ABSTRACT
Emerging architectures in embedded space are expected to make use of a diverse mix of multicores, vector-based units, GPU cores and special function accelerators. In order to facilitate mapping onto diverse architectures, different models of computation have been considered. Polyhedral Process Networks (PPNs) have been extensively used in automatic generation of task and pipeline parallel programs for embedded architectures. However, the single program multiple data (SPMD) type of data parallelism has not been addressed in the PPN model. In this paper, we propose a Data Parallel View (DPV) on PPNs which introduces abstractions necessary for capturing and exploiting data parallelism on top of the PPN model. As a proof of concept, we demonstrate how a PPN can be mapped onto a modern GPU using the DPV. By complementing the native PPN support for task and pipeline parallelism with the DPV support for data parallelism, we expect to make the best use of different types of architectural components and types of parallelism on heterogeneous architectures.

1. INTRODUCTION
Applications in the area of mobile computing, health care, imaging and signal processing, impose large computational demands. Recent trends in the embedded space indicate that in order to satisfy the ever increasing demand for processing, it is necessary to move towards heterogeneous architectures. Emerging architectures in embedded space are expected to feature a diverse mix of multicore processors, vector-based units, GPU cores and special function accelerators. While architectural diversity enables a designer to handle different types of computational needs with the best possible architectural matches, it also introduces different levels and types of parallelism in the already complex picture. There is a variety of domain-specific parallel programming languages with different levels of expressiveness (e.g. SA-C, StreamIt, Canals), compiler extensions and libraries at different level of abstraction (e.g. PThreads, OpenMP, Intel’s TBB, MPI) and programming models and APIs (e.g. CUDA, OpenCL, CTM) targeting different parallel architectures. However, it is neither clear how to keep track of the ever increasing number of languages and APIs, nor how to achieve code and performance portability and maintainability across different models. Automatic parallelization of sequential programs is becoming increasingly popular in the last years [1–5]. We believe that the Process Networks [6] model of computation is an excellent match when it comes to mapping sequential applications to heterogeneous architectures. In the PN model, the applications are represented as networks of concurrent processes communicating over channels. The salient feature of the PN model is a clear separation of concerns, i.e. distribution of computation and communication, which enables efficient mapping onto diverse architectures.

Figure 1: DPV as a bridge to the future embedded space featuring a diverse mix of multicore processors, vector processing units, GPU cores, special function units, etc.

Using state of the art polyhedral techniques and tools, it is possible to automatically generate a Polyhedral Process Network (PPN) [7] of an application from its sequential source code. 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, FP-GAs) and programming model (e.g. PThreads, SystemC, etc) [5, 8, 9]. Many streaming applications, besides pipeline and task parallelism, contain a lot of inherent fine-grain data parallelism. While pipeline and task parallelism have been addressed by the previous work on the PPN model, the sin-
In a PPN, a process is generated for each loop nest encapsulating a statement (a function call). Thus, in Fig 2 we see a PPN network composed of three nodes (ND1, ND2, ND3), each representing one statement (P, T, C) from the sequential code of the predictor example. In the polyhedral model, the for loops surrounding the statements are analyzed and used to construct the iteration space of the statement. The iteration space of the statement is represented as a polytope containing all iteration points, i.e. the iteration domain, in which the statement is executed. A PPN process contains a single statement, which is a pure function call, and its defining polytope, which is referred to as a node domain. When executed, a PPN process executes operations of the node domain in the sequential order given in the source code. Each process function can have a number of input and output arguments, e.g. node ND2 containing the predict function reads two input arguments and produces one output at each execution. The argument values are communicated via links (ED1 – ED5) that connect nodes in the PPN network, and are implemented as FIFO channels. Self-links, ED1 and ED3, connecting input and output ports of the same process, are used to capture the state and the transitions during the PPN process execution. Due to the data dependencies in the loop nest of the predict function, node ND2 starts reading the values of the arguments \( A[i-1][j] \) and \( A[i][j-1] \) generated by its previous executions, when \( i \geq 2, j \geq 2 \), to produce a new result. The PPN links are implemented as FIFO buffers. The data is read/written through the ports. Each port is associated with a port domain, a subset of the node domain that contains only the iteration points which use this port to read/write an argument. The producer-consumer relationship between the input and output port domains (IPDs/OPDs) of a source and a target port is specified by the mapping matrix of the channel that connects them. In polyhedral process networks, the mapping matrix is a result of the exact data dependency analysis.

2. POLYHEDRAL PROCESS NETWORKS

A KPN model describes an application as a network of concurrent autonomous processes that communicate over unbounded FIFO channels. A PPN model is a more restricted form of the KPN model which uses fixed-size FIFO buffers and blocking reads and writes for communication. The specific characteristic of the PPN model is that the executions and inputs and outputs are described as polytopes obtained by polyhedral analysis of nested loop programs [7]. We will illustrate this with a PPN of the predictor example in Fig 2 as generated by the Compaan [5] framework.

3. PROBLEM DESCRIPTION

3.1 Computation

The PPN model is typically processed using a single thread of execution per node domain. The node domain is processed sequentially. In contrast, modern data-parallel architectures, such as GPUs, have a large number of processing elements (e.g. NVIDIA Tesla 2050 features 448 streaming processors). The CUDA programming model exposes GPU resources as a hierarchy of lightweight threads. On the hardware level, threads are structured into warps (sets
of 32 threads), which are scheduled and executed together following the Single Instruction Multiple Thread (SIMT) paradigm. The distinguishing characteristic of the GPU architecture is its use of parallelism as the main mechanism for latency hiding. The GPU employs a hardware scheduler to find warps (unit of 32 threads) ready to execute while some other warps are waiting for memory accesses to complete. Thus, if there is enough parallelism in the code, the GPU can cover up for the long memory latencies by performing additional computations. Therefore, in order to map the PPN model onto the GPU, one of the key questions to resolve is how to get from the single-thread per node mode of PPN processing to the many-threaded processing on the target architecture. The PPN model already exposes task and pipeline parallelism. In this paper, we show how to expose model data parallelism contained within the PPN node domains in order to enable many-threaded data-parallel processing.

3.2 Communication
In the PPN model, processes communicate through channels, which are implemented as FIFO buffers. The PPN implementations use blocking reads and writes. On the other hand, the data parallel accelerators typically make use of a shared memory architecture. For example, CUDA features a complex, multi-level shared memory hierarchy that is managed by the programmer. Similar multi-level memory hierarchies can be also found in high performance embedded systems. The CUDA model imposes several constraints on the communication on each level, e.g. threads within a thread block can communicate through shared memory, but the thread blocks have to be independent, i.e. there is no inter-block communication allowed. In addition, on data parallel architectures, there is no hardware implementation of FIFOs, and software implementation is slow as it requires extensive synchronization. Thus, to execute the PPNs on a shared memory architecture, it is necessary not only to find a well-suited approach for data communication on shared memory architectures, but also to enable PPN ports and links to provide data values in parallel. Hence, the second question that we address in the DPV is how to design and implement support for data-parallel communication.

3.3 Contributions
The main contribution of this paper is the design of a novel Data Parallel View on PPNs, which provides the abstractions for capturing and exploiting data parallelism. We use the information captured within PPNs and state of the art polyhedral techniques, such as scheduling and tiling, as a basis for construction of the data parallel view. The DPV model exposes identified data parallelism, adopts the PPN components for mapping onto shared memory architectures, and allows generation of source code for various data parallel programming models, such as OpenCL, CUDA and Intel’s ARBB. In this paper, we address the following points:

- Introduction of novel abstractions for capturing and expressing data parallelism on top of the PPN model.
- Characterization of data parallel nodes (parallel width, parallel depth) under scheduling transforms.
- Demonstration of how to automatically generate data parallel code from Data Parallel View components on the CUDA example.

4. THE DATA PARALLEL VIEW
To address data parallelism in heterogeneous architectures, we create a Data Parallel View on Polyhedral Process Networks. The DPV enables a many-thread per node domain mode of PPN processing and can be mapped to data parallel shared-memory targets, such as vector processing units and GPUs. The DPV incorporates a minimal set of extensions to the PPN model required for the generation of well structured data parallel kernels.

Figure 3: A Data Parallel View on the PPN. The node ND2 is processed by a single thread, while the data parallel node DP – ND2 can be processed by multiple threads in parallel.

We approach the DPV construction as follows: First, we construct a DP View for each process in the PPN model and transform ports and channels accordingly. A DP Node encapsulates a PPN node and its self-links, as indicated by the dashed line on the example PPN in Fig 3. A PPN node is specified by a computational function and associated domains: iteration domain and input/output port domains. The number and type of node domains is preserved in the associated DP Node. As an extension to the PPN model, we expose data parallelism available within a node domain by performing exact data dependency analysis, computing a time-space mapping, and finally applying the time-space transformation to obtain data parallel domains. The details of the transformation into a DP node domain are presented in Section 4.1. The DP Nodes are concurrency safe, i.e. a DP Node domain can be processed by multiple threads in parallel. Threads processing a DP Node in parallel scan the DP node domain in a sequence of time steps, as specified by the time-space mapping.

Second, we model the channels (links) as linear arrays, and map them to a shared memory space, e.g. global memory on the GPU. The linear arrays follow the single assignment code (SAC) semantics of the PPN model. One array has only one producer node. Each element of the array uniquely corresponds to an iteration point execution in the producer node. Depending on the node domain size, this may result in large arrays. When mapping to the global memory of the GPU the memory size is not an issue. However, in other cases (e.g. mapping to CUDA shared memory), we develop alternative strategies. In the DP View, the channels are data parallel, i.e. they can provide multiple data elements at the same time step. The self-links of a PPN process are encapsulated in the corresponding DP Node to capture producer-consumer relationships between time steps. Internal operation of DP nodes is discussed in Section 4.2. DP ports can be read and written by multiple threads simultaneously. We explain how to achieve data-parallel memory accesses in Section 4.3.

Third, we consider the network mapping implications. The DPV is applicable on a single PPN node, on a PPN net-
work as a whole, or on a part of the PPN network. In all cases, the boundaries of the network have to be carefully mapped onto the target architecture. On GPUs, the global memory (DRAM) is used as the main point for communication with the host and other components. In this case, we map the boundaries of the DPF network onto the global memory and copy data in/out of the global memory using the CUDA API. The task parallelism characteristic for the process networks is captured by mapping DPNodes to CUDA kernels in different streams. Following this approach, DPNodes with no dataflow dependencies can be executed concurrently provided that the target architecture supports concurrent kernel execution, as in the case of Fermi GPUs. We model dataflow dependencies between processes as dependencies between streams and use CUDA events to enable kernel execution when inputs are available. On the CUDA architecture, communication and synchronization between thread blocks is not supported. In the first order approach presented in this paper, our goal is to explore means of introducing data parallelism and thus we restrict ourselves to coarse-grain pipeline parallelism between nodes in order to enable execution of PPNs on data parallel targets, such as CUDA.

### 4.1 Data Parallelism in The PPN Model

Our approach for discovery of data parallelism in PPN nodes is based on data dependency analysis. If there are no dependencies between two or more iteration points, these iteration points can be executed in parallel. Using the exact data flow analysis, we find the exact dependence relationships between iteration points in a domain. Polyhedral tools, such as Feautrier’s scheduling [10], can be used to compute a schedule for a node domain on the basis of the exact dependence relationships. The schedule assigns each operation to a time step at which it can be executed, while preserving the data dependencies. When data parallelism is available, multiple iteration points are scheduled at the same time step and thus can be executed in parallel.

#### 4.1.1 Domain Transformation

To illustrate extraction of data parallelism, we use a time-optimal scheduling function \( \mathcal{t} \) that maps each operation to the earliest possible time step and an allocation function \( \mathcal{p} \) that maps each operation to a processing element, e.g. a thread. To compute the scheduling and allocation functions, we used the Feautrier algorithms implemented in the LoPo compiler [11]. Other algorithms can be applied without a loss of generality. Together, the schedule \( \mathcal{t} \) and the allocation \( \mathcal{p} \) form a time-space mapping \( \mathcal{T} \). Fig 5 illustrates how the time-space mapping \( \mathcal{T} \) is applied on the node ND2 in the predictor example \( (N = 4) \).

#### 4.1.2 Characterization of a Target Domain: \((W, D)\) Parameters

We have shown how data parallelism in a node domain can be identified. By constructing and applying time-space mappings for each domain in the PPN model, we obtain a set of target polytopes. This is the first step towards establishing a data parallel view on a PPN node, which we call a DP Node. The mapping of iteration points to the time line give us a sequence of steps within a DP Node, and the mapping to the space dimension tells us which operations are executed in parallel.

Once we obtain the target node domain using the selected time-space mapping, we can characterize a DP Node by its maximal width and its minimal depth. The depth \( D \) and the width \( W \) of the target domain are computed as a combination of the lexicographic maximum and minimum of the...
4.2 Data Parallel Process Execution

In the previous section, we showed how to transform a PPN node into a data parallel node that can be processed by multiple threads. In this section, we show how to capture the state between time steps and how to handle different types of data parallel processing within a node.

4.2.1 Transformation of Self-Links

In the PPN model, a node domain is processed by a single thread and the state between process executions is captured by self-links. The self-links represent the intra-loop dependencies and are used to pass the data produced in one iteration point of a node domain to another. For example, the self-links ED1 and ED3 in Fig 2 pass the values of the output argument out0 to the subsequent executions of the process ND2, where its value is consumed. However, on a data parallel architecture, a node domain is processed by a block of threads in a series of time steps. In a DP Node, the self-links are replaced by an internal memory buffer, which holds the intermediate results that may be required in subsequent time-steps. Since the data element produced by one thread may later be consumed by another thread, the DP Node buffer is of shared memory type. Shared memory is used as a communication medium between the threads, thus enabling a cooperative parallel processing of a node domain.

4.2.2 Independent and Cooperative Parallelism

We identify two types of data-parallelism resulting from self-links: cooperative parallelism and independent parallelism. The data parallel node domain in Fig 5(c) is a typical example of cooperative parallelism, since the results produced by one thread are consumed by other threads at later time steps. The dependence vectors show that the data produced by thread $p=1$ at time step $t=1$ is consumed by threads $p=1$ and $p=2$ at $t=2$, and thus has to be communicated from thread $p=1$ to thread $p=2$.

Let us define the types of DP Node parallelism as follows: Cooperative parallelism occurs when there is a memory location that is written by a thread $p_i$ at a time step $t_s$ and read by a different thread $p_j$ at a time step $t_{s'}$, such that $t_{s'} \geq t_s$. Contrary, independent parallelism occurs if there are no threads that require data produced by different threads.

These two concepts are illustrated in Fig 6. The arrows between points (function executions) in the DP Node Domain indicate the flow of data between time steps. By default, we assume that a DP Node is executed with cooperative parallelism. The shared memory buffer introduced in the previous section is used as a communication medium between different producer-consumer threads. After each time step, a synchronization on the DP Node level is required to ensure that no race conditions occur. The synchronization points are indicated by the dashed lines in Fig 6(a).

To determine the exact type of data parallelism within a DP Node, we developed a set of tests. When independent parallelism is detected (Fig 6(b)), some requirements on the mapping can be relaxed. As there is no communication of data values between different threads, the synchronization between time steps is not required. In addition, the shared memory buffer can be replaced by thread private memories, such as registers.

4.3 Data Parallel Memory Accesses

A process in the PPN model uses read/write ports to read/write data elements to/from the channels. The DP View preserves the configuration of the PPN ports, i.e. the type of ports and function arguments associated with the ports stay the same. The input and output port domains (IPDs/OPDs) are transformed to the same target space as the node domain of the process, as described in Section 4.1. However, the time-space transformation of port domains alone is not sufficient to fully determine how data is loaded/stored on a shared-memory architecture. First, the PPN model assumes that ports write data to FIFO buffers, which are typically not available on data parallel architectures. In addition, when using the FIFOs, on shared-memory architectures, we found that the PPN channels map naturally to memory buffers, implemented as linear arrays. The channels that represent the same SAC variable and connect the same nodes are represented by a single memory buffer. The shared memory buffers can be accessed by multiple threads in parallel. The procedure for reading/writing to a channel is modified to accommodate these changes.

Thus, to truly support data parallel execution, a DP Node must be capable of reading/writing multiple data elements to the memory in parallel, as illustrated in Fig 4. The read ports IP2 and IP4 of the Node ND2 in Fig 4(a) read values from the Ncritical variables in the source code, such as $A[i-1][j]$ and $A[i][j-1]$, one data element at a time. The function arguments are read from the channels ED2 and ED4, which represent the SAC array $A_1$ and are implemented as FIFO buffers in the PPN. However, the read ports IP2 and IP4 of the DP Node $DP - ND2$ in Fig 4(b) can be accessed by multiple threads to read multiple data elements. Each thread calculates its own address and uses it to read the argument from the memory array $gA_2$, which is the global memory storage for the SAC array. The channels ED2 and ED4 are implemented as FIFO buffers in the PPN. Similarly, the data parallel port $OP1d2$ can be used to write multiple data elements to $gA_2$ simultaneously. To resolve the data-parallel memory accesses, the PPN’s reads/writes of data elements from/to a FIFO buffer have to be mapped to a linear address space. This means that for each function argument, the associated address space and the exact memory location have to be determined.

To accomplish data-parallel read/writes, we augment the PPN link and port models with the following information:

- **Address Space** - Association of an address space (e.g.,
global memory, shared memory) depending on the type of the associated channel (external vs. self-link).

- **Array Name** - Association with a (SAC) array variable. The function arguments can be read from different array elements in parallel, or written to different elements in parallel.

- **Address Function** - Association of an affine address function for indexing the elements of an array variable. The address functions of the DP Ports are obtained as a composition of mappings from the iteration domain of the node to the data parallel domain in \((t, p)\) coordinates and from the \((t, p)\) domain to the linear address space.

Examples of data parallel extensions to PPN ports including address generation are illustrated for CUDA in Section 5.

### 5. EXECUTION ON THE GPU

The Data Parallel View on the PPNs, presented in the previous section, has been prototyped in the KPN2GPU tool. The KPN2GPU is a Java-based extension of the Compaan Compiler targeting the GPU architecture, however we designed the DPV model in such way that it can be used for generation of vector code in other models too. In this section, we explain how to use the Data Parallel View on the PPNs to derive the components of a CUDA/GPU kernel automatically. By providing infrastructural tools to generate all necessary components of a CUDA kernel, such as variables, function calls, look-up addresses and conditions, we provide the basic building blocks, which can be further analyzed and recombined to generate fully optimized, high-performance GPU code.

The GPU architecture is based on a parallel array of programmable streaming processors, which are exposed by the CUDA API as a grid of thread blocks, each containing multiple (typically 128-512) threads. Threads within a thread block can execute independently or cooperatively. If cooperation is required for the execution of a thread block, the threads communicate via low-latency shared memory. The GPU’s global memory is accessible to all threads executing on the GPU and can be used for intra-kernel communication, however it has much higher memory latency.

As DP Nodes capture different forms of parallelism, including cooperative parallel execution which requires fine-grained synchronization, they map naturally to CUDA thread blocks. For large values of the domain’s structural parameters we first perform top-level partitioning of the PPN nodes. The top-level partitioning allows us to decompose a single PPN node into multiple sub-nodes, each of which is mapped onto a single CUDA thread block in the computational grid. This approach allows us not only to scale up the domain size, but also to more efficiently utilize GPUs streaming multiprocessors. The communication between DP Nodes is achieved through channels, which are mapped to linear arrays in GPU address space. More specifically, we map external links to the global memory of the GPU, following the SAC semantics. Self-links are replaced by shared memory buffers. We generate the addresses of the input/output arguments and loading/storing conditions from the specification of the DP Ports.

The CUDA programming model allows the programmer to define C functions, called kernels, that, when called, are executed \(N\) times in parallel by \(N\) different CUDA threads [13]. To demonstrate the mapping between DPV components and CUDA code, in the listing below we show a skeleton of a base-line CUDA kernel for node \(ND_2\). The ND_2Kernel is executed by \(W\) parallel threads in \(D\) steps.

```c
void ND_2Kernel(const float* gA_1, float* gA_2) {
    extern __shared__ int sA_2[];
    float in0, in1, out0;
    int p0 = threadIdx.x + THREAD_OFFSET;
    int t0 = 0;
    bool active;
    //Port conditions
    bool ND2IP1, ND2IP2, ND2IP3, ND2IP4;
    bool ND2OP1, ND2OP1d1, ND2OP1d2;
    //Port addressing
    int ND2IP2Adr, ND2IP1Adr, ND2IP4Adr,
    ND2IP3Adr;
    int ND2OP1Adr, ND2OP1d1Adr, ND2OP1d2Adr;

    for(t0 = 0; t0 < D; t0++) {
        activeND2 = ... //test if (t0, p0) in target domain
        ...}

    //Computation of port conditions and addresses
    ND2IP1 = (activeND2 & k & (t0−p0 >= 0));

    ...}
```
To execute DPV on a GPU, we map the space dimension $p$ of the DP Nodes to CUDA threads (each thread is $p$ executing the kernel code), and the time dimension $t$ to the number of steps within the loop in the kernel body. The number of threads in a thread block and the number of time steps are derived from the DP Node’s $(W, D)$ parameters. Each thread that executes on the GPU has a unique thread ID that can be obtained within the CUDA kernel through the 3-dim CUDA built-in variables threadIdx and blockIdx. Thus, we map the space dimension $p$ to the local thread identifier threadIdx.x. The function arguments are represented by the automatic variables in0, in1, out0. The actual values of the arguments are obtained through the ports. This requires generation of port conditions and addresses functions from the DP View.

In order to load the value of the function argument in0, which matches the value of the element $A[i−1, j]$ in the source code, for each function execution we must first determine which read port is active, and then derive the appropriate lookup code on the basis of its address function and associated array and memory space. The address of an element in a 2D data block is computed as follows:

$$\text{Adr} = \text{RowSize} \times \text{ElementIdx}.y + \text{ElementIdx}.x,$$

where $\text{RowSize}$ denotes the width of the data block, and $\text{ElementIdx}$ is a multi-dimensional variable which represents the coordinates (indices) of the data element within the given data block. In the PPN model, the output argument is always stored to the location determined by the values of the source node iterators $(i, j)$ for the given iteration point. On the other hand, the array indices of the input arguments must be reconstructed using the expressions in the mapping matrix of the PPN link from which the given port reads the data values. Let’s suppose that the argument in0 is loaded through the port IP2. The mapping matrix of the link ED2 connected to the port IP2 is used to reconstruct the index expressions as follows:

$$\begin{bmatrix}
  i_{det} \\
  j_{det} \\
  1
\end{bmatrix} = \begin{bmatrix}
  1 & 0 & -1 \\
  0 & 1 & 0 \\
  0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
  i_{src} \\
  j_{src} \\
  1
\end{bmatrix}.$$

From the consumer-producer relation above, we find the indices of the data element $\text{ElementIdx}.y = i − 1$ and $\text{ElementIdx}.x = j$. As the next step, the index vector components $(i, j)$ are transformed into the target domain given in $(t, p)$ coordinates using the time-space mapping $T$:

$$T^{-1} = \begin{bmatrix}
  1 & -1 & 2 \\
  0 & 1 & 0 \\
  0 & 0 & 1
\end{bmatrix} \Rightarrow \begin{bmatrix}
  i \\
  j
\end{bmatrix} = \begin{bmatrix}
  t-p+2 \\
  p
\end{bmatrix}.$$
captured within the DPView, we selected three test cases on the basis of their dataflow dependencies: parallel2D, predictor and grid. The main computational kernels of these test cases have dependencies which are characteristic for many imaging, simulation, and scientific applications:

- parallel2D - absolutely parallel
- predictor - pipelined + cooperative parallelism
- grid - pipelined + independent (sync-free) parallelism

The parallel2D is representative for application fields, such as image processing, which have shown significant benefits from acceleration on the GPU architecture. Each operation, e.g. such as computation of a new pixel value, is executed independently. The predictor example, with the dependencies depicted in Fig. 2, is an example of pipelined parallelism. The two-way dependencies between iteration points impose fine-grained synchronization requirements. The grid example features pipelined parallelism. However, the DPV tests indicate that each thread processing the domain can execute independently, thus requiring no synchronization between time steps. As an illustration, in Table 1 we show the characteristics of the main transformer nodes in the PPNs.

Table 1: PPN Transformer Node Characteristics

<table>
<thead>
<tr>
<th>Case Name</th>
<th>Dim</th>
<th>Size</th>
<th>#Deps</th>
<th>Direction Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel2D</td>
<td>(2D)</td>
<td>$M \times N$</td>
<td>0</td>
<td>(0,0)</td>
</tr>
<tr>
<td>Predictor</td>
<td>(2D)</td>
<td>$M \times N$</td>
<td>2</td>
<td>(1,0),(0,1)</td>
</tr>
<tr>
<td>Grid</td>
<td>(2D)</td>
<td>$M \times N$</td>
<td>1</td>
<td>(1,1)</td>
</tr>
</tbody>
</table>

Table 2: Example Domain Schedules using Feautrier

As an example input for the time-space transformation, we derived time optimal scheduling functions using Feautrier’s algorithm [10]. The schedule functions and their matching allocations are given in Table 2. The time-optimal schedule assigns execution of each iteration point of a node domain to the earliest time step according to the dataflow dependencies. Only predictor case has a standard affine schedule. In parallel2D, the schedule is $t = 0$, which identifies the embarrassingly parallel case. In the grid example, the time optimal schedule is piecewise affine. The effect is that in order to achieve maximal data parallelism, different parts of the grid node domain are transformed using different affine scheduling functions.

<table>
<thead>
<tr>
<th>Case Name</th>
<th>Schedule</th>
<th>Allocation</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel2D</td>
<td>$t=0$</td>
<td>$p_0=i$, $p_1=j$</td>
<td>Affine</td>
</tr>
<tr>
<td>Predictor</td>
<td>$t=i+j-2$</td>
<td>$p=j$</td>
<td>Affine</td>
</tr>
<tr>
<td>Grid</td>
<td>$t=i-j$</td>
<td>$t=N-j$</td>
<td>Piece-wise aff.</td>
</tr>
</tbody>
</table>

Table 2: Example Domain Schedules using Feautrier

The statistics of the nodes under DPV are given in Table 3. The form of data parallelism determines the type and the shape of the target domain. In Table 3, column $W$ gives the maximal number of threads that could be used for processing the target domain, column $D$ gives the number of sequential steps executed by each thread, and the last column indicates reduction in the number of sequential time steps assuming large enough number of processing resources. The most general case is the predictor, which shows how a 2D node domain is transformed into a target domain that has one space dimension (i.e. it maps to a 1D thread block) and 1D time, which is reflected in the for loop over time steps in the kernel code. A special case is the target domain of the parallel2D transformer node. It is assigned a trivial schedule $t = 0$, which results in a singular transformation matrix. All iteration points of the source node converge into one time point. We characterize this case as an absolutely parallel target domain (APTD), and handle it by mapping all source domain dimension to space dimensions in target domain. In the grid case, the maximum data parallelism is achieved using a piecewise-affine schedule. In the DPV, we introduce the support for piece-wise affine functions by allowing transformation of each source node domain into one or more complementary target (sub-)domains. This feature is exploited when mapping onto the GPU by generating concurrent kernels or by merging the target sub-domains into a single domain (e.g. triangular shape). Experiments on the grid example indicate a performance improvement of 20-40% depending on the grid size.

Figure 7: Node ND2 GPU Execution time

The GPU execution time for the transformer nodes of the example PPNs is given in Fig 7. Experiment setup is as follows: Fermi architecture supporting concurrent kernel execution, CUDA kernels processing a fixed workload 16MAD, node domain partitioning (block sizes case dependent), input/output data in the GPU DRAM, performance results are given for global memory accesses only. The parallel2D represents an application that perfectly matches the GPU architecture. Pipelined computations, such as predictor and grid, map well onto the GPU only if the size of the target node domain fits a single thread block, since their dependencies impose inter-block communication. One of the main constraints of the CUDA architecture however, is the requirement that all thread blocks execute independently. If it is necessary to preserve dataflow dependencies between thread blocks, multiple kernel invocations are required to process a PPN node. The global synchronization points introduce additional overheads and highly increase the processing time, as illustrated by an order of magnitude larger run time of grid and predictor kernels compared to parallel2D in Fig 7.

In Fig 7, we show performance results when using global memory (GM) for self-links. However, CUDA kernels generated from DPV are highly regular. The information contained in DPLinks can be utilized to better match different memories supported by the target architecture. CUDA provides low latency shared memory (SM), which is accessible
in read/write manner by all threads in a thread block and thread private registers. In the predictor example, we experimented with mapping self-links to CUDA GM and SM in different variants. When SM is used in the SAC manner (sm-sac), the SM size quickly becomes prohibitively large, as indicated by Fig 8(a). The max problem size supported by sm-sac was 64K iteration points. The alternative approach (sm-sp) imposes much lower memory requirements and scales better with the problem size, as shown in Fig 8(b).

Although sm-sp requires introduction of an additional barrier to ensure that all threads have finished loading shared memory data before another round of writing starts, it results in up to 3x over the baseline version. In the grid case, the self links can be completely mapped onto registers as DPV identifies it as independent pipeline parallelism. In Fig 8(c), we show significant performance improvement that is obtained by replacing GM self-links by registers only.

Finally, we compared performance of CUDA applications generated directly from the DPV on an NVIDIA Tesla C2050 card to the C code running on an AMD Phenom(tm) II X4 965 Processor. All PPN nodes are executed on the GPU. In these experiments, we observed speedups of up to 150x for parallel2D, 7x for predictor and 30x for grid. The results are given for all (inter- and intra-threadblock) dependencies satisfied. As all three cases are bandwidth bound, further increase in the workload per iteration point showed that the extra computation improves the speedups. In the previous section, we showed how CUDA kernels can be generated by simply traversing the DPV and mapping its components on the CUDA constructs. The kernels are well structured and exhibit highly regular computation pattern. As future work, we plan to develop optimizations for simplification of guard conditions and memory access patterns.

7. RELATED WORK

Data parallel platforms have received a lot of attention in the last years, mainly due to the success of GPUs as general purpose computing accelerators, however most people program them directly on top of the vendors APIs, such as CUDA. While programming directly in CUDA allows writing fast, highly optimized computation kernels if you are an experienced CUDA programmer, little attention is given to code portability and maintainability. With the emergence of heterogeneous embedded architectures, which not only contain multicore processors and special function units, but also manycore GPUs and vector processing units, the question of a unified programming model becomes even more important. The OpenCL is an emerging standard for cross-platform, parallel programming of modern processors, servers and embedded devices. However, OpenCL programming is still very low level in nature and requires major code rewriting.

Numerous projects intend to make code generation and offloading to accelerators easier, but many of them are centered on a single form of parallelism and/or require extensive use of proprietary pragmas to guide the transformation process. For example, HMPP [14] allows programmers to explicitly specify which parts of the sequential application are to be offloaded on a target device, such as GPU or Cell processor, and in which way. Although this approach accelerates code generation, it still requires good understanding of the target architecture in order to provide appropriate directives. The second class of compiler frameworks transforms C applications to parallel codes. For example, CETUS compiler is capable of generating data parallel OpenMP and subsequently CUDA code automatically from C code [2,15]. However, they use classical compiler data parallelization techniques, whereas we use polyhedral model.

Identification of data parallel operations using scheduling and tiling techniques has been well known in the polyhedral community [16, 17]. For example, a recent extension of PLuTo framework [4] includes automatic transformation of C source code into CUDA code, using an affine tiling function optimized for locality and coarse-grain parallelism extraction. The model presented in this paper is complementary to these approaches, and can make use of different polyhedral transforms for data parallelism. Although we build on the same polyhedral techniques and tools, our contribution is a model for capturing and characterization of these and other transforms for data parallelism. Introducing another layer of abstraction, not only allows us to support more complex functions which may result in multiple target domains, but also to perform characterization, different kinds of analysis and transforms on the model, such as merging target domains and/or distributing load across different nodes, before going into the code generation phase. Furthermore, the PLuTo framework works directly on the polyhedral reduced dependence graph (PRDG) representation, which is an excellent choice for data parallelism. Our starting model PPN is one level above, allowing us to work with data parallelism, pipeline parallelism and task parallelism. The clean separation of communication and computation, makes the mapping onto heterogeneous architectures possible.

The PPN representation features inherent task and pipeline parallelism, and is already extensively used for automatic generation of pipeline and task parallel programs for a given target architecture (e.g. x86, FPGAs, IXP) and programming model (e.g. PThreads, SystemC, etc). The Compaan framework [5] performs exact data flow analysis of an application and automatically generates a polyhedral KPN representation that features all task and pipeline parallelism available in the source code, but it does not consider data parallelism. The plane cutting technique [18] enables extraction of coarse grain data parallelism from PPNs, however it requires close synchronization between all threads and does not support SPMD model. The data parallelism
is extracted by node replication, which fits well multicore CPUs, but is not scalable for massively parallel architectures. On the other hand, our approach addresses fine grain data parallelism and provides a view on the PPN which enables mapping onto vector-based data parallel architectures.

8. CONCLUSION

In this paper, we presented the Data Parallel View on PPNs. The PPNs model is automatically generated from sequential applications and is utilized to generate task and pipeline parallelism. We extend the scope of supported parallelism and enable mapping of PPNs onto architectures that also feature data parallelism, such as future embedded platforms featuring diverse mix of multicores, vector-based units, GPU cores and special function accelerators. Within the Data Parallel View, we introduced abstractions required to expose and make use of data parallelism that can be found within PPN node domains. Furthermore, we showed how to construct the Data Parallel View, how to characterize it, and how to use it to generate data parallel code for GPUs from applications PPNs. The DPV is a general model, and as such maps well onto GPUs, but can be also used for other vector-based targets. The first order results are promising and we are currently investigating how to best utilize the concepts and infrastructure developed for this paper to generate optimized GPU kernels. We believe that the extension of the PPNs with data parallelism has a potential to make it a holistic approach for exploiting data, task and pipeline parallelism in the automated code generation for heterogeneous embedded architectures.

9. REFERENCES