The Case for Polymorphic Registers in Dataflow Computing

Cătălin Bogdan Ciobanu, Georgi Gaydadjiev, Christian Pilato & Donatella Sciuto

International Journal of Parallel Programming

ISSN 0885-7458

Int J Parallel Prog
DOI 10.1007/s10766-017-0494-1

Springer

Int J Parallel Prog
DOI 10.1007/s10766-017-0494-1

The Case for Polymorphic Registers in Dataflow Computing

Cătălin Bogdan Ciobanu[1,2] · Georgi Gaydadjiev[2,3] · Christian Pilato[4] · Donatella Sciuto[5]

Received: 10 May 2015 / Accepted: 21 February 2017

Abstract Heterogeneous systems are becoming increasingly popular, delivering high performance through hardware specialization. However, sequential data accesses may have a negative impact on performance. Data parallel solutions such as Polymorphic Register Files (PRFs) can potentially accelerate applications by facilitating high-speed, parallel access to performance-critical data. This article shows how PRFs can be integrated into dataflow computational platforms. Our semi-automatic, compiler-based methodology generates customized PRFs and modifies the computational kernels to efficiently exploit them. We use a separable 2D convolution case study to evaluate the impact of memory latency and bandwidth on performance compared to a state-of-the-art NVIDIA Tesla C2050 GPU. We improve the throughput up to 56.17X and show that the PRF-augmented system outperforms the GPU for 9×9 or larger mask sizes, even in bandwidth-constrained systems.

✉ Cătălin Bogdan Ciobanu
c.b.ciobanu@uva.nl; c.b.ciobanu@tudelft.nl

Georgi Gaydadjiev
g.n.gaydadjiev@tudelft.nl; georgi@maxeler.com

Christian Pilato
christian.pilato@usi.ch

Donatella Sciuto
donatella.sciuto@polimi.it

1 System and Network Engineering Group, University of Amsterdam, Amsterdam, The Netherlands

2 Distributed Systems Group, Delft University of Technology, Delft, The Netherlands

3 Maxeler Technologies Ltd, London, UK

4 Faculty of Informatics, University of Lugano, Lugano, Switzerland

5 Dip. di Elettronica, Informazione E Bioingegneria, Politecnico di Milano, Milan, Italy

Published online: 10 May 2017

Keywords Dataflow computing · Parallel memory accesses · Polymorphic register file · Bandwidth · Vector lanes · Convolution · High performance computing · High-level synthesis

1 Introduction

Heterogeneous High-Performance Computing (HPC) systems are becoming increasingly popular for data processing. For example, the Cell Broadband Engine [1] 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 are designed to excel in data-intensive computations, executing the most time-consuming parts of the applications. Another approach is to combine General Purpose Processors (GPPs) with specialized accelerators, implemented as custom chips or on reconfigurable devices, like Field-Programmable Gate Arrays (FPGAs). GPPs are mainly used for I/O and control-dominated functions, while the specialized components accelerate the application's computationally intensive parts. For example, the Maxeler MaxWorkstation [2] combines Intel x86 processors with multiple dataflow engines powered by, e.g., Xilinx Virtex-6 FPGA devices. This system adopts the dataflow computational model and organizes the data into highly regular streams flowing through the functions implemented in hardware, obtaining efficient implementations for streaming applications [3,4]. This platform allows the designer to focus on the high-level application development, while the corresponding HDL is generated using a dedicated Java-to-HDL high-level synthesis process by means of a proprietary compiler (MaxCompiler).

Generally, data-processing applications consist of relatively simple kernels, which are applied to large amounts of data. For example, image processing algorithms usually apply digital filters at some stages. These filters process the stream in blocks through "masks" (i.e., regular patterns for accessing the data). However, the existing dataflow systems are limited in terms of memory accesses to the *Local Store* (LS): this may become a serious bottleneck, heavily affecting the performance of the final system. In fact, the values on which the mask is applied are usually loaded sequentially from the LS, limiting the parallelism that can be extracted from the computational kernels. Therefore, multi-module parallel memory access schemes are of interest.

This work focuses on the integration of the **Polymorphic Register File** (PRF) in dataflow computing. The 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. However, the creation of PRF-based systems is not trivial. The efficient design of such systems demands: (i) a careful identification of the most-suited data structures and access patterns to configure the PRF, (ii) an analysis of the potential benefits to determine the architectural parameters, and (iii) a modification of the algorithms in order to take advantage of the newly-generated PRF memory. Designers currently need to perform the above steps by hand. This process is generally tedious and error-prone. In this context, this article aims at facilitating the

integration of PRFs into heterogeneous HPC systems for accelerating dataflow applications. We start from a C-based description of the accelerator where the designer can use custom pragmas to annotate the variables to be placed into the PRF and the corresponding memory access patterns. We then propose a semi-automatic methodology, which is integrated with existing tool-chains for creating the target streaming architecture. This methodology includes a compiler-based step, which is based on the LLVM compiler [6], to leverage the available high-level synthesis tools and generate the hardware accelerators able to interface with the properly customized PRF instance. Finally, we rely on the vendor-specific tools for the technology mapping, place and route. One crucial point for PRF-based systems is the latency of memory accesses to external data. Hence, we extensively analyze possible system-level organizations of the target architecture, along with the characteristics of the corresponding memory subsystem. In order to provide useful guidelines to the designers, we perform a comprehensive simulation-based exploration of different parameters (e.g., memory latency and bandwidth of the local scratchpads) in terms of the resulting performance. More specifically, the contributions of this article are:

- A methodology for enhancing existing dataflow architectures with PRFs in order to support parallel memory accesses, providing dedicated high-bandwidth memory interfaces to preselected data blocks that substitute sequential loads and stores;
- A compiler-based methodology supporting the automatic creation of these PRF-based systems to accelerate the access to specific variables;
- A comprehensive study of the Separable 2D Convolution, including:
  - An evaluation of the impact of memory latency on the estimated performance and efficiency of PRF-augmented designs;
  - An evaluation of the impact of the LS bandwidth on the PRF throughput. For small mask sizes, the PRF throughput is mainly constrained by the available LS bandwidth;
  - An efficiency study of multi-lane PRF designs. For each configuration, we quantify the relative throughput improvement when doubling the number of PRF lanes (e.g., the gain obtained by doubling the lane count from 8 to 16, 16 to 32, etc.). Our results suggest that PRFs with large number of lanes are more efficient when the mask size is large;
  - A comparison of the PRF throughput with the NVIDIA Tesla C2050 GPU. The results indicate that our PRF implementation is able to outperform the GPU for 9×9 or larger mask sizes. Depending on the mask size, the throughput improvements range between 1.95X and 25.60X in the case of bandwidth-constrained systems and between 4.39X and 56.17X in high bandwidth systems.

The article continues as follows: background information and related work are presented in Sects. 2 and 3, respectively. Section 4 describes our target PRF-based system, while Sect. 5 details our compiler-based methodology. Section 6 describes our case study, along with a detailed description of the PRF architecture and a sensitivity analysis on the architectural parameters. Finally, Sect. 7 concludes the article.

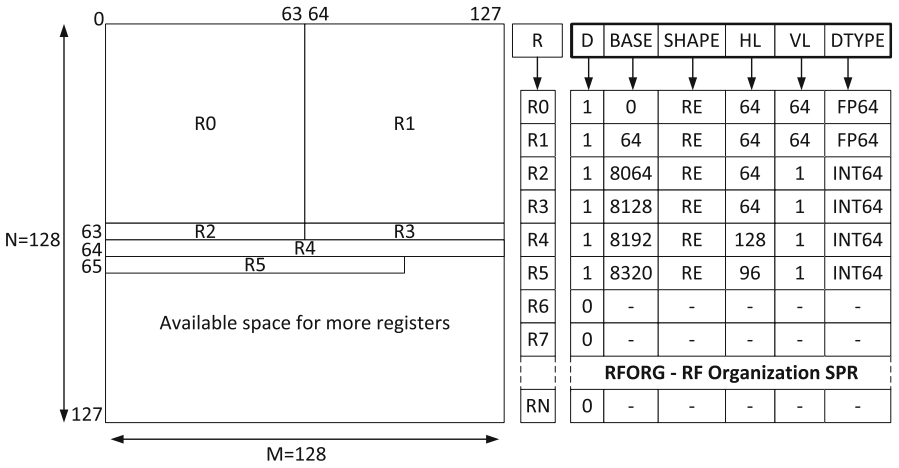

Fig. 1 The Polymorphic Register File

2 Background

The PRF was developed as part of the Scalable computer ARChitecture (SARC) project as its Scientific Vector Accelerator [7]. A PRF is a parameterizable register file, logically reorganized under software control to support multiple register dimensions and sizes simultaneously [8]. The total storage 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 into 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 Fig. 1. For each logical register, we need to specify its position (i.e. the base), the shape (e.g., rectangle, row, column, main or secondary diagonal), its dimensions (i.e. horizontal and vertical length) and the data type (e.g., integer or floating-point, 8/16/32/64 bits).

For the PRF hardware implementation, previous works considered a 2D array of $p \times q$[1] linearly addressable memory banks and used 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

[1] In the following, we use "×" to refer to a 2D matrix, and "·" to denote multiplication.

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 [5,9] and [10]. Specifically, results targeting the Virtex-7 XC7VX1140T-2 FPGA show feasible clock frequencies between 111 and 326 MHz and reasonable logic resources usage (less than 10 and 15% of the available LUTs and BRAMs, respectively). When targeting a 90 nm ASIC technology, the PRF clock frequency is between 500 and 970 MHz for storage sizes of up to 512 KB and up to 64 vector lanes. Power consumption remains reasonable, not exceeding 8.7 W and 276 mW for dynamic and static power, respectively [10].

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 an efficient utilization of the available bandwidth. This is achieved with a parallel access to multiple data elements and requires dedicated logic for its implementation. In fact, the most performance-efficient solution is to support this with parallel memory access schemes in hardware. The main 2D PRF benefits are:

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

Previous works show that, in some representative cases, PRFs can reduce the number of committed instructions by up to three orders of magnitude [8]. Compared to the Cell CPU, PRFs decrease the number of instructions for a customized, high performance dense matrix multiplication by up to 35X [7] and improve performance for Floyd and sparse matrix vector multiplication [8]. A Conjugate Gradient case study evaluated the scalability of up to 256 PRF-based accelerators in a heterogeneous multi-core architecture, with two orders of magnitude performance improvements [11]. Furthermore, potential power and area savings were shown by employing fewer PRF cores compared to a system with Cell processors. A preliminary evaluation of the PRF 2D separable convolution performance showed that PRFs can outperform state-of-the-art GPUs in terms of throughput [12].

3 Related Work

3.1 Data Organization

Efficient processing of multidimensional matrices has been targeted by several works. One approach is to use a memory-to-memory architecture, such as the Burroughs Scientific Processor (BSP) [13]. The BSP machine is 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. 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 to store the 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 [14] is a memory-to-memory architecture which allows the processing of two-dimensional data streams of arbitrary lengths with no data registers. 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 choose 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 relies 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) [15] 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 *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. These restrictions are not present in our PRF architecture, which provides a much higher degree of freedom for partitioning the register file. Therefore, our vector instructions can operate on matrices of arbitrary dimensions, reducing resizing overhead of *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 [16]. 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 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 does not 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) [17]. It supports eight 96-bit 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 [18] leverages 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 into a fixed number of equally-sized registers. This allows a higher degree of control when dynamically partitioning the register file.

The Register Pointer Architecture (RPA) [19] focuses on providing additional storage to a scalar processor, thus reducing the overhead associated with the updates 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 for the processor, while the RPs provide indirect access to the DRF. RPA is similar to our proposal as it also facilitates the indirect access to a dedicated register file by using dedicated indirection registers, even if the parameters stored in these registers are completely different. In RPA, each indirection register maps to a scalar element. In PRFs, one indirection register maps to a sub-matrix in the 2D register file. This is more suitable for expressing the data-level parallelism in multidimensional (matrix) vector processing.

3.2 Dataflow Architectures

Several FPGA-based HPC architectures have been recently proposed. The Cray XD1 [20] incorporates 12 AMD Opteron processors and six Xilinx Virtex-II PRO XC2VP30-6 or XC2VP50-7 FPGAs with dedicated QDR RAM and 3.2 GB/s interconnect. The Cray XR1 blade [21] is a dual-socket 940 design, incorporating one AMD Opteron processor in the first socket, 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 [22] platforms featured a modular blade design and incorporated the Non-Uniform Memory Architecture (NUMA) shared-memory NUMAflex architecture. The compute blade contained Intel Itanium processors, while dedicated memory, I/O and special-purpose blades were also available. The Reconfigurable Application Specific Computing (RASC) [23] blades facilitated two Virtex-4 LX200 FPGAs and dedicated memory DIMMs.

The Convey HC-1 [24] consists of two stacked 1 Unit chassis: one contains the processor, while the other one contains the FPGAs. The CPU chassis 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 HC-1 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 processor 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 our architectures can also be obtained with these alternative FPGA-based HPC systems.

3.3 Hardware Accelerator Design

Given the time-consuming and error-prone process of designing the HDL code for FPGA-based heterogeneous systems, several C-to-HDL compilers exist. The study in [25] provides a comprehensive overview of academic and industrial solutions for high-level synthesis (HLS). These tools are usually based on compilers (e.g., SUIF, LLVM or GCC) to produce Verilog/VHDL accelerators with interfaces for system-level integration [26]. However, these tools are usually limited in terms of memory accesses, requiring additional tools for the creation of multi-port memory systems [27]. Our approach is also built on the top of existing compilers (e.g., LLVM), but it does not explicitly perform HLS. Instead, it includes a preprocessing step that generates a modified C code ready for the HLS with existing vendor tools (e.g., Maxeler MaxCompiler or Xilinx Vivado HLS) to leverage the PRFs. Hence the manual intervention of the designer is minimized. Moreover, the programmer is not expected to add any additional C or HDL glue code to obtain a functional system.

Finally, for NVIDIA Graphics Processing Unit (GPU)-based [28] heterogeneous computing, the CUDA [29] 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. Unlike our approach, programmers still need to parallelize the algorithms in terms of threads and thread blocks, partition memory and schedule synchronizations.

4 Target System Organization

In domains such as HPC or embedded systems, applications usually process large amounts of data with fixed and relatively simple functionality. In these cases, all aspects of the computation can be determined at design time to specialize the system architecture. Designers are thus increasingly using heterogeneous systems in these domains, combining processor cores (for preparing the data and collecting the results for the users) and specialized hardware accelerators with customized memory subsystems for achieving energy-efficient high performance. In this context, modern HLS tools can create specialized hardware directly from a high-level language, mitigating the burden for the designer, who can focus on the algorithm development at a

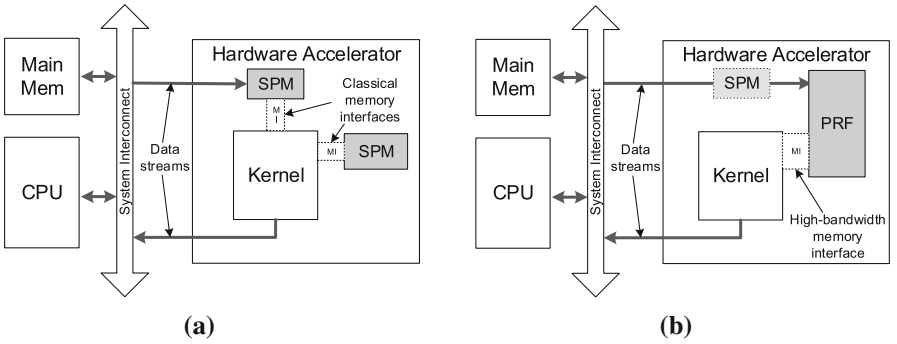

Fig. 2 System organization for streaming architectures. **a** Without PRF, **b** With PRF

higher level of abstraction. In the rest of this section we clarify the organization of the dataflow system and its enhancement with the PRF. For this, let us assume that we aim at implementing a computational kernel that operates on a data stream with a $N \times N$ mask (e.g., an image filter). The mask coefficients can be eventually updated at runtime so they are stored in memory instead of hard-coded in the accelerator logic.

A classical implementation for heterogeneous dataflow systems is shown in Fig. 2a. In these systems, a local *scratchpad memory* (SPM)[2] is used so that the accelerator can have low-latency access to the data. Specifically, SPMs can store the following types of data, based on the kernel requirements:

- *streaming data* they represent input/output data (e.g., N rows of the image flowing through the accelerator);
- *local data* they represent intermediate results or parameters (e.g., the set of coefficients used within the computational kernel).

In case of streaming data, the designer usually implements a circular buffer for storing these rows. To continue the computation, the kernel only requires a new row from the CPU, while $N - 1$ rows already stored in the SPM will be accordingly shifted. Local data, such as the coefficients, are usually represented as arrays in the original application and accordingly stored in the SPM at specific locations. In both cases, the kernel specifies which element is being accessed with a memory-based approach. For streaming data, elements are usually identified by offsets with respect to the current stream position (represented by a pointer to the circular buffer). For local data, elements are identified by their position in the array. This kind of architecture is limited by the available bandwidth between the SPM and the accelerator, as usually a small number of ports is available. Multiple read/write operations cannot be thus performed in parallel, and therefore must be serialized.

Polymorphic Register Files can be thus adopted in the architecture to enhance the performance of memory accesses. In case of streaming data, PRFs substitute the memory elements containing the stream values and must be customized in order to create local registers of proper sizes, i.e. the maximum number of values to be simultaneously stored. Additionally, when new data arrives, it is necessary to determine

[2] In the following, we will use Local Store (LS) and Scratchpad Memory (SPM) interchangeably.

where they must be stored based on the current PRF configuration and how to update the accesses to the current stored values. Thereafter, by collecting information on the input stream accesses and the corresponding values, it is possible to determine the memory access patterns and customize the PRFs accordingly. PRFs can be also used for storing local data, but without any shifting operations. The PRF registers are initialized with these values. Similarly to streaming data, it is possible to identify the kernel memory access patterns to perform multiple operations in parallel. Figure 2b illustrates the resulting system after PRF integration for streaming data. Compared to the reference implementation depicted in Fig. 2a, wider buses are available in the memory interface, effectively allowing the kernel to access multiple values in the same clock cycle. Note that hundreds of cycles are required to access the data directly received from the main memory through the interconnection system. To future overlap communication, a local SPM can be used between the interconnect and the PRF.

PRF-enhanced architectures can be easily created in different technologies, either FPGA or ASIC. For example, the Xilinx Zynq Evaluation Board [30] is based on the Zynq-7000 SoC and combines a dual-core ARM Cortex-A9 Processing System (PS) with a Xilinx XC7Z020 FPGA, also called Programmable Logic (PL). These are connected to SDRAM, Flash memory controllers and other peripheral blocks through the ARM AMBA AXI-based interconnect. The on-chip PS is attached to the corresponding Zynq device's PL through nine ARM AMBA AXI ports, allowing the CPU to send data to the local memory of the accelerators in few clock cycles. Additionally, Xilinx provides an end-to-end development kit to synthesize applications onto this architecture. *Xilinx Vivado HLS* is a commercial tool for HLS, which takes a C-based application as input and produces the corresponding RTL implementation. This generated IP can be then integrated with the rest of the system through *Xilinx Vivado*, which performs all the steps necessary to generate the bitstream. Finally, *Xilinx SDK* is an environment for developing the embedded applications that run on the CPU and interface with the hardware accelerators. Embedded BRAMs can be arranged as local SPMs on the PL, directly connected to the the kernel (see Fig. 3a). Otherwise, we expect future FPGAs to integrate high density PRF modules as hard macros, for low latency data access.

Similarly, the Maxeler MaxWorkstation [2] combines an Intel CPU with one or two Data Flow Engines (DFEs). The DFEs, based on state-of-the-art Xilinx Virtex-6 SX475T FPGAs, are connected to the Intel CPU via PCI Express. Furthermore, each DFE board contains up to 48 GB of DDR3 DRAM. The DFE implementation consists of one or more *Kernels* and a *Manager*, both written in a Java-based meta-language. The *Kernels* describe the data paths which implement the target algorithm, while the *Manager* describes the data flow between Kernels and with the off-chip stream I/O (e.g., CPU, DRAM). The application running on the CPU can be written in many high-level languages (e.g., C, Python, FORTRAN, etc.) and uses a set of API calls to communicate with the DFE components. The Maxeler run-time system (*MaxCompilerRT*) and *MaxelerOS* software libraries enable the communication between the CPU and the DFE, and abstract the low-level details of the DMA transfers over PCI Express. The *MaxCompiler* creates a DFE design through HLS and provides libraries that allow

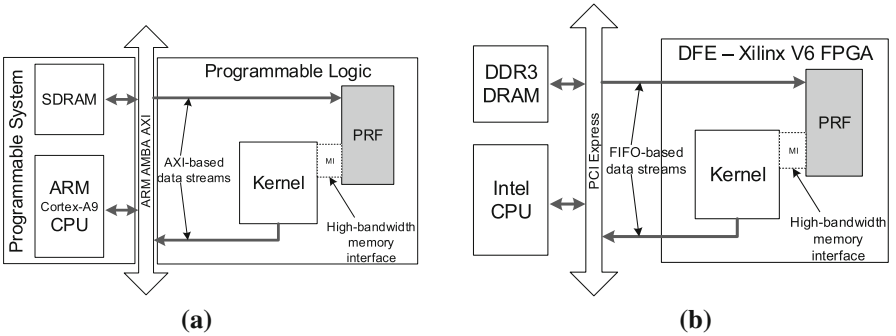

Fig. 3 System organization for streaming computation on different architectures. **a** Xilinx Zynq evaluation board, **b** Maxeler MaxWorkstation

the CPU to communicate with the kernels. For the device configuration, MaxCompiler leverages the Xilinx toolchain. Moreover, MaxCompiler allows hand-crafted HDL code to be connected to MaxJ Kernels. We leverage this to instantiate our PRF module (see Fig. 3b).

PRF-based architectures can be also created as custom chips, where the processor core is connected to the hardware accelerators via a standard or proprietary interconnection system (either bus or network-on-chip). For example, these specialized components can be generated by means of HLS with Cadence Stratus, starting from SystemC and targeting the given technology library. On the other hand, the internal PRF storage is Static RAMs (SRAMs) for fast accesses from the accelerators.

5 Proposed Methodology

This section describes the proposed methodology to automatically create a system augmented with PRFs (as described in Sect. 4). Our methodology starts from a C-based description of the application, where the designer manually introduced custom pragmas to specify relevant information on the variables to be stored in the PRF. The overall methodology is shown in Fig. 4, where the gray boxes highlight the steps proposed in this work: *Variable Extraction*, *PRF Customization* and *Code Generation*. In *Variable Extraction* (detailed in Sect. 5.1), the input C code is analyzed with a source-to-source preprocessing step based on LLVM that extracts the pragma-annotated variables. The step also produces an XML, which contains the list of variables and a description of the memory access patterns required for accessing them. *PRF Customization* (detailed in Sect. 5.2) starts from this XML file to generate the PRFs and allocate the identified variables onto them. These PRFs are then configured for providing values with the required access patterns. The corresponding HDL code is then generated, along with the wrapper to interact with the computational kernel. The *Code Generation* implements an additional compiler step to produce the input files for HLS starting from the information on PRF variables and access patterns identified during *Variable Extraction*. More specifically, sequential memory accesses are substituted by parallel PRF accesses as detailed in Sect. 5.3. This step is

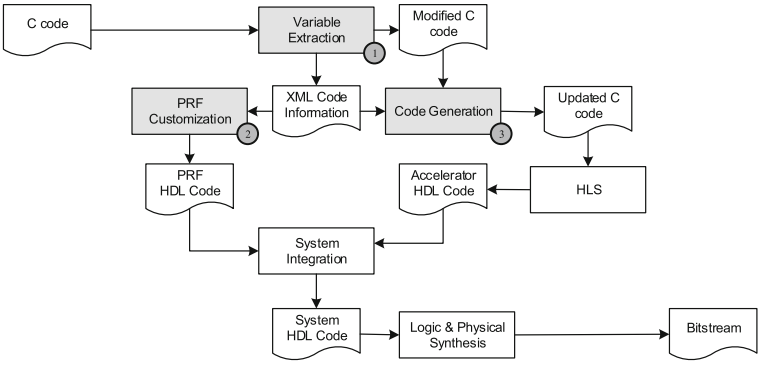

Fig. 4 Overall methodology for supporting automatic PRF integration

technology-dependent and it must be partially specialized based on the target architecture. For example, considering the examples described in above, this step generates the C code synthesizable with Xilinx Vivado HLS when targeting the Xilinx Zynq evaluation board, a Java-like kernel description ready for MaxCompiler when targeting the Maxeler MaxWorkstation or a SystemC-based description for HLS with Cadence C-to-Silicon. At this point, the *System Integration* creates the system-level description of the final architecture starting from the HDL code of the PRF and the computational kernel. Finally, vendor-specific tools are used for mapping, place and route.

5.1 Variable Extraction

This section describes our preprocessing step for automatically extracting the PRF variables from the input C code. We implement the analysis of the custom pragmas using the Mercurium source-to-source compiler [31]. To better understand how these annotations are specified and translated, let us assume that we have a computational kernel that performs a weighted average on three column pixels of a 64 $\times$ 64-pixel image. This can be represented in C with the piece of code shown in Listing 1, where the infinite loop is used to represent the processing of input streaming data as long as the CPU provides values.

For computing each output value, the kernel requires to read six different values from memory. Two types of pragma annotations are currently supported:

- `#pragma prf variable` it specifies the variable name and the space needed for its allocation in the PRF;
- `#pragma prf access` it specifies a memory access to a PRF block, along with information on the specific accessed element.

Listing 1 Example of C-based kernel implementing the weighted average on pixels.

```
void kernel(int* in, int* out)
{
   volatile int K[3] = {3, -1, 3};
   ...
   while(1)
   {
      ...
       out[i,j] = (K[0]*in[i-1][j] + K[1]*in[i][j] +
                                     K[2]*in[i][j]) /
(K[0]+K[1]+K[2]);
      ...
   }
}
```

The pragma `variable` is defined as follows:

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

The parameter `name` is the variable name. We assume that this variable can represent either stream or static data (`<type>=stream` or `<type>=static`, respectively). The parameter `size` instead represents the space to be reserved in the PRF for this variable. For example, considering Listing 1, the code is annotated as follows:

```
#pragma prf variable in 129 stream
#pragma prf variable K 3  static
```

Variable `in` represents a streaming data and the parameter `size` represents the number of consecutive elements of the input data stream to be simultaneously stored so that the computational kernel can operate. In our example, two consecutive rows plus one element (129 elements) of the image must be stored to allow the computation of each output value. When a new pixel is received, the data must be accordingly shifted. For the local variable `K`, this value is the size of the array itself, entirely stored into the PRF. In this case, the array must be initialized with a set of predefined values (e.g., `{3, -1, 3}`), which will be reported in the output XML file to properly initialize PRF registers. The pragma `access` is, instead, defined as:

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

The parameter `var` represents the PRF variable name, with the index specified by the second parameter. For streaming variables (e.g., variable `in`), the index represents the offset with respect to the current value and thus it can be either positive or negative. Conversely, for local data, it simply represents the position within the array. The last parameter specifies the name used to identify this access. For example, the operation in Listing 1 is annotated as follows:

```
#pragma prf access in -64 in0
#pragma prf access in 0 in1
#pragma prf access in 64 in2
#pragma prf access K 0 k0
#pragma prf access K 1 k1
#pragma prf access K 2 k2
```

```
out[i][j] = (K[0]*in[i-1][j] + K[1]*in[i][j]
 + K[2]*in[i+1][j]) ...;
```

The code is then transformed as follows to simplify the computational kernel code generation (see Sect. 5.3):

```
int in0 = in[pos-64];
int in1 = in[pos]
int in2 = in[pos+64]
int k0 = K[0];
int k1 = K[1];
int k2 = K[2];
out[i][j] = (k0*in0 + k1*in1 + k2*in2) ...;
```

5.2 PRF Customization

In this phase, the variables and the corresponding access patterns are analyzed to customize the PRF and generate the corresponding HDL code. Each block stored in the PRF is represented as a logical register, whose base address is used to access it from the kernel. We also need to determine the shape and the corresponding dimensions (horizontal and vertical length) based on the number of stored values and their data types. Moreover, when a block stores an array with initial values (e.g., the array `K` in our example), the corresponding registers are initialized with the values specified in the XML file.

5.3 Code Generation

At this stage, the PRF has been already customized based on the information provided by the *Variable Extraction*. Considering the simple example described above, the variable `in` has been moved to the PRF and 129 consecutive elements are stored at each moment. Each read operation will require three elements at specific locations, as shown in Fig. 5. As a consequence, a simple implementation of the PRF for this variable can provide a 96-bit interface to the kernel that is then unpacked to get the actual values of `in` for the computation. However, note that this is just an example of the implementation that can be obtained with our compiled-based methodology.

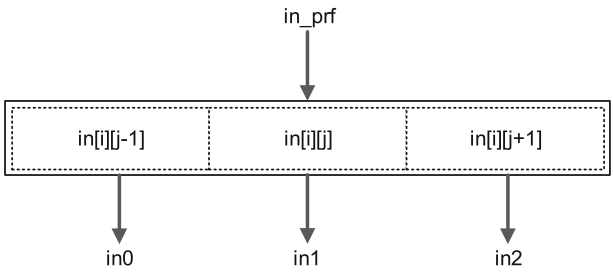

Fig. 5 Extraction of values from the parallel one provided by the PRF

More complex examples of PRF-based memory accesses can be found in Sect. 6.1. For this reason, we implemented another source-to-source transformation to support the subsequent HLS as a dynamic step in the LLVM compiler [6] (version 3.2). After the PRF load, the code is restructured as follows:

```
in_prf = read_prf(in, pos);
k_prf = read_prf(k);
int in0 = trunk_32(in_prf >> 64);
int in1 = trunk_32(in_prf >> 32);
int in2 = trunk_32(in_prf);
int k0 = trunk_32(k_prf >> 64);
int k1 = trunk_32(k_prf >> 32);
int k2 = trunk_32(k_prf);
out[i][j] = (k0*in0 + k1*in1 + k2*in2) ...;
```

where `read_prf` is a support function that reads multiple values in parallel from the PRF (based on the given configuration), while the function `trunk_32` generates a 32-bit value starting from its parameter value.

Even such a simple example shows the potential advantages of PRFs. In fact, in the resulting description, we require only two memory operations, which are implemented with customized parallel PRF accesses, for computing each output value (instead of six). The time for performing the unpacking operations and extract the values from the high-bandwidth memory interface is negligible because these are simply bit-select operations.

6 Case Study

In this section, we first introduce our case study, namely the Separable 2D Convolution, and a detailed description of the corresponding PRF architecture. We then present a comprehensive exploration of the different PRF parameters (e.g., memory latency and bandwidth) that can result from different organizations of the target architecture. This study includes an analytical PRF validation and simulation-based results when sweeping these values.

6.1 Vectorized Separable 2D Convolution

In this work, we use the Separable 2D Convolution as our case study. Convolution is used among others in image and video processing for filtering signal values. For example, Gaussian blur filters can be used to reduce the image noise and detail. Another good example is the Sobel operator, 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 a 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 later.

In digital signal processing, each output of the convolution is computed as a weighted sum of its neighboring data items. The coefficients of the products are defined by a mask (also known as *the convolution kernel*), which is applied to all elements of the input array. Intuitively, the 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 with data reuse.

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 ten-element 1D input $I = [\,20 \quad 22 \quad 24 \quad 26 \quad 28 \quad 30 \quad 32 \quad 34 \quad 36 \quad 38\,]$ and a three-element mask $M = [\,2 \quad \mathbf{5} \quad 11\,]$. 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 input edges, 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 article, we will refer to those as "halo" elements where a convention is made for their default values. If we consider all halo elements to be all 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 and the mask are 2D matrices. For example, let us consider the case of the following 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. 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. So, 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.

Assuming the 2D mask has **MASK_V** rows and **MASK_H** columns, **MASK_V** · **MASK_H** multiplications required to compute a single element. On the other hand, separable 2D convolutions (e.g., the Sobel operator) can be computed as two 1D convolutions on the same data. For example, in [32], 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 $[\,-1 \quad 0 \quad 1\,]$. Separable 2D convolutions are widely used because fewer arithmetic operations are required as 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 work, we focus on 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 need to transpose the 1D convolution outputs while processing the data. This can be performed conflict-free by using our Row Column (**RoCo**) scheme [9].

A vectorized separable 2D convolution algorithm, which avoids strided memory accesses when accessing column-wise input data, is introduced in [12]. The input matrix contains **MAX_V** × **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** × **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** $= 2 \cdot \mathbf{R} + 1$, where the convolution Radius **R** is a positive integer.

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** × **HSIZE** input elements;
2. Row-wise Convolution with **VSIZE** × **HSIZE**× **MASK_H** arithmetic operations (multiply-and-accumulate);

3. Column-wise Store **VSIZE** × **HSIZE** results;
4. Left Move **VSIZE** × **R** input elements used as halos by the next iteration;
5. If unprocessed data remains, go to step 1.

6.2 Polymorphic Register File Architecture

The main challenge when implementing the Polymorphic Register File in hardware is the design of the parallel memory used to instantiate the logical registers capable of supplying multiple data elements at each clock cycle. Furthermore, a realistic implementation requires multiple register file ports. It is important that our parallel memory design is implementable with practically feasible clock frequency and power consumption in contemporary technologies. We envision that the extra logic required to resolve dependencies between instructions does not contribute significantly to the total PRF area as compared to the 2D memory complexity. Furthermore, this additional logic will not appear on the critical path of the PRF hardware implementation [5]. Therefore, we implement a parallel memory corresponding to the left part of Fig. 1 only. We have not implemented the Special Purpose Registers or other additional logic needed in a complete PRF design.

The PRF contains $N \times M$ data elements, distributed among $p \times q$ physical memory modules, organized in a 2D matrix with p rows and q columns. Depending on the parallel memory scheme employed, such an organization allows the efficient use of up to $p \cdot q$ lanes. The data width of each memory module is *sram_width*. The number of lanes is $n\_lanes = p \cdot q$.

For simplicity, in Figs. 6 and 7, we only show the signals required for 1 read and 1 write ports. In the actual design, the signals are replicated according to the actual number of read and write ports. Figure 6(a) shows all inputs and outputs of our PRF design, and specifies the bit-width of each signal. The solid lines indicate the data signals, while the dashed lines are used for address and control signals. The inputs and outputs of our top level module, depicted in Fig. 6a, are:

1. *Prf_data_in* and *prf_data_out* the PRF input and output data, for *n_lanes* vector lanes and a data path *sram_width* bits wide;
2. *Memory_scheme* the parallel memory scheme, which can be one of the following: (a) Rectangle Only; (b) Rectangle Row; (c) Rectangle Col; (d) Row Col; and (e) Rectangle Transposed Rectangle;
3. *Read/write* control signals enabling reads and writes to the PRF;
4. *Read/write i, j* the upper left read/write block coordinate, $log_2 N / log_2 M$ bits wide;
5. *Read/write access type* the shape of the read/write block, which can be one of the following: (a) rectangle; (b) row; (c) column; (d) main diagonal; (e) secondary diagonal; and (f) transposed rectangle;
6. *Clock* the clock signal.

The Address Generation Unit (*AGU*) is shown in Fig. 6b. Starting from i and j—the upper left coordinates of the block being accessed and the access type (e.g., rectangle), the AGU computes the individual coordinates of all PRF elements which are accessed. For example, if a $p \times q$ rectangle is accessed, the AGU computes $p \cdot q$ pairs of values $(i + \alpha, j + \beta)\,, \alpha \in [0 \ldots p - 1]\,, \beta \in [0 \ldots q - 1]$.

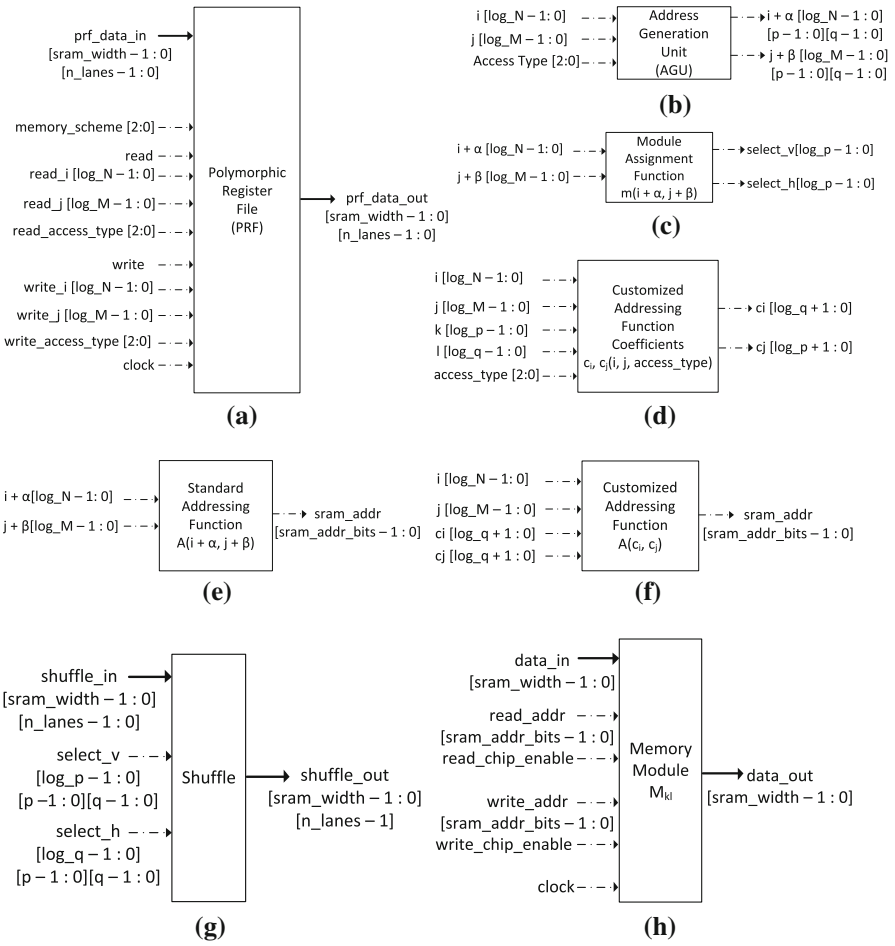

Fig. 6 PRF block modules. **a** PRF top module, **b** Address Generation Unit, **c** Module Assignment Function, **d** Customized Addressing coefficients, **e** Standard Addressing Function, **f** Customized Addressing Function, **g** Read/Write Shuffle, **h** SRAM modules

A PRF data element with coordinates $i + \alpha$, $j + \beta$ is assigned to one of the memory modules using the Module Assignment Function (*MAF*) m (Fig. 6c). The MAF computes the vertical and horizontal indexes of the corresponding memory module (*select_v* and *select_h*). The intra-module address is computed using one of the two Addressing Function modules A:

- The Standard Addressing Function (Fig. 6e) computes the intra-module address using the individual coordinates of all the elements which are being accessed ($i+\alpha$ and $j + \beta$, computed by the *AGU*);
- The Customized Addressing Function (Fig. 6f) only requires the coordinates of the accessed block (i and j), and two additional coefficients c_i and c_j. These coefficients are computed independently for each memory module using the coor-

Table 1 The ω constants

p/q	2	4	8
2	1	1	1
4	3	1	1

(a) ω_{q+1}

p/q	2	4	8
2	1	1	1
4	1	3	3

(b) ω_{q-1}

p/q	2	4	8
2	1	3	3
4	1	1	5

(c) ω_{p+1}

p/q	2	4	8
2	1	1	1
4	1	3	3

(d) ω_{p-1}

dinates of the block (i and j), the position of each memory module in the $p \times q$ matrix (denoted as k and l in Fig. 6d) and the access type (e.g., main diagonal). In order to compute these coefficients for the main and secondary diagonals, the ω constants are to be computed, which represent the multiplicative inverses of the pairs $(q + 1; p)$, $(q - 1; p)$, $(p + 1; q)$ and $(p - 1, q)$. Table 1 contains the ω constants for $p = 2 \ldots 4$ and $q = 2 \ldots 8$.

The Read and Write Shuffles (Fig. 6g) rearrange the inputs according to the select signals. The *shuffle_in* inputs are *sram_width* bits wide for each lane. The select signals (*select_v* and *select_h*), computed using the Module Assignment Function, specify the position of each output (*shuffle_out*). Figure 6h describes the Memory Modules, while the block diagrams of the read path for an 8 vector lane PRF, with $p = 2$ and $q = 4$, are shown in Fig. 7a. The inputs are the coordinates of the data block which is being read (i and j), and the shape of the block (set by *Access Type*). The data output (*PRF Data Out*) consists of 8 elements. The AGU computes the individual coordinates of all the data elements which are accessed and forwards them to the Module Assignment and the intra-module Addressing functions. The MAF m controls the read data shuffle, which reorders the data. Since accessing the memory modules introduces a delay of one clock cycle, the select signals for the data Shuffle block should be delayed accordingly. The Address Shuffle is required to provide the correct address to memory modules. Figure 7b depicts the PRF write path when using the customized Addressing Function. The data input (*PRF Data In*) has 8 elements and the same control signals as described above. In this case, the address shuffle is avoided, as the customized addressing function provides the correct address to each memory module using c_i and c_j. Figure 7c superimposes the standard and customized block diagrams. The shaded blocks, part of the standard design, are replaced by the c_i and c_j coefficients and the customized addressing function to simplify the organization.

FPGA Set-Up As a proof of concept, we implemented a PRF prototype design in SystemVerilog (excluding the Special Purpose Registers) with 2 read and 1 write ports with 64-bit data path, using Synplify Premier F-2011.09-1, and targeting a Virtex-7 XC7VX1140T-2. This prototype implementation uses full crossbars as read and write address shuffle blocks. We have coupled two dual-port BRAMs and duplicated the data in order to obtain 2 read and 1 write ports.

ASIC Set-Up We synthesized a PRF prototype design with two read and one write ports with 64-bit data path described in SystemVerilog, targeting the TSMC 90 nm technology. We used Synopsys Design Compiler Ultra version F-2011.09-SP3 in topographical mode, which accurately predicts both leakage and dynamic power with standard switching activity. In all experiments, the tool was configured to optimize for best timing. In all considered designs, the shuffle networks have been implemented using full crossbars. Using the Artisan memory compiler, a 1 GHz 256 × 64-bit dual-

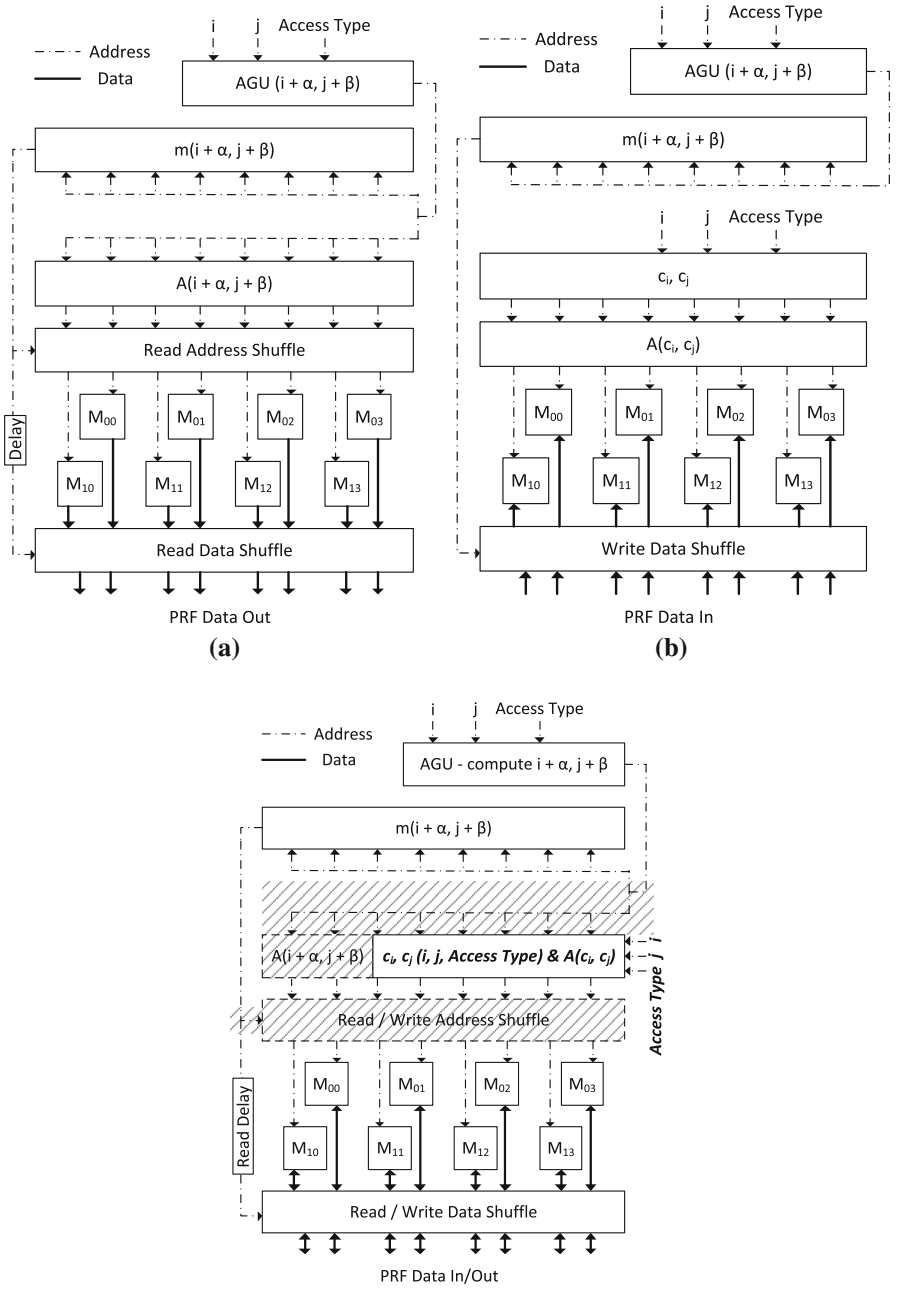

Fig. 7 PRF block diagrams, 8 lanes, p = 2, q = 4. **a** Standard Addressing Read, **b** Customized Addressing Write, **c** Superimposed Read / Write Standard and Customized Addressing

Table 2 128 KB PRF maximum clock frequency (MHz)

Target/number of lanes	8 (2 × 4)	16 (2 × 8)	32 (4 × 8)	64 (4 × 16)
TSMC 90 nm ASIC	909.1	847.5	813.0	613.5
VX1140T FPGA	194.0	165.4	118.9	–

Table 3 128 KB PRF Virtex-7 FPGA resource usage

Resource/number of lanes	8 (2 × 4)	16 (2 × 8)	32 (4 × 8)
BRAMs	64 (10%)	64 (10%)	96 (15%)
LUTs	4699 (<1%)	18207 (2%)	76406 (8%)

port SRAM register file was generated and used as the atomic storage element for our PRF. When the required capacity of the memory modules exceeded the available maximum of 256 64-bit elements, several SRAM modules were used together to aggregate the overall capacity. We coupled two dual-port SRAMs and duplicated their data in order to feature two read ports.

The synthesis results for TSMC 90 nm ASIC and Virtex-7 XC7VX1140T-2 FPGA for a 128 × 128 (128 KB) PRF with two read ports and one write port employing the customized **RoCo** memory scheme are presented in Tables 2 and 3. The difference in clock frequency between the ASIC and the FPGA implementations is 4.7X for a 8-lane PRF and 6.8X for a 32-lane PRF. Table 3 presents the usage of BRAMs and LUTs when targeting the FPGA. For configurations featuring up to 32 lanes, less than 10% of the available LUTs are used. The FPGA LUT usage grows quadratically when increasing the number of lanes, increasing up to 8% for the 32-lane PRF. For 128 KB PRFs, the BRAM usage varies between 10 and 15%.

6.3 Analytical Validation

We first analytically estimate the speed-up potentially introduced by the PRFs in the target architecture. 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. Our estimations assume the following row-wise convolution scenario: 32 × 32 × 64-bit elements block size (**HSIZE** = **VSIZE** = 32), 9 elements mask size (**R** = 4) and the multiply-and-accumulate latency is 12 clock cycles. In this experiments, we vary the average memory-load latency from 1 to 200 clock cycles.

Table 4 shows the estimated convolution execution time expressed in clock cycles, along with the estimated duration of the Load, Convolution, Move, and Store phases. In the baseline case (1 Lane), the PRF can only provide one data element per port at each clock cycle. This corresponds to using a simple serial memory for storing the input and output convolution data. For the baseline scenario and 100-cycle load latency, 1024 data elements need to be loaded, which takes 1124 clock cycles. The convolution requires $32 \cdot 32 \cdot 9 = 9216$ multiply-and-accumulate operations which consume 9228

cycles. Similarly, we estimate the duration of the halo moving stage as 128 cycles since $4 \cdot 32$ elements need to be moved. In this case, storing 1024 data elements is expected to take 1024 cycles. The combined duration of the four convolution steps is 11,504 cycles. The rows for the other memory latencies are computed in a similar fashion. For the other columns of Table 4, we estimate the number of cycles by considering the load and multiply-and-accumulate latencies and dividing the remaining cycles by the number of PRF lanes.

The absolute speedups for all scenarios presented in Table 4 are shown in Fig. 8a and the relative speedups are illustrated in Fig. 8b. The *absolute speedup* is estimated by using the single-lane PRF as baseline. The fastest configuration is the 256-lane, 1-cycle latency scenario, having a speedup of 197X. For the same number of lanes and a latency of 200 cycles, the estimated absolute speedup is 45X. For each scenario, we estimate the *relative speedup* of each PRF with respect to the one with half as many lanes (e.g., the 1-lane PRF is the baseline for the 2-lane relative speedup, 2-lane PRF is the baseline for the 4-lane PRF, etc.). The relative speedup is useful for estimating the efficiency of a multi-lane PRF implementation and measures the performance improvements when doubling the number of PRF lanes. As expected, Fig. 8b suggests that the memory latency becomes more important as the number of PRF lanes increases. A 256-lane PRF is estimated to be only 17% faster than the 128-lane configuration when the memory latency is 200 cycles. If the load latency would be reduced to 1 cycle, a 256-lanes PRF becomes 76% faster than a 128-lane configuration. Figure 8c shows the estimated PRF speedups for two load latencies: 11 cycles and 100 cycles. This corresponds to the expected latencies of the architectures presented in Fig. 2b with and without the local SPM, respectively. The absolute and relative speedups are illustrated with solid and dashed lines, respectively. The 100-cycle latency is representative for accessing data in DRAM, while the 11-cycle is closer to a scenario with on-chip memory [33]. In case of 256 lines, we forecast a speedup of 73X with a 100-cycle latency and a speedup of 168X with a 11-cycle latency. For the 100-cycle latency, a 32-lane PRF is 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 lanes. When the load latency is 11 cycles, a 32-lane PRF becomes 94% faster than a 16-lane configuration, with an efficiency for having 128 lanes of 79%. A 75% efficiency can be obtained for up to 32 lanes for a 100-cycle load latency. On the other hand, with a latency of 11 cycles, up to 128 lanes can be efficiently utilized (more than 75% of relative speedup). In the following, we focus on a scenario with an on-chip SPM having a latency of 11 cycles (i.e., the LS latency of the Cell processor [33]).

6.4 Simulation-Based Analysis of Architectural Parameters

For conducting our analysis, we use the single-core simulation infrastructure described in [5]. The PRF is implemented as part of the Scientific Vector Accelerator (SVA), which processes data from its Local Store (LS). We assume that all input data are present in the LS when the SVA starts processing. This situation is practically achieved with DMA transfers and double buffering, masking the communication overhead. Fur-

Table 4 Convolution estimated execution time (cycles)

Load latency (cycles)	Stage\lanes	1	2	4	8	16	32	64	128	256
1		1025	513	257	129	65	33	17	9	5
11		1035	523	267	139	75	43	27	19	15
50	Load	1074	562	306	178	114	82	66	58	54
100		1124	612	356	228	164	132	116	108	104
200		1224	712	456	328	264	232	216	208	204
–	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
1		11405	5709	2861	1437	725	369	191	102	58
11		11415	5719	2871	1447	735	379	201	112	68
50	Total	11454	5758	2910	1486	774	418	240	151	107
100		11504	5808	2960	1536	824	468	290	201	157
200		11604	5908	3060	1636	924	568	390	301	257

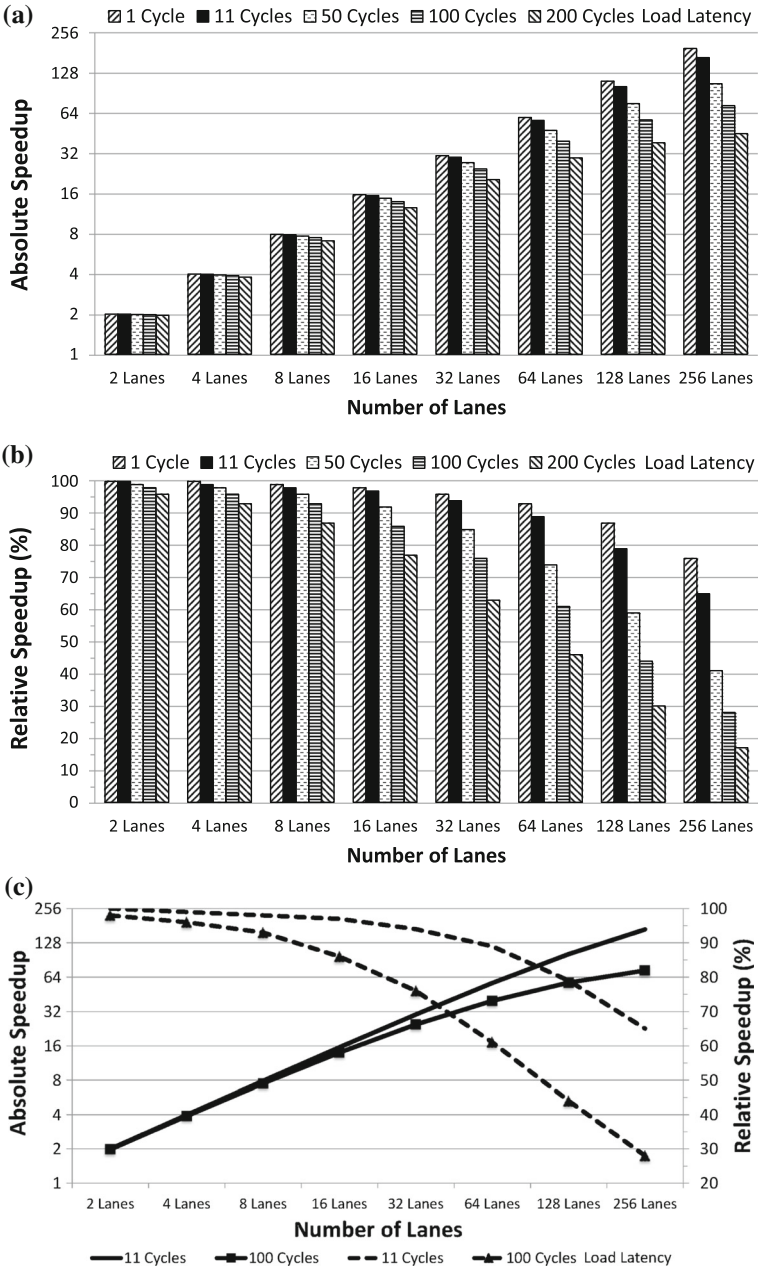

Fig. 8 Convolution Estimated Speedup. **a** Absolute Speedup, **b** Relative Speedup, **c** Combined Speedup

Table 5 Vector lanes range providing at least 75% efficiency

LS bandwidth / Mask Size	3 × 3	5 × 5	9 × 9	17 × 17	33 × 33
16 Bytes/Cycle	2	4	8	16	32
32 Bytes/Cycle	4	8	8	16	32
64 Bytes/Cycle	4	8	16	32	32
128 Bytes/Cycle	4	8	16	32	64
256 Bytes/Cycle	4	8	16	32	64

thermore, we assume only 1D contiguous vector loads and stores. Therefore, a simpler multi-bank LS, which uses low-order interleaving, can provide sufficient bandwidth for the PRF. We set the LS access latency to 11 cycles, taking into account the overhead incurred by the 1D vector memory accesses, and we vary the bandwidth between the PRF and the LS starting from 16 Bytes/Cycle, equal to the bus width used in the Cell processor between the Synergistic Processor Unit and its LS. Exploring these values corresponds to analyzing different architectural templates. This way, the designer can obtain practical guidelines on how to configure the PRF based on the characteristics of the target platform.

We compare our solution with the execution of the same 2D separate convolution on a NVIDIA Tesla C2050 GPU [34], which is based on the Fermi architecture. The C2050 has a 384-bit memory interface connected to a 3 GB off-chip GDDR5 memory clocked at 1.5 GHz, with a memory bandwidth of 144 GB/s. The maximum power consumption of the C2050 is 247 W. The C2050 GPU consists of 14 Streaming Multiprocessors, each one with 32 SIMD lanes (also known as CUDA cores), 64 KB of private RAM, and 768 KB unified L2 cache. The C2050 features a total of 448 SIMD lanes running at 1.15 GHz. This clock frequency is comparable to our ASIC synthesis results for the PRF [35]. Therefore, we express the throughput for both the PRF and the NVIDIA C2050 in terms of pixels/1000 cycles. The absolute throughput in pixels/s can be easily obtained by correlating our simulation results with the PRF configurations clock frequencies in Table 2 for both ASIC and FPGA technologies.

We study the throughput of multiple PRF configurations, ranging from 1 to 256 vector lanes. The peak throughput for the C2050 was obtained for an image size of 2048 × 2048 elements, which is the size we will use in our comparison study below. The details regarding the separable convolution implementation on the NVIDIA C2050 GPU used as a baseline here can be found in [32]. The input data for the PRF was set to 128 × 128 elements, as larger inputs did not additionally improve performance. The mask sizes are varied between 3 × 3 and 33 × 33 elements, representing realistic scenarios. For the PRF experiments, we selected $\mathbf{HSIZE} = \mathbf{VSIZE} = 32$.

Next, we present two pairs of figures for each considered LS bandwidth. The first one, which we will refer to as the A-set (Figs. 9a, 10a, 11a, 12a and 13a) depicts the absolute throughput measured in Pixels/1000 cycles. The second set of figures, referred to as the B-set (Figs. 9b, 10b, 11b, 12b and 13b), quantifies the relative throughput improvement when doubling the number of PRF lanes (e.g., the 16-lane PRF relative performance improvement uses the 8-lane PRF as its baseline) and is

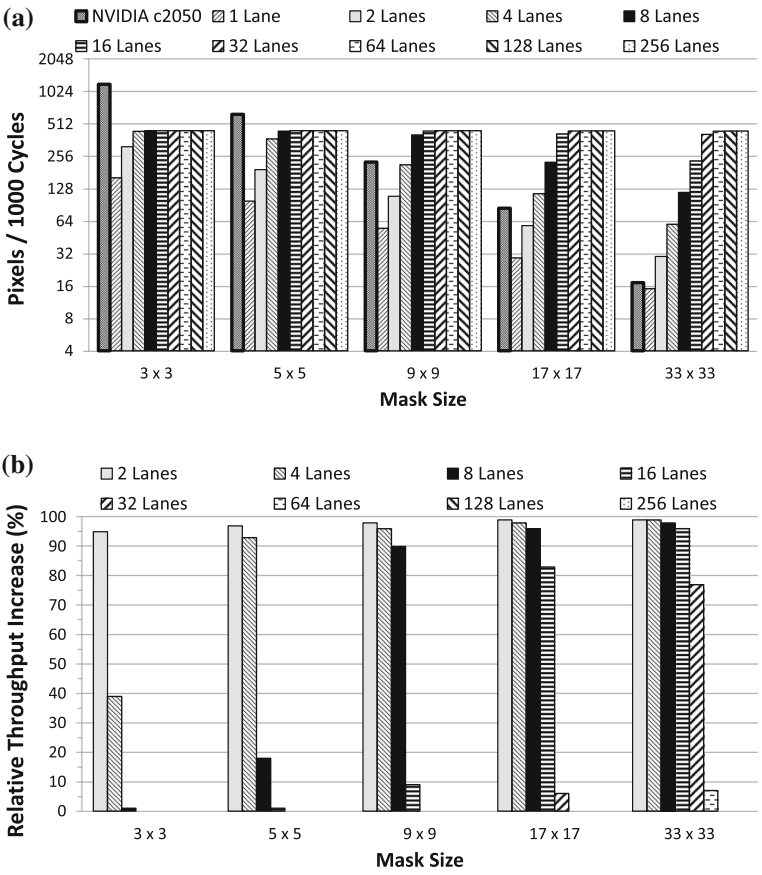

Fig. 9 Throughput, Input size = 128 × 128, LS BW = 16 Bytes/Cycle. **a** Throughput, **b** Relative performance improvement (in %)

measured in %. For the B-set figures, for each considered mask size, the baseline PRF configuration has half as many vector lanes. For example, for the 3 × 3 mask size, the baseline for the relative throughput increase of the 256-lane PRF is the 128-lane PRF. In addition, Table 5 quantifies the trends between LS bandwidth, mask size and multi-lane efficiency by summarizing the B-set of figures. For each LS bandwidth and each mask size, the highest number of lanes which can be used with an efficiency of at least 75% is shown. This represents the maximum configuration in which it was convenient to double the number of lanes with respect to the previous configuration.

With a LS bandwidth equal to 16 Bytes/Cycle, the GPU is faster than the PRF for small masks of 3 × 3 or 5 × 5 (Fig. 9a). In fact, the PRF is limited by the LS bandwidth when more than 8 lanes are used. However, as the mask size grows, the convolution becomes more computationally expensive and the throughput of the GPU decreases. Conversely, the PRF scales well with the number of lanes and outperforms the GPU starting from 9 × 9 masks. For a 256-lane PRF configuration, the throughput improvement varies between 1.95X (9 × 9 mask) up to 25.6X (33 × 33 mask). For

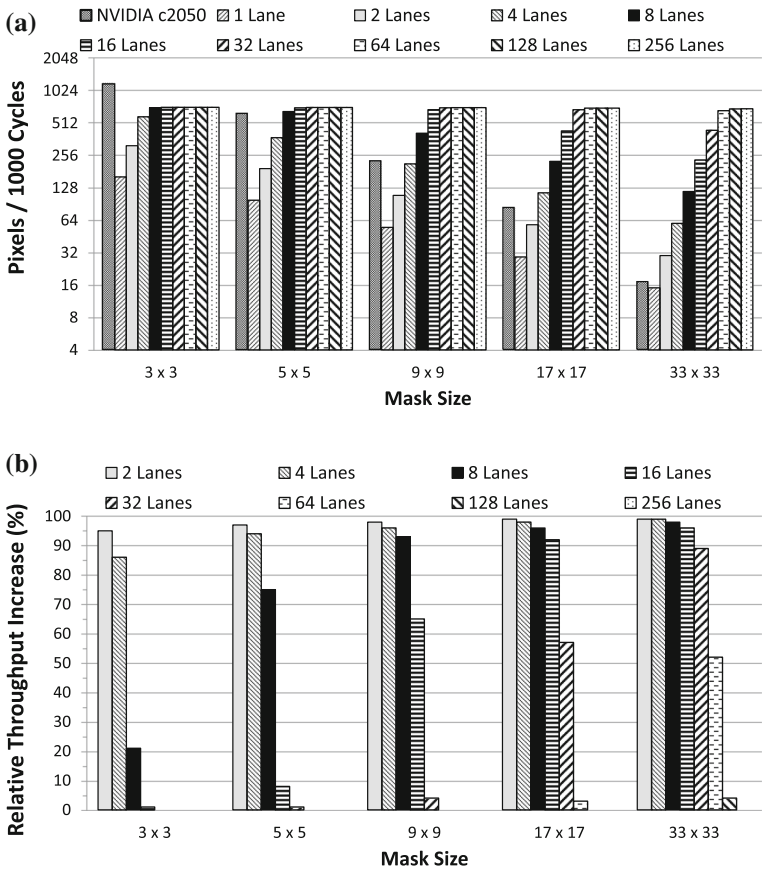

Fig. 10 Throughput, Input Size = 128 × 128, LS BW = 32 Bytes/Cycle. **a** Throughput, **b** Relative performance improvement (in %)

17 × 17 masks, as depicted in Fig. 9b, doubling the number of vector lanes from 8 to 16 increases throughput by 83%, compared to 9% in the 9 × 9 convolution. For the 33 × 33 mask, only the 1-lane PRF provides less throughput than the GPU. The PRF performance saturates with 32 lanes, and using 64 lanes can increase the throughput by only 7%. For the 3 × 3 mask, a 2-lane PRF is the most efficient (see Table 5). In case of 33 × 33 masks, up to 32 lanes can be efficiently used.

When doubling the LS bandwidth to 32 Bytes/Cycle, as shown in Fig. 10a, 8 vector lanes are sufficient for the PRF to match the GPU performance for a 5 × 5 mask, while only 4 lanes are required for 9 × 9 masks. When increasing the number of lanes from 2 to 4 for 3 × 3 masks, the throughput increases by 86%. The relative efficiency drops to 21% for 8 lines. For 17 × 17 masks, a 32-lane PRF provides 57% more throughput than 16 lanes, but then saturates as only 3% extra performance can be gained with 64 lanes. For the largest mask (i.e. 33 × 33), the PRF performance begins to saturate for configurations having 64 lanes. Doubling the number of lanes from 64 to 128 only increases throughput by 4%. Table 5 suggests that, compared to the 16 Bytes/Cycle

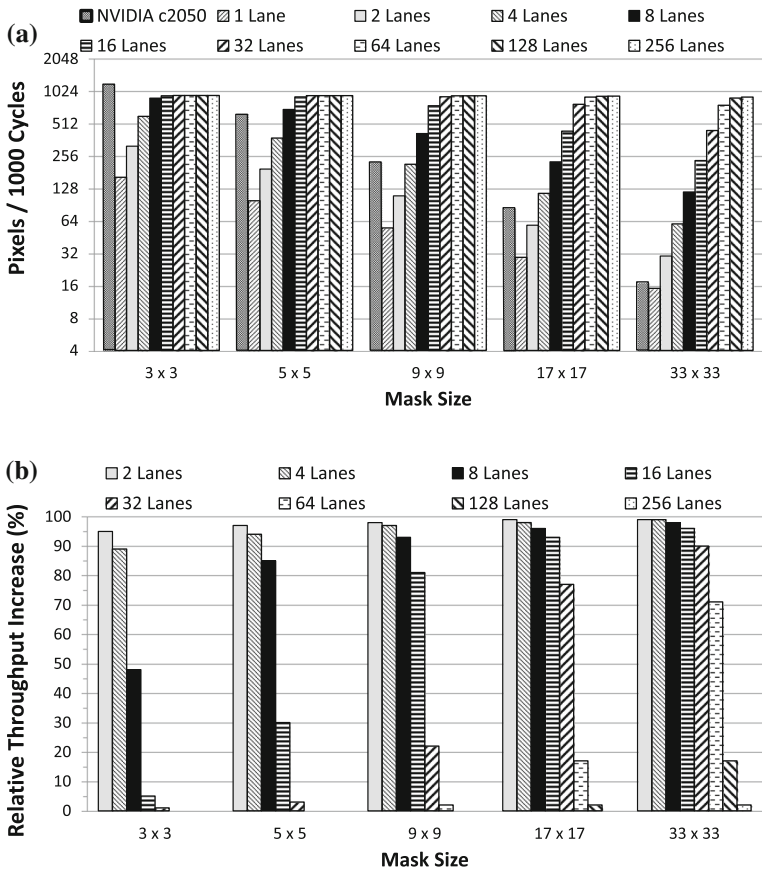

Fig. 11 Throughput, Input size = 128 × 128, LS BW = 64 Bytes/Cycle. **a** Throughput, **b** Relative performance improvement (in %)

bandwidth scenario, we can efficiently use up to 4 lanes for 3 × 3 masks, and 8 lanes for 5 × 5 masks. However, 16 lanes can be used with 65% efficiency for 9 × 9 masks, while 32 and 64 lanes achieve efficiencies higher than 50% for 17 × 17 and 33 × 33 masks, respectively.

When the LS bandwidth is set to 64 Bytes/Cycle as shown on Fig. 11a, the peak throughput of 16-lane PRFs more than doubles for 3 × 3 masks compared to the scenario where the LS provides only 16 Bytes/Cycle (Fig. 9a). In order to match the GPU throughput, 4 lanes are required for 9 × 9 masks. To outperform the GPU, 8 lanes are sufficient for 9 × 9 masks, while only 2 lanes are required for 33 × 33 masks. Figure 11b shows that the 16-lane PRF outperforms the 8-lane configuration by only 5% when using a 3 × 3 mask. The advantage is increased to 30% for the 5 × 5 mask and to 81% for the 9 × 9 case. For the 33 × 33 mask, the 256-lane PRF is around 2% faster than the 128-lane one, and 20% faster than the 64-lane configuration. Table 5 suggests that compared to the scenario where the LS delivers data at a rate of 32 Bytes/Cycle

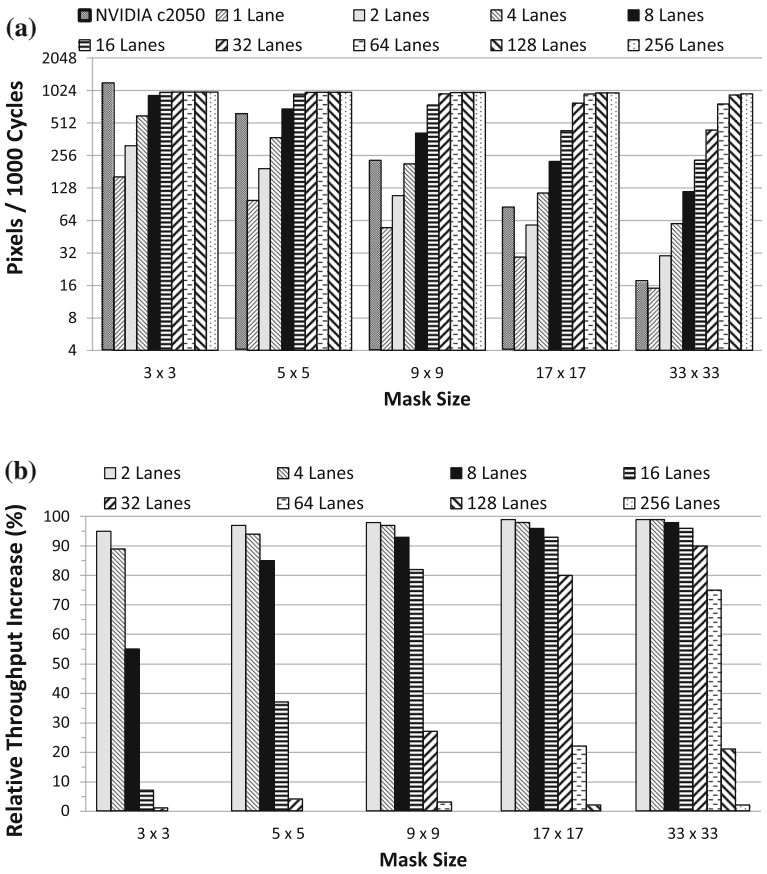

Fig. 12 Throughput, Input Size = 128 × 128, LS BW = 128 Bytes/Cycle. **a** Throughput, **b** Relative performance improvement (in %)

32 lanes can be efficiently used starting from 17 × 17 masks. A 64-lane configuration provides 71% more throughput than 32 lanes for 33 × 33 masks.

Figure 12a presents the scenario with a LS bandwidth of 128 Bytes/Cycle. For the 3 × 3 and 5 × 5 masks, performance saturates with 16 lanes, which offer only 7 and 37% extra throughput, respectively, compared to an 8-lane PRF. Figure 12b shows that for the 5 × 5 mask, switching from 16 to 32 lanes translates into only 4% performance improvement, and adding more lanes does not further increase the throughput. For 9 × 9 masks, using 32 lanes increases the throughput by 27% compared to the 16-lane PRF. When using larger masks, doubling the number of lanes up to 32 increases the throughput by 80 and 90% for the 17 × 17 and 33 × 33 masks, respectively. With 33 × 33 masks, the efficiency of adding more lanes starts decreasing from the 64-lane configuration which offers a 75% improvement. Doubling the number of lanes (from 64 to 128) increases the throughput by just 21%. Table 5 shows that, compared to a 64 Bytes/Cycle bandwidth, increasing the number of lanes from 32 to 64 improves the throughput by 75%.

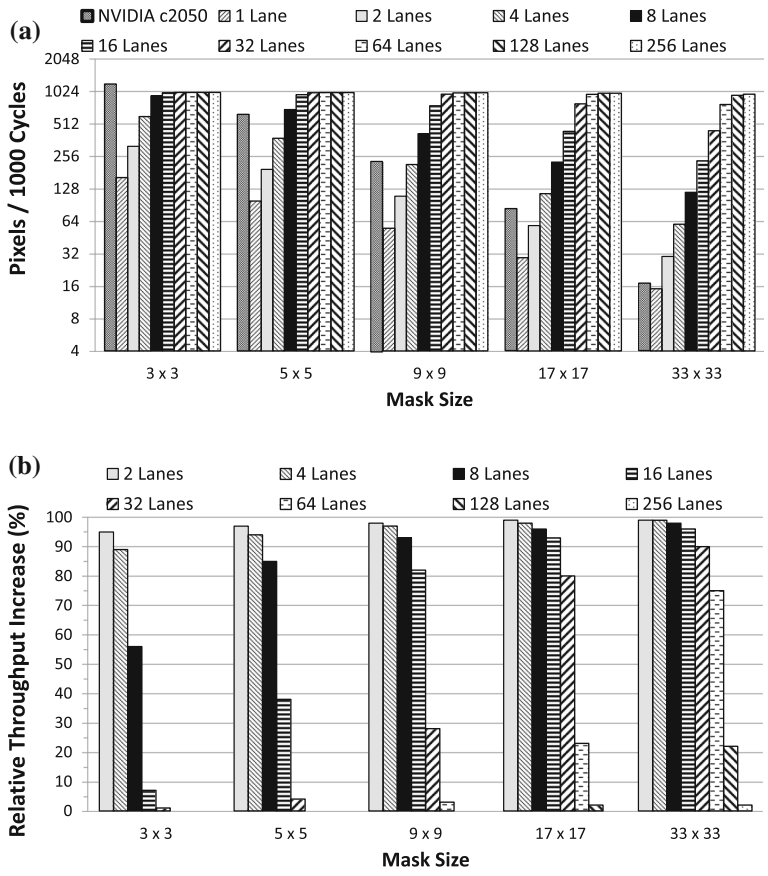

Fig. 13 Throughput, Input Size = 128 × 128, LS BW = 256 Bytes/Cycle. **a** Throughput, **b** Relative performance improvement (in %)

Figure 13a, b illustrate the scenario of a LS bandwidth of 256 Bytes/Cycle. The PRF performance only increases by less than 1% compared to the previous scenario (128 Bytes/Cycle). In this case, the PRF performance is mainly limited by the latency of the LS and, as a consequence, there are no improvements in increasing the bandwidth.

Figure 14a, b summarize the A-set Figs. 9a, 10a, 11a, 12a and 13a and illustrate the PRF throughput improvement using the NVIDIA C2050 GPU as the baseline. In these figures, the mask size is constant (9 × 9 and 33 × 33, respectively) in order to determine whether the LS bandwidth represents a performance bottleneck for a specific PRF configuration. The two figures suggest that, for each LS bandwidth configuration, it is possible to identify a threshold after which there is little to no performance gain from increasing the number of lanes. This is summarized in Table 5.

For the 9 × 9 mask size (Fig. 14a), the highest throughput improvement for a 256-lane PRF ranges from 1.95X for a LS bandwidth of 16 Bytes/Cycle and 4.39X for the 256 Bytes/Cycle scenario. The results suggests that for PRF configurations with a small number of lanes (e.g., the 8-lane configuration), increasing the LS bandwidth

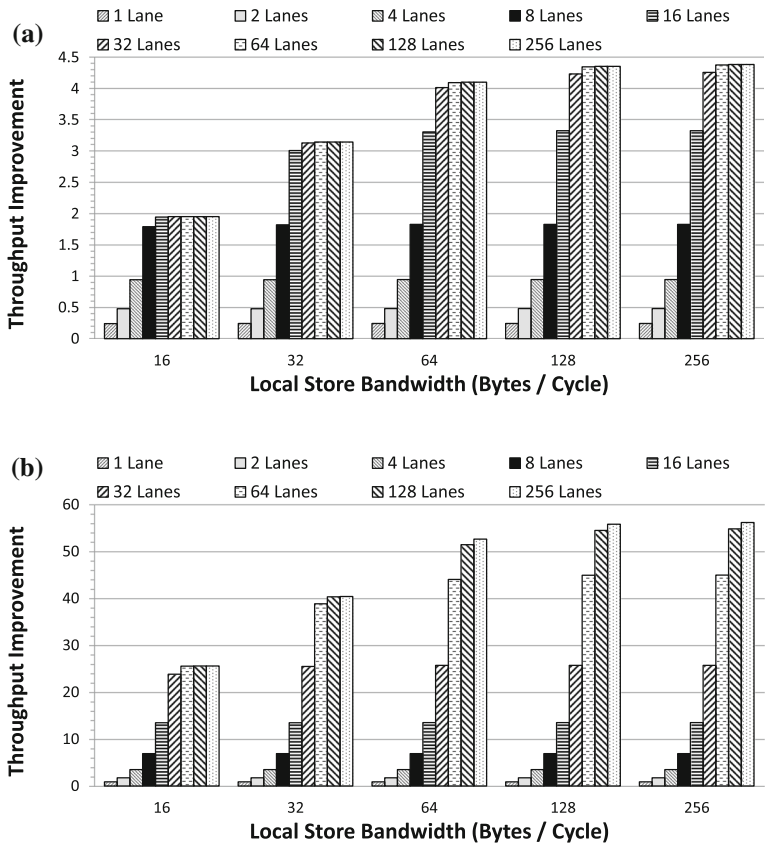

Fig. 14 Throughput improvement versus NVIDIA C2050 GPU, fixed mask size. **a** Mask Size = 9 × 9. **b** Mask Size = 33 × 33

will not improve the throughput. However, starting from the 16-lane PRF, there is a noticeable performance improvement when the LS bandwidth is increased up to 64 Bytes/Cycle. The results suggest that for 9 × 9 masks, increasing the LS bandwidth beyond 64 Bytes/Cycle does not offer performance advantages.

When the Mask Size is fixed at 33 × 33 (Fig. 14b), the 256-lane PRF improves the throughput of the NVIDIA GPU 25.60X for a LS bandwidth of 16 Bytes/Cycle and up to 56.17X for a bandwidth of 256 Bytes/Cycle. The 64 Bytes/Cycle case offers more than 93% of the performance of the 256 Bytes/Cycle scenario, a result similar to the 9 × 9 mask scenario. For 9 × 9 masks, a bandwidth larger than the baseline 16 Bytes/Cycle improves performance for PRFs with 16 or more lanes. However, for 33 × 33 masks, only PRF configurations with 64 lanes or more benefit from LS bandwidths larger than 16 Bytes/Cycle. This interesting result suggests that for separable 2D convolution, the point at which the LS bandwidth becomes a bottleneck depends not only on the number of PRF lanes but also on the specific mask size used. More specifically, the smaller the mask size used, the fewer PRF lanes can be employed without the LS bandwidth becoming the bottleneck.

Figure 14a, b suggest that, for 256-lane PRFs, scaling LS bandwidth from 16 to 256 Bytes/Cycle can increase the performance by a factor of 2.24X for 9×9 masks and 2.19X for 33×33 masks.

All of the above supports our claim that, depending on the configuration, the PRF brings large benefits for a system performing a separable 2D convolution of various mask sizes. The PRF can be proportionally adjusted to the instant requirements of the running algorithm. Moreover, the results discussed above confirm the close relationship between the mask size, the available LS bandwidth and the PRF scalability with the number of vector lanes. By increasing the mask size, the input data can be reused for more operations in the PRF, allowing the efficient use of multiple vector lanes. Depending on the mask size and the number of PRF lanes employed, the 2D convolution kernel can be memory bandwidth limited. The results show that for 256-lane PRFs increasing the LS bandwidth from 16 to 64 Bytes/Cycle leads to more than double improvement of the PRF throughput.

In a more generic scenario, unused vector lanes can be switched off in order to save power. Furthermore, the interface between the LS and the PRF can be customized in order to provide sufficient bandwidth for the high-end configurations with many vector lanes, or to save resources and power when fewer lanes are needed. This can further enhance the runtime adaptability of the computing system in combination with the ability to resize the vector registers in the PRF.

7 Conclusions

This article analyzed the impact of Polymorphic Register Files (PRFs) on state-of-the-art dataflow computing systems. First, it presents a semi-automatic methodology for integrating PRFs in existing architectures. Specifically, our compiler-based methodology extracts the information provided by the designer, integrates and customizes the corresponding registers accordingly, and properly modifies the computational kernels in order to exploit the parallel memory accesses. Next, we presented our separable 2D convolution case study to quantify PRF advantages in dataflow computing. We studied the impact of the memory-load latency. Our estimations suggested that PRFs can potentially speed up the convolution algorithm on dataflow computing platforms by up to two orders of magnitude. Furthermore, we evaluated the performance of the separable 2D convolution algorithm executing on multi-lane PRFs, and compared its throughput against a state-of-the-art NVIDIA Tesla C2050 GPU. We gained throughput improvements by up to 56.17X and showed that the PRF-augmented system outperforms the GPU for 9×9 or larger mask sizes, even in bandwidth-constrained systems. Furthermore, our experiments show that PRFs with large number of lanes are more efficient for large convolution masks. For small mask sizes, the 2D convolution kernel is mainly constrained by the available bandwidth to the Local Store.

Acknowledgements The authors would like to thank Wen-Mei W. Hwu and Nasser Salim Anssari from the University of Illinois at Urbana-Champaign for assisted us with obtaining the NVIDIA Tesla C2050 2D separable convolution results. This work was partially funded by the European Commission in the context of the FP7 FASTER Project (#287804) and the EU Horizon 2020 research and innovation programme under Grant No. 671653.

References

1. Gschwind, M., Hofstee, H., Flachs, B., Hopkin, M., Watanabe, Y., Yamazaki, T.: Synergistic processing in Cell's Multicore Architecture. IEEE Micro **26**(2), 10–24 (2006)
2. Maxeler MaxWorkstation. http://www.maxeler.com/products/desktop/
3. Tomas, C., Cazzola, L., Oriato, D., Pell, O., Theis, D., Satta, G., Bonomi, E.: Acceleration of the anisotropic PSPI imaging algorithm with Dataflow Engines. In: Society of Exploration Geophysicists (SEG) Technical Program Expanded Abstracts, pp. 1–5 (2012)
4. Fu, H., Osborne, W., Clapp, R., Mencer, O., Luk, W.: Accelerating seismic computations using customized number representations on FPGAs. EURASIP J. Embed. Syst. **2009**(1), 1–13 (2009)
5. Ciobanu, C.: Customizable Register Files for Multidimensional SIMD Architectures. PhD Thesis, Delft University of Technology, Delft, March (2013)
6. The LLVM Compiler Infrastructure. http://llvm.org
7. Ramirez, A., Cabarcas, F., Juurlink, B., Alvarez Mesa, M., Sanchez, F., Azevedo, A., Meenderinck, C., Ciobanu, C., Isaza, S., Gaydadjiev, G.: The SARC architecture. IEEE Micro **30**(5), 16–29 (2010)
8. Ciobanu, C., Kuzmanov, G.K., Ramirez, A., Gaydadjiev, G.N.: A Polymorphic Register File for matrix operations. In: Proceedings of SAMOS, pp. 241–249. July (2010)
9. Ciobanu, C., Kuzmanov, G.K., Gaydadjiev, G.N.: On implementability of Polymorphic Register Files. In: Proceedings of ReCoSoC, pp. 1–6 (2012)
10. Ciobanu, C., Kuzmanov, G.K., Gaydadjiev, G.N.: Scalability study of Polymorphic Register Files. In: Proceedings of DSD, pp. 803–808 (2012)
11. Ciobanu, C., Martorell, X., Kuzmanov, G.K., Ramirez, A., Gaydadjiev, G.N.: Scalability evaluation of a Polymorphic Register File: a CG case study. In: Proceedings of ARCS, pp. 13–25 (2011)
12. Ciobanu, C., Gaydadjiev, G.: Separable 2D Convolution with Polymorphic Register Files. In: Proceedings of ARCS, pp. 317–328 (2013)
13. Kuck, D., Stokes, R.: The Burroughs Scientific Processor (BSP). IEEE Trans. Comput. **C–31**(5), 363–376 (1982)
14. Juurlink, B., Cheresiz, D., Vassiliadis, S., Wijshoff, H.A.G.: Implementation and Evaluation of the Complex Streamed Instruction Set. In: Proceedings of PACT, pp. 73–82 (2001)
15. Panda, D., Hwang, K.: Reconfigurable Vector Register Windows for fast matrix computation on the orthogonal multiprocessor. In: Proceedings of ASAP, pp. 202–213, 5–7 (1990)
16. Corbal, J., Espasa, R., Valero, M.: MOM: a matrix SIMD Instruction Set Architecture for multimedia applications. In: Proceedings of the ACM/IEEE SC99 Conference, pp. 1–12 (1999)
17. Shahbahrami, A., Juurlink, B., Vassiliadis, S.: Matrix Register File and extended subwords: two techniques for embedded media processors. In: Computing Frontiers '05, pp. 171–180. May (2005)
18. Derby, J.H., Montoye, R.K., Moreira, J.: VICTORIA: VMX indirect compute technology oriented towards in-line acceleration. In: Proceedings of CF, pp. 303–312 (2006)
19. Park, J., Park, S.-B., Balfour, J.D., Black-Schaffer, D., Kozyrakis, C., Dally, W.J.: Register Pointer Architecture for efficient embedded processors. In: Proceedings of DATE, pp. 600–605 (2007)
20. Osburn, J., Anderson, W., Rosenberg, R., Lanzagorta, M.: Early experiences on the NRL Cray XD1. In: HPCMP Users Group Conference, pp. 347–353 (2006)
21. Cray XR1 Reconfigurable Processing Blade. http://www.cray.com/Assets/PDF/products/xt/CrayXR1Blade
22. SGI Altix 4700. http://www.sgi.com/products/servers/?/4000/configs.html
23. Stojanovic, S., Bojic, D., Bojovic, M., Valero, M., Milutinovic, V.: An overview of selected hybrid and reconfigurable architectures. In: Proceedings of ICIT, pp. 444–449 (2012)
24. Brewer, T.M.: Instruction set innovations for the convey HC-1 computer. IEEE Micro **30**(2), 70–79 (2010)
25. Nane, R., Sima, V.-M., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y.T., Hsiao, H., Brown, S., Ferrandi, F., Anderson, J., Bertels, K.: A survey and evaluation of FPGA High-Level Synthesis tools. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **35**(10), 1591–1604 (2016)

26. Villarreal, J., Park, A., Najjar, W., Halstead, R.: Designing modular hardware accelerators in C with ROCCC 2.0. In: Proceedings of FCCM, pp. 127–134 (2010)
27. Pilato, C., Mantovani, P., Di Guglielmo, G., Carloni, L.P.: System-level optimization of accelerator local memory for heterogeneous systems-on-chip. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **36**(3), 435–448 (2017)
28. Nickolls, J., Dally, W.: The GPU computing era. IEEE Micro **30**(2), 56–69 (2010)
29. Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., Volkov, V.: Parallel computing experiences with CUDA. IEEE Micro **28**(4), 13–27 (2008)
30. Xilinx Zynq-7000 All Programmable SoC User Guide. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM
31. Balart, J., Duran, A., Gonzlez, M., Martorell, X., Ayguad, E., Labarta, J.: Nanos mercurium: a research compiler for openmp. In: European Workshop on OpenMP (EWOMP'04), pp. 103–109 (2004)
32. Podlozhnyuk, V.: Image convolution with CUDA. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable
33. Eichenberger, A.E., et al.: Using advanced compiler technology to exploit the performance of the Cell Broadband Engine Architecture. In: IBM Systems Journal, pp. 59–84 (2006)
34. TESLA C2050 / C2070 GPU Computing Processor. Supercomputing at 1/10th of the cost. www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores
35. Ciobanu, C., Kuzmanov, G.K., Gaydadjiev, G.N.: Scalability study of Polymorphic Register Files. In: Proceedings of DSD, pp. 803–808 (2012)