) = i + j - 2\) and a 1D allocation function: \(p(i,j) = j\), which form a time-space mapping \(T\). The time-space mapping \(T\) is used to transform the node domain into the target domain in Fig 3(c). The producer-consumer relationship that arises from data dependencies imposes cooperative parallelism, since data consumed by one thread may be produced on a different processing element, thus synchronization and coordination are necessary.

\(\text{Grid} (\text{Fig 4})\) computation function in the PPN node \(T\) has the form: \(a[i+j] = \text{grid}(a[i+j])\), where \(a\) in this example is a 1D-array of data elements, and \(\text{grid}\) is the transformation function with data dependencies as depicted in Fig 4(a). As can be seen in Fig 4(b), all operations along the lines can be executed independently of each other one time step at a time. The Grid is characterized by a composite 1D schedule and 1D allocation function. Producer-consumer relationship is sync-free and can be realized by independent parallelism. The schedule is a combination of two elementary affine scheduling functions:

\[
t(i,j) = \begin{cases} 
 i-1, & \text{if } i + j \leq 5; \\
 -j + 4, & \text{otherwise.} 
\end{cases}
\]

The composite schedule requires division of the PPN node domain into sub-domain based on the schedule condition \(i + j \leq 5\), as shown in Fig 4(b). Each of sub-domain is transformed using the matching scheduling function. This results into two target sub-domains in Fig 4(c), which are independent and can be executed concurrently.

5.3 Phase III: Mapping to Thread Hierarchy

In the previous section, we showed how to identify fine grain data parallelism in the PPN nodes, and how to transform domains of PPN nodes from \((i,j)\) into a \((p,t)\) space representation using polyhedral scheduling and allocation techniques. For execution on a real-world architecture, such as GPU, the abstract processing elements \(p\) have to be mapped on multiple threads. In this section, we show how to transform the \textit{Single-Thread per Node} mode of processing PPN nodes (Fig 1c) into \textit{Many-Threads per Node} processing (Fig 1e) and how to map processing elements onto a multi-level thread hierarchy, on the example of CUDA programming model for GPUs. CUDA features a structured hierarchy of
threads, where threads are organized into thread blocks, and thread blocks are organized into a computation grid. The following constraints apply: communication between threads in a thread block is allowed through the shared memory assigned to the thread block, while thread blocks must be executed independently.

As an illustration of mapping to CUDA, let us consider the predictor example in Fig 5. The transformed node domain (Fig 3(c)) in \((t, p)\) space contains cooperative parallelism and naturally maps to execution of iteration points by threads in a thread block. At each time step of a CUDA kernel, the threads of a thread block (Fig 5 (b)) process a set of independent iteration points in parallel, as shown by bounded boxes that enclose all iteration points assigned to the same time step in Fig 5(c). Processing of an iteration point by a GPU thread involves loading function arguments from the GPU memory hierarchy, computing the result, and storing the result at the destination location in memory. Thus, each discrete time step in a CUDA kernel corresponds to the execution of multiple instructions on the GPU by threads active in that time step. For communication between threads we do not rely on FIFOs, but use linear memory arrays as is common in CUDA applications.

Threads in a thread block can work cooperatively or independently to process a node domain. Coordination is required between time steps when data produced in one step is used in a successive time step as input argument by another thread. An example is shown in Fig 5(c): to produce the value of data element \(A_{23}\) at time step \(t = 3\), the thread corresponding to \(p = 3\) requires data produced by threads \(p = 2\) and \(p = 3\) at \(t = 2\). For implementing cooperative parallel execution, we used shared memory buffers for communication and synchronization primitives.

The result of this phase is a model called Data Parallel View (DPV) on PPNs [3], which introduces abstractions required to capture and exploit data parallelism on top of the PPN model. The DPV model exposes identified data parallelism, adopts the PPN components for mapping onto shared memory architectures, and enables generation of data parallel source code for different parallel programming models. A simple, functionally correct CUDA kernel can be obtained by traversing the data model behind DPV.

To address resource utilization on the GPUs, it is necessary to have large number of independent thread blocks executing concurrently. We approach this problem in two ways: First, if concurrent kernel execution is supported by the target architecture, we map PPN nodes into different streams in order to leverage the task parallelism within the PPN network. Use of CUDA shared memory model requires synchronization between memory accesses. Inter-node synchronization is achieved by letting each PPN node run to completion and using CUDA events to signal data availability. Second, for large values of domain parameters we perform top-level partitioning of the PPN domains and map partitions to thread blocks. Formalization of partitioning in PPN networks is a topics of our current research work.

6. RESULTS AND DISCUSSION

The approach described in the previous section is implemented as an extension of the Compaan framework called kpn2gpu. It includes a CUDA backend and enables automatic generation of functionally correct kernels featuring maximal fine-grain data parallelism. First order results indicate that we can effectively extract fine grain data parallel kernels from PPN nodes and map it on the GPU architecture.

In Fig 6, we present a speedup as a ratio of processing time using sequential code and processing time using many-threads to process a node. The comparison in Fig 6 is given for the case when a single thread runs on a CPU core (AMD Phenom II X4 965 CPU), and processing by many threads is performed on a Fermi architecture GPU (NVIDIA Tesla C2050). To illustrate the impact of arithmetic intensity on the performance, we included traces for a different number of 32-bit integer multiply-add (MAD) operations per function execution, since the amount of computation per function is not prescribed by the PPN model.

We observed a significant difference in the execution time of grid and predictor examples. Neither of the examples is embarrassingly parallel, i.e. there are dependencies between different time steps of the same thread block, as well as dependencies between different threads blocks. As the later is not allowed on CUDA, we circumvent this constraint by multiple kernel invocations in order to satisfy dependencies. This allows us to execute only independent thread blocks at each kernel.
invocation, but introduces large overheads. The average kernel execution time for 1M iteration points is approx. 1.5ms for the grid kernel and approx. 8ms for the predictor kernel. The first difference between the two kernels comes in the compute-to-global memory access ratio, with predictor requiring 2x more global memory loads per output data element. Second, grid kernel exhibits more data parallelism and requires fewer time steps to complete. Due to the simple producer-consumer relationships in the target domain of the grid example, it can be processed by independent parallel threads. The predictor kernel requires global or shared memory for internal communications and synchronization at each time step. For the grid example we generate a kernel in which each thread works only on its private data that is stored in registers and without synchronization between time steps of the schedule. Experiments indicate that significant improvements can be achieved by standard GPU optimizations such as memory access coalescing, prefetching and thread block coarsening. Kernels generated by kpn2gpu exhibit a highly regular structure, which we plan to leverage for future architecture-specific tuning.

7. CONCLUSION

In this paper we presented a three-phase approach for automatic identification of fine-grain data parallelism in a Polyhedral Process Network (PPN) specification. This work is built on top of the Compaan framework, which automatically converts C code into a PPN representation that offers task and pipeline parallelism. We introduced automatic extraction of data parallelism and showed that this methodology allows for a mapping onto many-threaded architectures. We implemented a proof of concept in the kpn2gpu tool and demonstrated a mapping to an NVIDIA Fermi-architecture GPU. First performance results show that we can effectively exploit data parallelism and map it on a massively parallel GPU. Future work includes exploration of different combinations of parallelism and tradeoffs in mapping nodes to different architectural components.

8. ACKNOWLEDGEMENTS

This work was supported by the European Commission under the CATRENE programme, project #2A718 TSAR. We would like to thank NVIDIA for their generous hardware donation.

9. REFERENCES


Figure 6: Performance results on NVIDIA Tesla C2050: Speedup Ratio (Tcpu/Tgpu) with variable arithmetic intensity.