Dataflow Computing with Polymorphic Registers

Cătălin Ciobanu, Georgi Gaydadjiev
Department of Computer Science and Engineering,
Chalmers University of Technology, Sweden
{catalin,georgig}@chalmers.se

Christian Pilato, Donatella Sciuto
Dipartimento di Elettronica, Informazione e Bioingegneria,
Politecnico di Milano, Italy
{pilato,sciuto}@elet.polimi.it

***Abstract*—Heterogeneous systems are becoming increasingly popular for data processing. They improve performance of simple kernels applied to large amounts of data. However, sequential data loads may have negative impact. Data parallel solutions such as Polymorphic Register Files (PRFs) can potentially accelerate applications by facilitating high speed, parallel access to performance-critical data. Furthermore, by PRF customization, specific data path features are exposed to the programmer in a very convenient way. PRFs allow additional control over the registers dimensions, and the number of elements which can be simultaneously accessed by computational units. This paper shows how PRFs can be integrated in dataflow computational platforms. In particular, starting from an annotated source code, we present a compiler-based methodology that automatically generates the customized PRFs and the enhanced computational kernels that efficiently exploit them.**

I. Introduction

Heterogeneous High-Performance Computing (HPC) systems are becoming increasingly popular for data processing. A typical example of a heterogeneous Chip Multi-Processor is the Cell Broadband Engine [1], which combines one Power Processor Element (PPE) and eight Synergistic Processor Elements (SPEs). The PPE runs the operating system and the application's control sections, while the SPEs, designed to excel in data intensive computations, execute the bulk of the applications. Another approach is to combine General Purpose Processors (GPPs) with reconfigurable devices, e.g., Field Programmable Gate Arrays (FPGAs). The GPPs are mainly used for I/O and control dominated functions, while FPGAs accelerate the application's computationally intensive parts. One representative example of such systems is the MaxWorkstation [2], combining Intel x86 processors with multiple Xilinx Virtex-6 FPGA accelerators. The MaxWorkstation adopts the dataflow computation model and organizes the data into highly regular streams flowing through the functions implemented in reconfigurable hardware. This platform allows the designer to focus on high-level application development, while the corresponding HDL (i.e., the traditional high-level synthesis) is generated using a dedicated Java-to-HDL compiler.

Very efficient implementations [3], [4] have been obtained while accelerating streaming applications using Maxeler dataflow engines. In almost all cases, these data processing applications consist of relatively simple kernels applied to large amounts of data. For example, image processing algorithms usually apply digital filters at some stages which process the stream in blocks through "masks" (i.e., regular patterns for accessing the data). Sequential loads of these rectangular blocks from a local memory may negatively impact performance. Therefore, parallel memory access schemes are of interest.

The Polymorphic Register File (PRF) [5] is a novel architectural solution targeting high performance execution. Using the Single Instruction, Multiple Data (SIMD) paradigm, the PRF provides simultaneous paths to multiple data elements. The PRF implements conflict-free access to the most widely used memory access patterns in the scientific and multimedia domains. Efficient design of PRF-based systems demands careful identification of the most suited data structures and the corresponding access patterns in addition to adjusting the algorithms in order to take advantage of the PRF memory. However, performing the above steps manually is tedious and error-prone. We strongly believe that semi-automated methodologies crafted to support the designer are preferable.

This paper aims at integrating PRFs into a state-of-the-art heterogeneous HPC system for accelerating dataflow applications. Our methodology augments the existing tools for defining streaming architectures. Starting from the initial C code, the designer annotates the variables that will be placed in PRF and the desired memory access patterns. To integrate the PRFs, our methodology supports the Java code creation required to describe the selected kernels. We rely on the vendor-specific tools for bitstream generation and FPGA device management. While our methodology currently targets the MaxWorkstation, it can be adapted for other FPGA-based systems as well, provided the necessary PRF interface is implemented. More specifically, the contributions of this paper are the following:

- PRF instantiation on the MaxWorkstation, providing parallel access to specific variables and accelerating access to preselected data blocks;
- Compiler-based methodology supporting automatic creation of the corresponding system organization;
- Methodology for enhancing the application with the parallel memory access management (e.g., substituting the sequential loads with parallel memory accesses).

The paper continues as follows: background information is provided in Section II, and the related work is presented in Section III. Section IV describes our target system including the PRF, while Section V details our compiler-based methodology. Section VI presents a case study of the Separable 2D Convolution algorithm. Finally, Section VII concludes the paper and outlines future work directions.

II. Background

When designing computing systems, architects anticipate the requirements of the target applications. However, as new workloads continuously emerge, it is close to impossible to

forecast the requirements of arbitrary workloads and design a single best configuration. The growing diversity of computational tasks drives the need for increasingly flexible computer systems. While software is able to abstract certain low-level (micro)architectural details, hardware support is preferable in domains such as HPC or embedded systems to avoid excessive performance and energy overheads. System adaptability can be introduced at different levels. Reconfigurable hardware allows circuit specialization according to the current workload. On the other hand, architectural level adaptability lets programmers focus on the algorithms instead of worrying about complex data transfers or other low-level details. In this work, we combine both approaches targeting an FPGA-based HPC system.

We target the Maxeler MaxWorkstation [2], which combines an Intel CPU with one or two Data Flow Engines (DFE). The DFEs, based on state-of-the-art Xilinx Virtex-6 SX475T FPGAs, are connected to the GPP via PCI Express. Furthermore, up to 48GB of DDR3 DRAM is available on each DFE board. The implementation of a Maxeler DFE consists of *Kernels* and a *Manager*, both written in Java. *Kernels* describe the data paths which implement the target algorithm. The *Manager* describes the data flow between Kernels and off-chip stream I/O (e.g., GPP, DRAM). The application running on the GPP is written either in C or in FORTRAN and uses a set of API calls to communicate with the *Kernels* and the *Manager*. The Maxeler Run-Time (MaxCompilerRT) and MaxelerOS software libraries enable the communication between the GPU and the DFE, and abstract the low-level details of DMA transfers over PCI Express. MaxCompiler provides the necessary compilers and libraries needed to create a DFE design and to allow the GPP to communicate with it. For the bitstream generation, MaxCompiler uses the silicon vendor tools (e.g., Xilinx ISE). Moreover, MaxCompiler allows instantiation of hand crafted HDL code, which connects to other Kernels via streams. We take advantage of this to instantiate our PRF hardware, described in detail in [6] and [7].

The PRF was developed as part of the Scalable computer ARChitecture (SARC) project as its Scientific Vector Accelerator [8]. A PRF is a parameterizable register file, logically reorganized under software control to support multiple register dimensions and sizes simultaneously [9]. The total store capacity is fixed, containing $N \times M$ data elements. Figure 1 provides an example of a two-dimensional (2D) PRF with a physical register file size of 128×128 64-bit elements. In this simple example, the available storage has been divided in six logical registers with different locations and dimensions, defined using the Register File Organization (RFORG) Special Purpose Registers (SPRs), shown on the right side of Figure 1. For each logical register we need to specify its position (the base), the shape (rectangle, main or secondary diagonal), its dimensions (horizontal and vertical length) and the data type (integer / floating point, 8/16/32/64 bits). The 2D PRF benefits:

- **Potential performance gains** - by reducing the number of committed instructions;
- **Improved storage efficiency** - the registers are defined to contain exactly as many elements as required, completely eliminating the potential storage waste inherent to organizations with fixed register sizes and maximizing the available space for subsequent use;

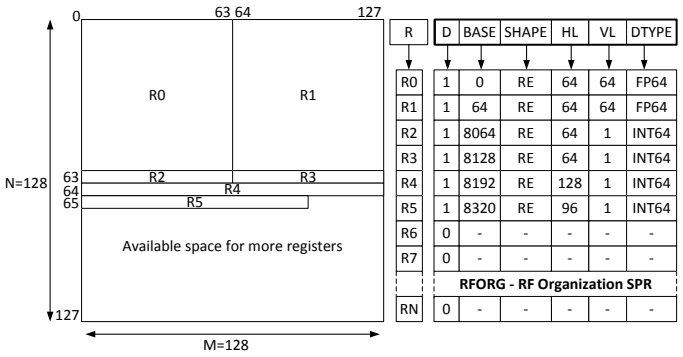

Fig. 1. The Polymorphic Register File

- **Customizable number of registers** - the number of registers is no longer an architectural limitation, contributing to the PRF runtime adaptability. Unlike fixed number of registers of predefined size in traditional systems, the unallocated space can be further partitioned into an arbitrary number of registers of arbitrary (1D or 2D) shapes and dimensions;
- **Reduced static code footprint** - the target algorithm may be expressed with fewer, higher level instructions.

One of the main objectives of the PRF is scalability in terms of performance and storage capacity. The key to high-performance PRFs lies on their capability to deliver aligned data elements to the computational units at high rates. Moreover, a properly designed PRF allows multiple vector lanes to operate in parallel with efficient utilization of the available bandwidth, which implies parallel access to multiple data elements. The most performance-efficient solution is to support the specific access patterns in hardware, using parallel memory access schemes.

Previous works show that PRFs can reduce the number of committed instructions by up to three orders of magnitude [9]. Compared to the Cell processor, PRFs decrease the number of instructions for a customized, high performance dense matrix multiplication by up to 35 times [8] and improve performance for Floyd and sparse matrix vector multiplication [9]. A CG case study [10] evaluated the scalability of up to 256 PRF based accelerators in a heterogenous multi-core architecture, with two orders of magnitude performance improvements. Furthermore, potential power and area savings were shown by employing fewer PRF cores compared to a Cell processors system. A preliminary evaluation of the PRF 2D separable convolution performance [11] showed that PRFs outperform state-of-the art GPUs in throughput for mask sizes of 9×9 elements of larger when bandwidth is constrained.

For the PRF hardware implementation, previous works considered a 2D array of $p \times q$ (we use "$\times$" to refer to a 2D matrix, and "$\cdot$" to denote multiplication) linearly addressable memory banks and use parallel access schemes to distribute the data elements in the corresponding memory bank. The memory schemes proposed in [5] are denoted as **ReRo**, **ReCo**, **RoCo** and **ReTr**, each of them supporting at least two conflict-free access patterns: 1) Rectangle Row (**ReRo**): $p \times q$ rectangle, $p \cdot q$ row, $p \cdot q$ main diagonals if p and $q+1$ and co-prime, $p \cdot q$ secondary diagonals if p and $q-1$ are co-prime; 2) Rectangle

Column (**ReCo**): $p \times q$ rectangle, $p \cdot q$ column, $p \cdot q$ main diagonals if $p+1$ and q are co-prime, $p \cdot q$ secondary diagonals if $p-1$ and q are co-prime; 3) Row Column (**RoCo**): $p \cdot q$ row, $p \cdot q$ column, aligned ($i\%p = 0$ or $j\%q = 0$) $p \times q$ rectangle; 4) Rectangle Transposed Rectangle (**ReTr**):$p \times q$, $q \times p$ rectangles (transposition) if $p\%q = 0$ or $q\%p = 0$. Conflict-free access is therefore supported for all of the most common vector operations for scientific and multimedia applications [5].

Synthesis results for FPGA and ASIC technologies have been presented in [7], [6] and [5]. Results targeting the Virtex-7 XC7VX1140T-2 FPGA show feasible clock frequencies between 111 MHz and 326 MHz and reasonable logic resources usage (less than 10% of the available LUTs). Using 90nm ASIC technology, the PRF clock frequency is between 500MHz and 970MHz for storage sizes of up to 512KB and up to 64 vector lanes. Power is also within reasonable limits, not exceeding 8.7W for dynamic and 276mW for leakage [6].

III. Related Work

Efficient processing of multidimensional matrices has been targeted by other architectures, as well. One approach is using a memory-to-memory architecture, such as the Burroughs Scientific Processor (BSP) [12]. The BSP machine was optimized for the Fortran programming language, having the ISA composed of 64 very high level vector instructions, called *vector forms*. A single *vector form* is capable of expressing operations performed on scalar, 1D or 2D arrays of arbitrary lengths. In order to store intermediate results, each BSP arithmetic unit includes a set of 10 registers which are not directly accessible by the programmer. The PRF also creates the premises for a high level instruction set. However, while BSP has a limited number of automatically managed registers which can be used for storing intermediate results, our approach is able to reuse data directly within the PRF. This offers additional control and flexibility to the programmer, the compiler and the runtime system and potentially improves performance.

The Complex Streamed Instructions (CSI) approach does not use data registers at all [13]. CSI is a memory-to-memory architecture which allows the processing of two-dimensional data streams of arbitrary lengths. One of the main motivations behind CSI is to avoid having the Section Size as an architectural constraint. Through a mechanism called auto-sectioning, PRFs allow designers to arbitrarily chose the best section size for each workload by resizing the vector registers, greatly reducing the disadvantages of a fixed section size as in CSI. To exploit data locality, CSI has to rely on data caches. As also noted for the BSP, our approach can make use of the register file instead, avoiding high speed data caches.

The concept of Vector Register Windows (VRW) [14] consists of grouping consecutive vector registers to form *register windows*, which are effectively 2D vector registers. The programmer can arbitrarily choose the number of consecutive registers which form a window, defining one dimension of the 2D register. However, contrary to our proposal, the second dimension is fixed to a pre-defined section size. Furthermore, all the *register windows* must contain the same number of vector registers, and the total number of windows cannot exceed the number of vector registers. The latter severely limits the granularity to which the register file can be partitioned. Such restrictions are not present in our PRF Architecture, providing a much higher degree of freedom for partitioning the register file. Therefore, our vector instructions can operate on matrices of arbitrary dimensions, reducing the overhead for resizing the *register windows*.

Two-dimensional register files have been used in several other architectures, such as the Matrix Oriented Multimedia (MOM). MOM is a matrix oriented ISA targeted at multimedia applications [15]. It also uses a 2D register file in order to exploit the available data-level parallelism. The architecture supports 16 vector registers, each containing sixteen 64-bit wide, elements. By using sub-word level parallelism, each MOM register can store a matrix containing at most 16×8 elements. The PRF also allows sub-word level parallelism, but doesn't restrict the number or the size of the two-dimensional registers, bearing additional flexibility.

Another architecture which also uses a two-dimensional vector register file is the Modified MMX (MMMX) [16]. It supports eight 96-bit wide, multimedia registers and special load and store instructions which provide single-column access to the subwords of the registers. Our PRF does not limit the matrix operations to only loads and stores and allows the definition of multi-column matrices of arbitrary sizes.

Based on Altivec, the Indirect VMX (iVMX) architecture [17] employs a large register file consisting of 1024 registers of 128 bits. Four indirection tables, each with 32 entries, are used to access the iVMX register file. The register number in the iVMX instructions, with a range from 0 to 31, is used as an index in the corresponding indirection table. The PRF also uses indirection to access a large register file, but does not divide the available RF storage in a fixed number of equally sized registers, therefore allowing a higher degree of control when dynamically partitioning the register file.

The Register Pointer Architecture (RPA) [18] focuses on providing additional storage to a scalar processor thus reducing the overhead associated with the updating of the index registers while minimizing the changes to the base instruction set. The architecture extends the baseline design with two extra register files: the Dereferencible Register File (DRF) and the Register Pointers (RP). In essence, the DRF increases the number of available registers to the processor, and The RPs provide indirect access to the DRF. RPA is similar to our proposal as it also facilitates the indirect accessing to a dedicated register file by using dedicated indirection registers. However, the parameters stored in the indirection registers are completely different. Using RPA, each indirection register maps to a scalar element. In PRFs, one indirection register maps to a sub-matrix in the 2D register file, being more suitable for multidimensional (matrix) vector processing by better expressing the available data-level parallelism.

Several alternatives to the Maxeler FPGA-based HPC systems have been introduced in recent years. The Cray XD1 [19] incorporates 12 AMD Opteron processors and six Xilix Virtex-II PRO XC2VP30-6 or XC2VP50-7 FPGAs with dedicated QDR RAM and 3.2 GB/s interconnect. The Cray XR1 blade [20] is a dual Socket 940 design, incorporating one AMD Opteron processor while the second socket holds a Xilinx Virtex-4 LX200 FPGA, communicating with the rest of the system using the high-speed HyperTransport interface.

The SGI Altix 4700 [21] platforms featured a modular blade design, and incorporated the Non-Uniform Memory Architecture (NUMA) shared-memory NUMAflex architecture. Dedicated compute, memory, I/O and special purpose blades were available for the Altix machine. The compute blade contained Intel Itanium processors. The Reconfigurable Application Specific Computing (RASC) [22] blades facilitated two Virtex-4 LX200 FPGAs and dedicated memory DIMMs.

The Convey HC-1 [23] consists of two stacked 1 Unit (U) chassis: one contains the GPP, while the other one contains the FPGAs. The CPU chasis consists of a dual-socket Intel motherboard, out of which one is populated with an Intel Xeon CPU. The other socket is used to route the Front Side Bus (FSB) to the FPGAs. The HC1 contains four Virtex-5 LX 330 Application Engines (AEs), connected to 8 memory controllers through a full crossbar. The memory controllers are connected to proprietary scatter-gather DIMMs. HC-1's memory system is designed to maximize the likelihood of conflict-free accesses. The GPP and the coprocessor memories are cache coherent, sharing a common virtual address space. The HC-1 also contains two additional Virtex-5 LX110 FPGAs which form the Application Engine HUB (AEH), one which interfaces with the FSB and manages the memory coherence protocol. The second AEH FPGA contains a scalar soft-core processor implementing a custom Convey ISA. The softcore acts as a coprocessor to the Intel CPU, and the AE FPGAs are coprocessors to the soft-core.

Note that, assuming that the high-bandwidth PRF interface is implemented, similar benefits as the ones obtained with Maxeler machines can also be obtained with these alternative FPGA-based HPC systems.

Given the time-consuming and error-prone process of designing the HDL code for FPGA-based heterogeneous systems, several C-to-HDL compilers exist. Most notably, the Riverside Optimizing Compiler for Reconfigurable Circuits (ROCCC) 2.0 [24] supports a subset of C and produces VHDL hardware accelerators. The resulting circuits are generic, interfacing to streams or memories. The integration of the ROCCC 2.0 generated code into the target platforms has to be done manually by the designers, requiring VHDL and C glue code in order to handle buffering and timing issues. ROCCC 2.0 is built using the SUIF and LLVM infrastructures and, for the C code, no annotations are required. Our approach is also built using LLVM, but requires additional pragmas in the C code. However, the programmer is not expected to add any additional C or HDL glue code in order to obtain a functional system. Our methodology is built on top of the MaxCompiler toolchain, translating C code into Java hence the manual intervention of the designer is minimized.

Finally, for NVIDIA Graphics Processing Unit (GPU)-based [25] heterogeneous computing, the CUDA [26] programming model has been widely used. CUDA extends the C programming language with additional keywords allowing, among others, explicit allocations of variables in the on-chip and off-chip GPU memory, data transfers between the GPP and the GPU. However, unlike with our approach, programmers still need to parallelize the algorithms in terms of threads and thread blocks, partition memory and schedule synchronizations.

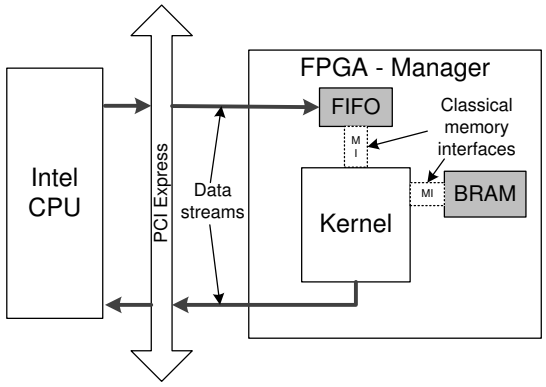

Fig. 2. Original system organization for streaming computation without PRFs

IV. Target System Organization

This Section describes how a classic dataflow computing system can be enhanced to integrate Polymorphic Register Files. To better clarify the proposed system organization, let us assume that we want to enhance a computational kernel that operates on a data stream with a mask of coefficients. The mask can be eventually updated during the computation. The original implementation of this system is shown in Figure 2. In this system, a FIFO, holds multiple values at the same time, based on the kernel requirements. When a new sample arrives, the values are shifted accordingly. Additional local memories store static data (such as coefficients), usually represented as arrays of values.

In both cases, a memory-based approach is usually provided to access these memories. The kernel specifies which element is being accessed. In the case of streaming data, elements are identified by offsets with respect to the current value. In the case of static data, elements are identified by their position in memory. PRFs can store these values types:

- **streaming data** flowing through the different modules, i.e., the processed image;
- **static data**: for example, the set of coefficients used within the computational kernel.

In the former case, PRFs substitute the memory elements usually inserted to store the stream values. The PRFs need to be customized in order to create local registers of proper sizes, i.e., the maximum number of values to be simultaneously stored. Thereafter, by analyzing the kernel behavior, it is possible to determine the memory access pattern. By collecting information about the input stream accesses and the corresponding values, it is possible to determine suitable parallel memory accessing schemes and customize the PRFs accordingly. It is then possible to implement the logic FIFO. Given a new data item, it is necessary to determine where it has to be stored depending on the current PRF configuration.

For storing static data, PRFs can also be used, but without any shifting operations. The PRF registers are initialized with these values. As also discussed for the streaming data, it is possible to identify the kernel memory access patterns.

Figure 3 illustrates the resulting system after PRF integration for streaming data. Compared to the reference implemen-

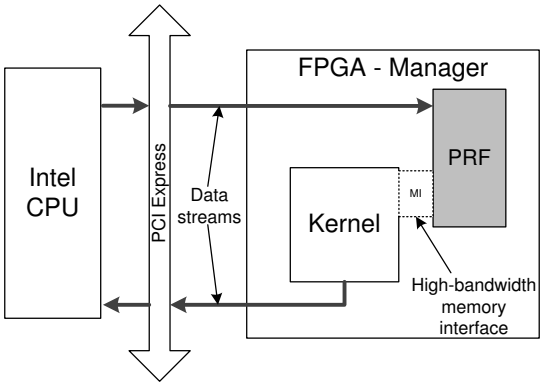

Fig. 3. System organization for streaming computation enhanced with PRFs

tation depicted in Figure 2, wider buses are available in the memory interface, providing multiple values in parallel.

V. Compiler-based Methodology

Figure 4 summarizes the overall methodology for the automatic creation of PRF-augmented systems, previously described in Section IV. In Figure 4, the gray boxes highlight the steps presented in this work. The methodology starts from annotated C code containing custom pragmas for variables stored in Polymorphic Registers Files. The proposed methodology is composed of the following phases: *variable extraction*, *PRF customization* and *code generation*.

As described above, we currently only support the Maxeler MaxWorkstation [2]. However, our methodology can be easily adapted to generate kernels for different architectures by implementing the corresponding code generation. For this reason, the methodology currently outputs the updated Java description of the computational kernel which needs to be compiled using the Maxeler toolchain. Next, the customized PRF is generated, ready to be processed by the synthesis toolchain (MaxCompiler and Xilinx synthesis tools) that generates the FPGA bitstream. Note that the Java-based *Manager* describing how the modules are interconnected also needs to be created. This step is currently done by hand, but can be automated.

In the **variable extraction**, the initial source code is analyzed with a C-to-C compiler step that extracts the variables previously annotated with custom pragmas. An XML file is produced, containing the list of variables and a description of the required memory access patterns required for accessing required by the initial source code. More details about this phase can be found in Section V-A.

The **PRF customization** starts from the XML file generated in the previous phase. The identified variables are allocated to PRFs introduced in the system. The PRFs are then configured for providing values with the required access patterns. The corresponding HDL code is then generated, along with the custom wrapper needed to provide data to the computational kernel. Additional details about this stage are provided in Section V-B.

The **code generation** starts from the initial kernel source code and the information concerning the variables and access

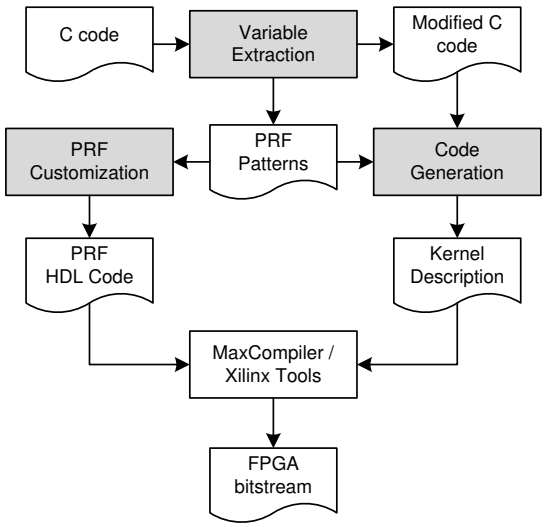

Fig. 4. Overall methodology for supporting automatic PRF integration

patterns previously identified. An additional compiler step automatically translates the behavioral specification into a Java-like kernel description. MaxCompiler is then used to generate the corresponding HDL. Besides the behavioral specification generation (i.e., arithmetic and logical operations), multiple sequential memory accesses are substituted by parallel PRF accesses followed by value unpacking operations. More details about this phase can be found in Section V-C.

A. Variable Extraction

This section describes the C-to-C automatic PRF variable extraction. We implement this analysis of custom pragmas step using the Mercurium compiler [27]. From the example presented above, we identify two annotation types:

- `#pragma prf variable`: specifies the PRF variable name (i.e., identifier of the block of data) and the required PRF space allocation;
- `#pragma prf access`: specifies a PRF block memory access, along with information about the specific accessed element.

The `variable` pragma is defined as follows:

```
#pragma prf variable <name> <size> <type>
```

The first parameter is the variable name. We assume that this variable can represent either stream or static data (`<type>=0` or `<type>=1`, respectively). The second parameter is the space that has to be reserved in the PRF for this variable. In case of input parameters, this value represents the number of consecutive elements of the input data stream that have to be simultaneously stored. If the variable represents static data, this value can be the size of the array itself, thus entirely stored into the PRF. In this case, if the array has been initialized with a set of predefined values (e.g., filter coefficients), these values are reported in the output XML file in order to properly initialize the PRF registers.

The `access` pragma is instead defined as follows:

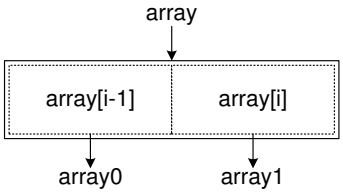

Fig. 5. Extraction of values from the parallel one provided by the PRF.

```
#pragma prf access <var> <index> <name>
```

The first parameter represents the PRF variable name, with the index specified by the second parameter. For streaming variables, the index represents the offset with respect to the current value and thus it can be either positive or negative. Otherwise, for static data, it simply represents the position within the block itself. The last parameter specifies the name used to identify this access. For example, the following code:

```
#pragma prf access array -1 array0
#pragma prf access array 0 array1
int var = array[i-1] + array[i];
```

is translated as follows:

```
int array0 = array[i-1];
int array1 = array[i];
int var = array0 + array1;
```

This transformation aims at simplifying the computational kernel code generation as described in the following section. Finally, all the information presented above is generated as an XML file for use in the subsequent phases.

B. PRF Customization

In this phase, the variables and the correponding access patterns are used to customize the PRF. Furthermore, the corresponding RTL description is generated and integrated into the Maxeler Manager representing the system. Each block stored in the PRF is represented as a logical register, whose base address is used to access it from the kernel. It is also required to determine the shape and the correponding dimensions (horizontal and vertical length) based on the number of stored values and their data types. Moreover, the registers may have to be initialized. Currently this procedure is not fully automated, but it is manually performed by the designer.

C. Code Generation

At this stage, the PRF has been generated. Considering the simple example described above, the variable `array` has been moved to the PRF and only two elements have been stored, with offsets `0` and `-1` with respect to the current position of the stream, as shown in Figure 5. As a consequence, the PRF provides a 64-bit value to the kernel that is unpacked to get the actual values for the computation. For this reason, after the PRF load, the code can be restructured as follows:

```
//array = read_prf(...);
int array0 = trunk_32(array >> 32);
int array1 = trunk_32(array);
int var = array0 + array1;
```

where `read_prf` is a support function that reads 64 bits in parallel from the PRF, while the function `trunk_32` generates a 32-bit value starting from the parameter value.

This transformation has been implemented as a dynamic step in the LLVM compiler [28] (version 3.2). The Java code of the kernel is generated as it is required by the Maxeler MaxCompiler [2]. Even such a simple example clarifies the potential advantages of Polymorphic Register Files. Two sequential memory accesses are substituted by a customized parallel PRF access. The time required for performing the unpacking operations and extract the values from the high-bandwidth memory interface is assumed to be negligible.

VI. Case Study and Analytical Validation

In this paper, we use the Separable 2D Convolution as our Case Study. Convolution is used in digital signal processing (e.g., image and video) for filtering signal values. For example, the Sobel operator is popular in edge detection algorithms. In addition, the Sobel operator is a separable function, allowing the use of two consecutive 1D convolutions to produce the same result as single, more computationally expensive, 2D convolution. The first 1D convolution filters the data in the horizontal direction, followed by a vertical 1D convolution. We will exploit this important property later in this Section.

In digital signal processing, each output of the convolution is computed as a weighted sum of its neighbouring data items. The coefficients of the products are defined by a mask (also known as the convolution kernel), which is used for all elements of the input array. Intuitively, convolution can be viewed as a blending operation between the input signal and the mask (also referred to as aperture from some applications prospective). Because there are no data dependencies, all output elements can be computed in parallel, making this algorithm very suitable for efficient parallel implementations.

The dimensions of a convolution mask are usually odd, making it possible to position the output element in the middle of the mask. For example, consider a 10 element 1D input $I = [\,20\ 22\ 24\ 26\ 28\ 30\ 32\ 34\ 36\ 38\,]$ and a 3 element mask $M = [\,2\ \mathbf{5}\ 11\,]$. The 1D convolution output corresponding to the 3rd input (the one with the value 24) is $2{\cdot}22{+}\mathbf{5}\cdot\mathbf{24}{+}11{\cdot}26 = 450$. Similarly, the output corresponding to the 4th input (26) is obtained by shifting the mask by one position to the right: $2\cdot 24 + \mathbf{5}\cdot\mathbf{26} + 11\cdot 28 = 486$.

When the convolution algorithm is used for the elements close to the edges of the input, the mask should be applied to elements outside the input array (to the left of the first element, and to the right of the last element of the input vector). Obviously, some assumptions have to be made for these "missing elements". In this paper, we will refer to those as "halo" elements. In practice, a convention is made for a default value of the halo elements. The halo elements can be either zeros, or replications of the boundary elements values or any other values, determined by the particular application algorithm. If we consider all halo elements to be 0, the output corresponding to the 10th input (38) is $2\cdot 36 + \mathbf{5}\cdot\mathbf{38} + \mathit{11}\cdot\mathit{0} = 262$.

In the case of 2D convolution, both the input data as well as the mask are 2D matrices. For example, let us consider the case of the 9×9 input matrix:

$$I = \begin{bmatrix} 3 & 5 & 7 & 9 & 11 & 13 & 15 & 17 & 19 \\ 13 & 15 & 17 & 19 & 21 & 23 & 25 & 27 & 29 \\ 23 & 25 & 27 & 29 & 31 & \mathbf{33} & \mathbf{35} & \mathbf{37} & 39 \\ 33 & 35 & 37 & 39 & 41 & \mathbf{43} & \mathbf{45} & \mathbf{47} & 49 \\ 43 & 45 & 47 & 49 & 51 & \mathbf{53} & \mathbf{55} & \mathbf{57} & 59 \\ 53 & 55 & 57 & 59 & 61 & 63 & 65 & 67 & 69 \\ 63 & 65 & 67 & 69 & 71 & 73 & 75 & 77 & 79 \\ 73 & 75 & 77 & 79 & 81 & 83 & 85 & 87 & 89 \\ 83 & 85 & 87 & 89 & 91 & 93 & 95 & 97 & 99 \end{bmatrix}$$

and the 3×3 mask $M = \begin{bmatrix} 4 & 6 & 8 \\ 9 & \mathbf{11} & 13 \\ 14 & 16 & 18 \end{bmatrix}$. Furthermore, the halo elements are assumed to be 0 in this example. In order to compute the 2D convolution output on position $(4, 7)$ we first compute the point-wise multiplication of the 3×3 sub-matrix of the input $\begin{bmatrix} 33 & 35 & 37 \\ 43 & 45 & 47 \\ 53 & 55 & 57 \end{bmatrix}$ with the mask, obtaining $\begin{bmatrix} 132 & 210 & 296 \\ 387 & 495 & 611 \\ 742 & 880 & 1026 \end{bmatrix}$. By summing up all the elements of this matrix, the value 4779 is obtained. Since we assume the halo elements to be 0, they do not contribute to the result and can therefore be omitted from the computation. Therefore, the result on position $(1, 9)$ is computed by the point-wise multiplication of the corresponding sub-matrices from the input $\begin{bmatrix} 17 & 19 \\ 27 & 29 \end{bmatrix}$ and the mask $\begin{bmatrix} 9 & 11 \\ 14 & 16 \end{bmatrix}$, obtaining $\begin{bmatrix} 153 & 209 \\ 378 & 464 \end{bmatrix}$ which accumulates to 1204. The complete 2D convolution result becomes:

$$O = \begin{bmatrix} 576 & 901 & 1063 & 1225 & 1387 & 1549 & 1711 & 1873 & 1204 \\ 1214 & 1809 & 2007 & 2205 & 2403 & 2601 & 2799 & 2997 & 1886 \\ 1934 & 2799 & 2997 & 3195 & 3393 & 3591 & 3789 & 3987 & 2486 \\ 2654 & 3789 & 3987 & 4185 & 4383 & 4581 & \mathbf{4779} & 4977 & 3086 \\ 3374 & 4779 & 4977 & 5175 & 5373 & 5571 & 5769 & 5967 & 3686 \\ 4094 & 5769 & 5967 & 6165 & 6363 & 6561 & 6759 & 6957 & 4286 \\ 4814 & 6759 & 6957 & 7155 & 7353 & 7551 & 7749 & 7947 & 4886 \\ 5534 & 7749 & 7947 & 8145 & 8343 & 8541 & 8739 & 8937 & 5486 \\ 3056 & 4171 & 4273 & 4375 & 4477 & 4579 & 4681 & 4783 & 2844 \end{bmatrix}$$

Assuming the 2D mask has **MASK_V** rows and **MASK_H** columns, the number of multiplications required to compute a single element is **MASK_V** $\cdot$ **MASK_H**.

Separable 2D convolutions (e.g., the Sobel operator) can be computed as two 1D convolutions on the same data. For example, in [29], the 2D convolution $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ is equivalent to first applying $\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$ and then $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$.

Separable 2D convolutions are widely used because fewer arithmetic operations are required compared to the regular 2D convolution. In our example, only **MASK_V** + **MASK_H** multiplications are needed for each output element. Moreover, separable 2D convolutions are more suitable for blocked SIMD execution because the individual 1D convolutions have fewer data dependencies between blocks. In this paper, we focus on accelerating separable 2D convolutions.

Separable 2D convolutions consist of two data dependent steps: a row-wise 1D convolution on the input matrix followed by a column-wise 1D convolution. The column-wise access involves strided memory accesses, which may degrade performance due to bank conflicts. In order to avoid these strided memory accesses, we propose to transpose the 1D convolutions output while processing the data. This can be performed conflict-free by using our **RoCo** memory scheme [7].

A vectorized separable 2D convolution algorithm which avoids strided memory accesses when accessing column-wise input data is introduced in [11]. The input matrix contains **MAX_V** $\times$ **MAX_H** elements. The two masks used for row-wise and column-wise convolutions have **MASK_H** and **MASK_V** elements respectively. We will refer to both as **MASK_H**, since both convolution steps are handled identically by the PRF. The PRF algorithm processes the input in blocks of **VSIZE** $\times$ **HSIZE** elements, vectorizing the computation along both the horizontal and the vertical axes. For clarity, we only present the steps required to perform one convolution step. The same code should be executed twice, once for the row-wise convolution and the second time for the column-wise version. The data will be processed **VSIZE** rows at a time. Without loss of generality, we assume that **MAX_V**$\%$**VSIZE** $= 0$ and **MASK_H** $= 2 \cdot$ **R** $+ 1$.

TABLE I. CONVOLUTION ESTIMATED EXECUTION TIME (CYCLES)

Stage \ Lanes	1	2	4	8	16	32	64	128	256
Load	1124	612	356	228	164	132	116	108	104
Convolution	9228	4620	2316	1164	588	300	156	84	48
Move	128	64	32	16	8	4	2	1	1
Store	1024	512	256	128	64	32	16	8	4
Total	11504	5808	2960	1536	824	468	290	201	157

TABLE II. CONVOLUTION ESTIMATED SPEEDUP

Speedup \ Lanes	1	2	4	8	16	32	64	128	256
Absolute	1	1.98	3.89	7.49	13.96	24.58	39.67	57.23	73.27
Relative	1	1.98	1.96	1.93	1.86	1.76	1.61	1.44	1.28

Because special care needs to be taken at the borders of the input, the vectorized algorithm has three distinct parts: the first (left-most) block, the main (middle) sections, and the last (right-most) one. The first iteration of the convolution takes into consideration the **R** halo elements to the left of the first input elements. Similarly, the last iteration handles the **R** halo elements on the right. The only modification required by the first and last iterations is resizing the PRF logical registers. However, the operations performed remain the same. We assume the dimensions of the input data are much larger than the processed block, allowing us to focus on the main convolution iterations. Moreover, in this scenario, the time required to define the PRF logical registers becomes insignificant compared to the memory and arithmetic operations. Therefore, in the rest of this section we only consider the memory and arithmetic operations of the main convolution iterations.

The row-wise convolution requires the following steps:

1) Row-wise Load **VSIZE**$\times$**HSIZE** input elements;
2) Row-wise convolution using **VSIZE**$\times$**HSIZE** $\times$**MASK_H** multiply-accumulate operations;
3) Column-wise Store the **VSIZE**$\times$**HSIZE** results;
4) Left Move the **VSIZE** $\times$ **R** input elements used as halo elements during the next iteration;
5) In case unprocessed data remains, go to step 1).

The PRF read and write ports can provide multiple data elements simultaneously to **L** computational lanes. Assuming sufficient memory bandwidth and functional units, each convolution step can execute up to **L** times faster.

Since this work is still in progress, we rely on estimations for determining the speed-up potentially introduced by the PRFs in the Maxeler architecture. For example, assuming the following row-wise convolution scenario: $32 \times 32 \times 64$-bit elements block size (**HSIZE** = **VSIZE** = 32), 9 elements (**R** = 4) mask size, average external memory latency of 100 clock cycles for the loads, and 12 clock cycles multiply and

accumulate latency. Table I shows the estimated convolution execution time expressed in clock cycles, and Table II illustrates the associated speedups. In the baseline case, the PRF can only provide one data element per port at each clock cycle. This corresponds to employing a simple serial memory for storing the input and output convolution data. The first four rows of Table I contain the estimated duration of the Load, Convolution, Move, and Store phases. For the baseline scenario (1 Lane), 1024 data elements need to be loaded, which takes 1124 clock cycles. The convolution requires $32 \cdot 32 \cdot 9 = 9216$ multiply-accumulate operations which consume 9228 cycles. In a similar way, we estimate the duration of the halo moving stage as 128 cycles, as $4 \cdot 32$ elements need to be moved. In this case, Storing 1024 data elements is forecasted at 1024 cycles. The combined duration of the four convolution steps is 11504 cycles. For the other columns of Table I, we estimate the number of cycles by assuming the load and multiply-accumulate latency and dividing the remaining cycles by the number of PRF lanes.

In Table II we show the potential PRF speedup using the values in Table I. The Absolute speedup is estimated using the single-lane PRF as the baseline. For 256 lanes, we forecast a speedup of the convolution algorithm of 73 times. The relative speedup is useful for estimating the efficiency of a multi-lane PRF implementation, and measures the performance improvement when doubling the number of PRF lanes. We estimate that a 32-lane PRF is 1.76 times faster than a 16-lane configuration. Furthermore, the efficiency of adding more lanes decreases below 50% with more than 64 lanes, as 128 lanes are only 44% faster than the 64-lane PRF-based system.

VII. Conclusions and Future Work

This paper described the integration of Polymorphic Register Files in a state of the art dataflow computing system. In particular, a compiler-based methodology is presented to extract the information provided by the designer, integrate and customize the corresponding registers and properly modify the computational kernels and exploit the parallel memory accesses. Our analytical estimations show that PRFs can potentially speed up the convolution algorithm on dataflow computing platforms by one order of magnitude. Future research is towards refinements of this semi-automated process, also including automatic identification of the variables to be stored in Polymorphic Register Files.

Acknowledgments

This work was partially funded by the European Commission in the context of the FP7 FASTER project (#287804).

References

[1] M. Gschwind, H. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki, "Synergistic Processing in Cell's Multicore Architecture," *IEEE Micro*, vol. 26, no. 2, pp. 10–24, 2006.

[2] "Maxeler MaxWorkstation." [Online]. Available: www.maxeler.com/products/desktop/

[3] C. Tomas, L. Cazzola, D. Oriato, O. Pell, D. Theis, G. Satta, and E. Bonomi, "Acceleration of the Anisotropic PSPI Imaging Algorithm with Dataflow Engines," in *Society of Exploration Geophysicists (SEG) Technical Program Expanded Abstracts*, 2012, pp. 1–5.

[4] H. Fu, W. Osborne, R. Clapp, O. Mencer, and W. Luk, "Accelerating Seismic Computations Using Customized Number Representations on FPGAs," *EURASIP Journal on Embedded Systems*, vol. 2009, no. 1, pp. 1–13, 2009.

[5] C. Ciobanu, "Customizable Register Files for Multidimensional SIMD Architectures," Ph.D. dissertation, Delft University of Technology, Delft, Netherlands, March 2013.

[6] C. Ciobanu, G. K. Kuzmanov, and G. N. Gaydadjiev, "Scalability Study of Polymorphic Register Files," in *Proc. of DSD*, 2012, pp. 803–808.

[7] ——, "On Implementability of Polymorphic Register Files," in *Proceedings of ReCoSoC*, 2012, pp. 1–6.

[8] A. Ramirez, F. Cabarcas, B. Juurlink, M. Alvarez Mesa, F. Sanchez, A. Azevedo, C. Meenderinck, C. Ciobanu, S. Isaza, and G. Gaydadjiev, "The SARC Architecture," *IEEE Micro*, vol. 30, no. 5, pp. 16–29, 2010.

[9] C. Ciobanu, G. K. Kuzmanov, A. Ramirez, and G. N. Gaydadjiev, "A Polymorphic Register File for Matrix Operations," in *Proceedings of SAMOS*, July 2010, pp. 241–249.

[10] C. Ciobanu, X. Martorell, G. K. Kuzmanov, A. Ramirez, and G. N. Gaydadjiev, "Scalability Evaluation of a Polymorphic Register File: a CG Case Study," in *Proceedings of ARCS*, 2011, pp. 13–25.

[11] C. Ciobanu and G. Gaydadjiev, "Separable 2D Convolution with Polymorphic Register Files," in *Proceedings of ARCS*, 2013, pp. 317–328.

[12] D. Kuck and R. Stokes, "The Burroughs Scientific Processor (BSP)," *IEEE Trans. on Computers*, vol. C-31, no. 5, pp. 363–376, May 1982.

[13] B. Juurlink, D. Cheresiz, S. Vassiliadis, and H. A. G. Wijshoff, "Implementation and Evaluation of the Complex Streamed Instruction Set," in *Proceedings of PACT*, 2001, pp. 73 – 82.

[14] D. Panda and K. Hwang, "Reconfigurable Vector Register Windows for Fast Matrix Computation on the Orthogonal Multiprocessor," in *Proceedings of ASAP*, 5-7 1990, pp. 202 –213.

[15] J. Corbal, R. Espasa, and M. Valero, "MOM: a Matrix SIMD Instruction Set Architecture for Multimedia Applications," in *Proceedings of the ACM/IEEE SC99 Conference*, 1999, pp. 1–12.

[16] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, "Matrix Register File and Extended Subwords: Two Techniques for Embedded Media Processors," in *Computing Frontiers '05*, May 2005, pp. 171–180.

[17] J. H. Derby, R. K. Montoye, and J. Moreira, "VICTORIA: VMX Indirect Compute Technology Oriented Towards In-Line Acceleration," in *Proceedings of CF*, 2006, pp. 303–312.

[18] J. Park, S.-B. Park, J. D. Balfour, D. Black-Schaffer, C. Kozyrakis, and W. J. Dally, "Register Pointer Architecture for Efficient Embedded Processors," in *Proceedings of DATE*, 2007, pp. 600–605.

[19] J. Osburn, W. Anderson, R. Rosenberg, and M. Lanzagorta, "Early Experiences on the NRL Cray XD1," in *HPCMP Users Group Conference*, 2006, pp. 347–353.

[20] "Cray XR1 Reconfigurable Processing Blade." [Online]. Available: www.cray.com/Assets/PDF/products/xt/CrayXR1Blade.pdf

[21] "SGI Altix 4700." [Online]. Available: www.sgi.com/pdfs/3867.pdf

[22] S. Stojanovic, D. Bojic, M. Bojovic, M. Valero, and V. Milutinovic, "An overview of selected hybrid and reconfigurable architectures," in *Proceedings of ICIT*, 2012, pp. 444–449.

[23] T. Brewer, "Instruction Set Innovations for the Convey HC-1 Computer," *IEEE Micro*, vol. 30, no. 2, pp. 70–79, 2010.

[24] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing Modular Hardware Accelerators in C with ROCCC 2.0," in *Proceedings of FCCM*, 2010, pp. 127–134.

[25] J. Nickolls and W. Dally, "The GPU Computing Era," *Micro, IEEE*, vol. 30, no. 2, pp. 56–69, 2010.

[26] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel Computing Experiences with CUDA," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, 2008.

[27] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos Mercurium: a Research Compiler for OpenMP," in *European Workshop on OpenMP (EWOMP'04)*, 2004, pp. 103–109.

[28] "The LLVM Compiler Infrastructure." [Online]. Available: llvm.org

[29] V. Podlozhnyuk, "Image Convolution with CUDA." [Online]. Available: developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf