Separable 2D Convolution with Polymorphic Register Files

Cătălin B. Ciobanu[1,2] and Georgi N. Gaydadjiev[1,2]

[1] Computer Engineering Laboratory,
EEMCS, Delft University of Technology,
The Netherlands
{c.b.ciobanu,g.n.gaydadjiev}@tudelft.nl
[2] Department of Computer Science and Engineering
Chalmers University of Technology,
Sweden
{catalin,georgig}@chalmers.se

Abstract. This paper studies the performance of separable 2D convolution on multi-lane Polymorphic Register Files (PRFs). We present a matrix transposition algorithm optimized for PRFs, and a 2D vectorized convolution algorithm which avoids strided memory accesses. We compare the throughput of our PRF to the nVidia Tesla C2050 GPU. The results show that even in bandwidth constrained systems, multi-lane PRFs can outperform the GPU for 9×9 or larger mask sizes.

1 Introduction

Processor designers consider various options to utilize the steadily increasing number of transistors of each new semiconductor technology generation [1]. Further increases of processor clock frequencies are infeasible, as current technology faces severe thermal and power constraints. In recent years, Chip Multiprocessor (CMP) designs became mainstream, along with accelerators targeting specific workloads (e.g., hardware support for encryption algorithms [2] and various Single Instruction Multiple Data Extensions (SIMD) [3] to exploit data level parallelism). Best performance is typically obtained by balancing single threaded performance and multi-processor scalability. When determining the characteristics of a new processor, the potential workloads are carefully examined. However, new, yet unknown workloads will appear in the future, making it close to impossible to provide a single solution. One possibility is to use reconfigurable hardware and runtime partial reconfiguration; ASIC solutions, however, are typically used to obtain the best performance.

When targeting vector architectures such as IBM 370 [4], General Purpose Processors (GPPs) with SIMD extensions such as Altivec [5] or Heterogeneous Multicores as the Cell Broadband Engine [6], the programs need to be optimized according to the width and number of the Vector Registers. In all these systems, the available register file storage is divided in a fixed number of equally sized registers. When a new design changes either the number or the width of the registers, software compatibility is broken and costly software adaptation effort is usually required. As part of the Scalable computer ARChitecture (SARC) [7], the Polymorphic Register File (PRF) [8] has been proposed to provide a relaxed way of programming high performance vector applications, compatible with both FPGA [9] and ASIC [10] technologies. The PRF is designed to be customizable to the various data structures, enabling the programmer to focus on the code functionality instead of describing complex, platform specific, data operations and transfers, while maintaining high performance levels. Contrary to the approach used in previous vector architectures, the PRF is able to dynamically divide the available register storage into multidimensional registers of arbitrary shapes and sizes at runtime. PRFs have been shown to be suitable for computationally intensive workloads such as the Conjugate Gradient (CG) method, Floyd, and dense matrix multiplication [8,7]. It was also suggested that PRFs can potentially save area and power in state of the art many-core systems [11]. The benefits of two-dimensional (2D) PRFs are: i) improved storage efficiency, as the number of registers, as well as their dimensions and sizes are dynamically customized to the workload requirements, and ii) performance gain, by greatly reducing the number of committed instructions.

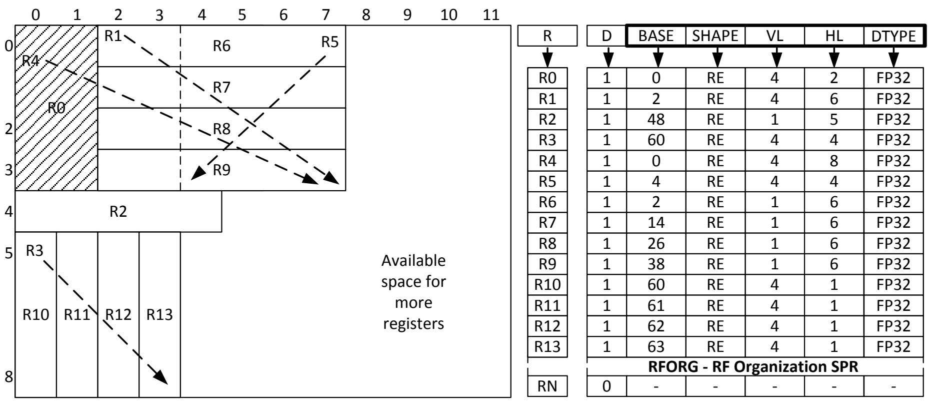
This paper studies the implementation of separable 2D convolutions using PRFs. More specifically, the main contributions of this paper are:

- A vectorized matrix transposition algorithm, optimized for PRFs;
- A vectorized separable 2D convolution algorithm utilizing our transposition, avoiding strided memory accesses while accessing column-wise input data;
- Performance evaluation of the separable 2D convolution kernel, comparing the throughput with the nVidia Tesla C2050 Graphics Processing Unit (GPU). Starting from mask sizes of 9×9 elements, the multi-lane PRFs outperform the GPU.

The remainder of this paper is organized as follows: the background information and related work are presented in Section 2. The experimental setup is introduced in Section 3. The 2D separable convolution kernel is introduced in Section 4. Our vectorized transposition and 2D convolution algorithms are described in Section 5, and the experimental results are studied in Section 6. Finally, Section 7 concludes the paper.

2 Background and Related Work

A PRF is a parameterizable register file, logically reorganized under software control, by the system / application programmer or by the runtime system, to support multiple register dimensions and sizes simultaneously [8]. Figure 1 provides an example of a 2D PRF of $\mathbf{N} = 9$ by $\mathbf{M} = 12$ elements, containing 14 registers defined by the Special Purpose Registers (SPR) contents. For each vector register it is necessary to specify the location of the upper left corner (BASE), the shape of the register (REctangular, Main Diagonal or Secondary Diagonal), the dimensions (Horizontal and Vertical Lengths) and the data type



R	D	BASE	SHAPE	VL	HL	DTYPE
R0	1	0	RE	4	2	FP32
R1	1	2	RE	4	6	FP32
R2	1	48	RE	1	5	FP32
R3	1	60	RE	4	4	FP32
R4	1	0	RE	4	8	FP32
R5	1	4	RE	4	4	FP32
R6	1	2	RE	1	6	FP32
R7	1	14	RE	1	6	FP32
R8	1	26	RE	1	6	FP32
R9	1	38	RE	1	6	FP32
R10	1	60	RE	4	1	FP32
R11	1	61	RE	4	1	FP32
R12	1	62	RE	4	1	FP32
R13	1	63	RE	4	1	FP32
	RFORG - RF Organization SPR					
RN	0	-	-	-	-	-

Fig. 1. The Polymorphic Register File (First iteration), N=9, M=12

(DTYPE) - INTeger 8/16/32/64 bit or Floating Point 32/64 bit. The benefits of the PRF are:

- Potential performance gain, by greatly reducing the number of committed instructions, and increasing the number of data elements processed with a single instruction using multi-axis vectorization;
- Improved storage efficiency, as the number of vector registers, their dimensions and sizes are dynamically adjusted during runtime according to the workload requirements, optimizing the use of the available register storage;
- Reduced static code footprint, as fewer, higher level instructions are used to describe the data processing. The same binary instructions may be used regardless of the shapes, dimensions and data types of the vector registers.

Previous works indicate that by employing PRFs, the number of committed instructions may be reduced by up to three orders of magnitude [8]. Compared to the Cell processor, PRFs decrease the number of instructions for a customized, high performance dense matrix multiplication by up to 35 times [7] and improve performance for Floyd and sparse matrix vector multiplication [8]. A CG case study [11] 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.

The mathematical foundations for multi-lane PRF hardware implementations, as well as synthesis results for FPGA and ASIC technologies have been presented in [9] and [10]. The PRF data is stored using a 2D matrix of memory modules with p rows and q columns, enabling the efficient use of up to $p \cdot q$ parallel vector lanes [12]. As in those papers, here we will use "$\times$" to refer to a 2D matrix, and "$\cdot$" to denote multiplication. Five parallel access schemes have been considered for the hardware implementation of the PRF: the single-view Rectangle Only (**ReO**) scheme, which supports conflict free accesses shaped as $p \times q$ rectangles, suggested in [13], and a set of four multi-view schemes, supporting conflict free access to the most common vector operations for scientific and multimedia applications [9]: 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 if $p\%q = 0$ or $q\%p = 0$. Using 90nm ASIC technology, the PRF clock frequency varies between 500MHz and 970MHz for storage sizes of up to 512KB and up to 64 vector lanes. Estimated power consumption is also within reasonable limits, up to 8.7W dynamic and 276mW leakage [10].

Related Work: The efficient processing of multidimensional matrices has been targeted by other architectures as well. One approach is to use a memory to memory architecture, such as the Burrows Scientific Processor (BSP) [14]. Being optimized for executing Fortran code, the ISA composed of high level vector instructions with a large number of parameters. The arithmetic units were equipped with 10 registers which are not directly accessible by the programmer. The Polymorphic register file also creates the premises for a high level ISA, but can reuse data directly within the register file. The Complex Streamed Instructions (CSI) [15] approach did not make use of data registers. CSI allows the processing of 2D data streams of arbitrary length, but requires data caches to benefit from data locality. Our approach suggests the register file as a cost-effective alternative of high speed data caches.

The Vector Register Windows (VRW) [16] concept allows grouping of consecutive vector registers in a 2D window. However, one of the dimensions is fixed, contrary to our proposal. The Matrix Oriented Multimedia (MOM) [17] also uses a 2D register file, but with a fixed number of registers which used sub-word parallelism in order to store up to 16x8 elements. The PRF also supports sub-word level parallelism, but doesn't restrict the number or shapes of the two dimensional registers. A Modified MMX (MMMX) [18] supports 8 multimedia registers, each 96 bits wide, with matrix operations limited to only loads and stores.

The Register Pointer Architecture(RPA) [19] extends scalar processors by adding two additional register files - Dereferencible Register File (DRF) and the Register Pointers (RP). The DRF provides the storage space, while the RP provide indirect access to the DRF. The PRF also uses indirect accessing to a dedicated register file, but the RPA maps scalar registers, while in our proposal each indirection register points to a matrix, being more suitable for vectors.

In order to adjust the number of registers and the total size of the physical register file in a VLIW, FPGA partial reconfiguration is used in [20]. Our approach assumes fixed physical register file size, but at a higher level logical view, offers variable fragmentation of the storage space, eliminating many overhead instructions and costly partial reconfigurations, potentially improving performance. While partial reconfiguration is only available in FPGAs, the PRF does not rely on any specific hardware technology, therefore it can be successfully implemented in both ASICs and FPGAs.

Accelerators for 2D convolutions have been previously implemented in reconfigurable technology [21], as well as bit level [22] and defect tolerant [23] systolic arrays in ASIC.

3 Experimental Setup

We use the simulation infrastructure introduced in [8], which consists of a cycle accurate simulator written in Unisim [24], an extension of SystemC. The PRF is implemented as part of the SARC Scientific Vector Accelerator (SVA), a loosely coupled processor, controlled by a General Purpose Processor (GPP). The experimental results take into consideration the communication between the GPP and the SVA, which is performed by using a number of exchange registers, similar to the Molen processor [25]. We also consider the overhead instructions required to reconfigure the Polymorphic Registers. The separable 2D convolutions are executed entirely on the SVA. Parallel execution between the GPP and the SVA is not considered. The SVA cannot directly accesses the main memory - it can only process data from its Local Store (LS), similar to the Cell Synergistic Processor Units (SPU) [6]. We assume that all input data is present in the Local Store when the SVA starts processing, a situation that can be practically achieved by using DMA transfers and double buffering. Furthermore, we assume that all PRF configurations have the same clock frequency, regardless of the number of vector lanes. Therefore, the experimental results represent an upper bound with respect to performance. A detailed description of the simulation environment is available in [8].

The SVA sends the load and store requests to a Local Store Controller (LSC). While in [8], the LSC was able to handle complex memory requests such as 2D and strided accesses, in this work we assume a more realistic scenario: only 1D contiguous vector loads and stores are supported. Therefore, a simple multi-banked Local Store which uses low order interleaving can provide sufficient bandwidth to the PRF. In our experiments, we set the latency of the LS to 11 cycles, taking into account the overhead incurred by the 1D vector memory accesses, and the bandwidth between the SVA and the LS to 16 bytes, equal to the bus width used in the Cell processor between the SPU and the LS.

4 Separable 2D Convolution

In digital signal processing, each output of the convolution is computed as a weighted sum of 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. Because there are no data dependencies, all output elements can be computed in parallel.

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, considering a 6 element 1D input $I = \begin{bmatrix} 20 & 21 & 22 & 23 & 24 & 25 \end{bmatrix}$ and a 5 element mask $M = \begin{bmatrix} 1 & 2 & \mathbf{3} & 4 & 5 \end{bmatrix}$. The 1D convolution output corresponding to the 3rd input (22) is $1 \cdot 20 + 2 \cdot 21 + \mathbf{3} \cdot \mathbf{22} + 4 \cdot 23 + 5 \cdot 24 = 340$. Similarly, the output corresponding to the 4th input (23) is obtained by shifting the mask by one position to the right: $1 \cdot 21 + 2 \cdot 21 + \mathbf{3} \cdot \mathbf{23} + 4 \cdot 24 + 5 \cdot 25 = 355$.

If the same convolution algorithm is used to 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). For the rest of this paper we will refer to those elements as "halo" elements. In practice, a convention is made for a default value for halo elements. If we consider the halo elements to be 0, the output corresponding to the 5th input (24) is $1 \cdot 22 + 2 \cdot 23 + \mathbf{3} \cdot \mathbf{24} + 4 \cdot 25 + 4 \cdot 25 + \mathit{5} \cdot \mathit{0} = 240$.

In the case of 2D convolutions, both the input data as well as the mask are 2D matrices. Assuming the 2D mask has **MASK_V** rows and **MASK_H** columns, the number of multiplications required to compute one output element using for 2D convolution is **MASK_V** · **MASK_H**. Separable 2D convolutions (e.g., the Sobel operator) can be computed as two 1D convolutions on the same data, requiring only **MASK_V** + **MASK_H** multiplications for each output element. For example [26], 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 $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$. In this work, we will 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 multi-bank memory systems. In order to avoid the strided memory accesses, we propose to transpose outputs of the 1D transpositions while processing the data. This can be performed conflict free by using our **RoCo** memory scheme introduced in Section 2.

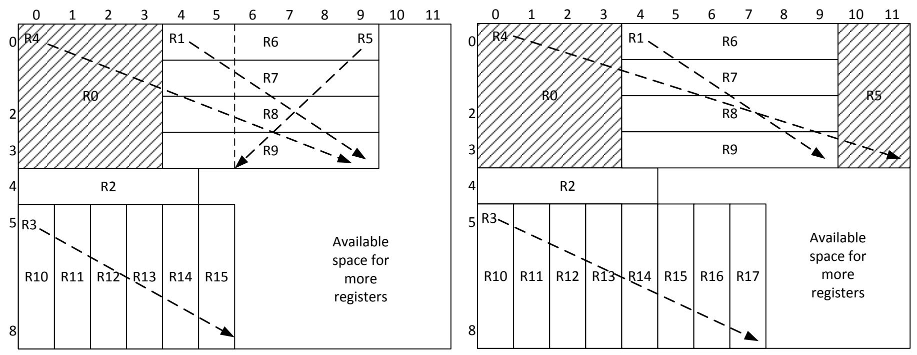
5 Vectorizing the 2D Convolution

In this Section, we first introduce the conflict free transposition algorithm. Then, we propose a 2D vectorized separable convolution algorithm for PRFs.

5.1 Conflict Free Transposition

In Figure 1, the dotted angled arrow is used to highlight the size of overlapping registers: R1, R4 and R5 overlap with R 6,7,8, 9, and R3 with R 10,11,12, 13. A block of input data of **VSIZE** = 4 rows and **HSIZE** = 6 columns is loaded from the LS in register R1, and the convolution result is stored in R3.

In order to perform the transposition, the data is loaded in the PRF into row registers, and stored from column registers. The input data consists of **VSIZE**



(a) Main iterations (b) Last iteration

Fig. 2. PRF configurations for the Separable 2D Convolution algorithm

rows containing **HSIZE** elements each - R 6,7,8 and 9, which are loaded from the LS as 1D accesses. R 10 - 13 single column registers are then stored using regular 1D accesses, effectively transposing the result. The Local Store will only send and receive data in consecutive addresses, fully utilizing the available bandwidth. For this example, if the PRF has at least 6 lanes and is implemented using the **RoCo** scheme introduced in Section 2, that allows conflict free accesses for both rows and columns, all the loads and stores used in the transposition can be performed conflict free. The input data matrix is processed in **VSIZE** $\times$ **HSIZE** blocks, and is stored transposed.

An additional requirement for the PRF is a modified auto-sectioning mechanism which stores the transposed output blocks in a top-to-bottom, left-to-right (column-wise). The regular auto sectioning instruction loads the input matrices in a left-to-right, top-to-bottom (row-wise) order. This is trivial to implement by slightly modifying the regular auto sectioning instruction **update2d**, which is described in detail [8]. Since two auto sectioning instructions are required, the one which handles the transposition must not perform a branch to the first instruction of the sectioning loop.

5.2 Our Implementation

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 both the horizontal and vertical axes. For clarity, we only present the code required to execute one convolution step. The code needs to be executed twice, once for the row-wise convolution and the second time for the column-wise one. However, only a small number of parameters needs to be updated (e.g., the pointer to the input and out matrixes) in the exchange registers, the binary instructions are similar for both passes.

The data will be processed in multiple steps (iterations), **VSIZE** rows at a time. Because special care needs to be taken at the borders of the input, we separate the vectorized algorithm in three distinct phases in order to properly process the first and last iterations. The layout for the PRF used in the first iteration is illustrated in Figure 1, for the last iteration in Figure 2(b) and for the rest in Figure 2(a) (to save space, we only show the first iteration SPR content). The shaded registers (R0 and R5) contain halo elements. Without loss of generality, we assume that $\mathbf{MAX\_V}\%\mathbf{VSIZE} = 0$ and $\mathbf{MASK\_H} = 2 \cdot \mathbf{R} + 1$.

Let $\mathbf{A} = \mathbf{MAX\_H}\%\mathbf{HSIZE}$ and $\mathbf{B} = \begin{cases} \mathbf{HSIZE}, & \text{if } \mathbf{A} = 0 \\ \mathbf{A}, & \text{otherwise} \end{cases}$

In all figures, $\mathbf{VSIZE} = 4$, $\mathbf{HSIZE} = 6$, $\mathbf{R} = 2$, $\mathbf{A} = 0$, and $\mathbf{B} = 6$.

For both Figure 1 and 2, the vector registers assignment is as follows:

- R0 contains the left hallo cells;
- R1 contains the input data which needs to be processed, and overlaps with R6-R9 which are used to load the data from the LS;
- R2 contains the mask;
- R3 stores the result of the convolution, and overlaps with R10-R17 which are used to transpose the result;
- R4 holds the halo elements as well as the loaded data;
- R5 holds the data which will become the halo of the next iteration, or the right halo cells for the last iteration.

The first iteration of the convolution takes into consideration the **R** halo elements to the left of the first input element. The algorithm performs the following steps:

```
f01. Define R2 as 1x(2*R+1), base=VSIZE
f02. Load R2
f03s.Define R0 as VSIZExR, base=0
f04. Initialize R0 with the default values for halos (e.g., 0)
f05. Resize R0 as VSIZEx2*R, base=0
f06. Define R4 as VSIZEx(HSIZE+R), base=0
f07. Define R1 as VSIZExHSIZE, base=R
f08. Define R6, R7,... R(6+VSIZE-1),as 1xHSIZE,
     base=R, R+M, ... R + (VSIZE-1)*M
f09. Define R(6+VSIZE),...R(6+VSIZE+HSIZE-R-1), as VSIZEx1,
     base=(VSIZE+1)*M, (VSIZE+1)*M+1, ... (VSIZE+1)*M + HSIZE-R-1
f10. Define R5 as VSIZEx2*R, base=HSIZE-R
f11. Define R3 as VSIZEx(HSIZE-R), base=VSIZE+1
f12. Load R6, R7,... R(6+VSIZE-1)
f13. Row-wise convolution: input=R4, Mask=R2, Output=R3
f14. Store R(6+VSIZE),...R(6+VSIZE+HSIZE-R-1)
f15. Move R5 to R0
f16. Update pointers to the input data and output data
```

The halo elements complicate the algorithm when processing the input data in blocks, as each new section of the data will require $2 \cdot \mathbf{R}$ elements from the previous iteration (R0 in Figure 2(a)). Our solution is to keep the halo elements in the PRF, and just move them to the left (from R5 to R0) before loading new data. This way, each input element is only loaded once for the horizontal pass and once for the vertical pass. The main iterations execute the following operations:

```
m01. Define R5 as VSIZEx2*R, base=HSIZE
m02. Define R1 as VSIZExHSIZE, base=2*R
m03. Define R4 as VSIZEx(HSIZE + 2*R), base=0
m04. Define R3 as VSIZExHSIZE, base=VSIZE+1
m05. Define R(6+VSIZE+HSIZE - R),...R(6+VSIZE+HSIZE - 1), as VSIZEx1,
     base=(VSIZE+1)*M+HSIZE-R, ... (VSIZE+1)*M + HSIZE-1
m06. Redefine R6, R7,... R(6+VSIZE-1),as 1xHSIZE, new base=2*R
m07s.Load R6, R7,... R(6+VSIZE-1)
m08. Row-wise convolution: input=R4, Mask=R2, Output=R3
m09. Store R(6+VSIZE),...R(6+VSIZE+HSIZE-1)
m10. Move R5 to R0
m11. Update pointers, continue if the last iteration follows
     or jump to instr. m07s otherwise
```

The last iteration needs to add the halo elements to the right of the last input element(Figure 2(b)). The width of the loads in the last iteration is **A**, and the number of stores is **A+R**. The pseudo-code for the last iteration is:

```
l01. Define R5 as VSIZExR, base=A+2*R
l02. Initialize R5 with the default values for halos (e.g., 0)
l03. Define R1 as VSIZExA, base=2*R
l04. Define R4 as VSIZEx(A+3*R), base=2*R
l05. Define R3 as VSIZEx(A+R), base=(VSIZE+1)
l06. Redefine R6, R7,... R(6+VSIZE-1),new size 1xA, new base=2*R
l07. Define R(6+VSIZE),...R(6+VSIZE+A+R-1) as VSIZEx1,
     base=(VSIZE+1)*M, ... (VSIZE+1)*M+A+R-1
l08. Load R6, R7,... R(6+VSIZE-1)
l09. Row-wise convolution: input=R4, Mask=R2, Output=R3
l10. Store R(6+VSIZE),...R(6+VSIZE+A+R-1),
l11. Update pointers and finish execution if last row
    or jump to instr. f03s otherwise
```

The algorithm also has two special cases, depending on the size of the input **MAX_H** and **HSIZE**. If $\mathbf{HSIZE} < \mathbf{MAX\_H} \leq 2 \cdot \mathbf{HSIZE}$, only the first and last iterations are executed. If $\mathbf{MAX\_H} \leq \mathbf{HSIZE}$, a single iteration is executed which processes full rows. Because of lack of space, we didn't include the corresponding pseudo-code and PRF configurations in this paper.

6 Experimental Results

The nVidia c2050 GPU [27] is running at 1.15 GHz, a frequency comparable to our ASIC synthesis results for the PRF presented in Section 2. Therefore,

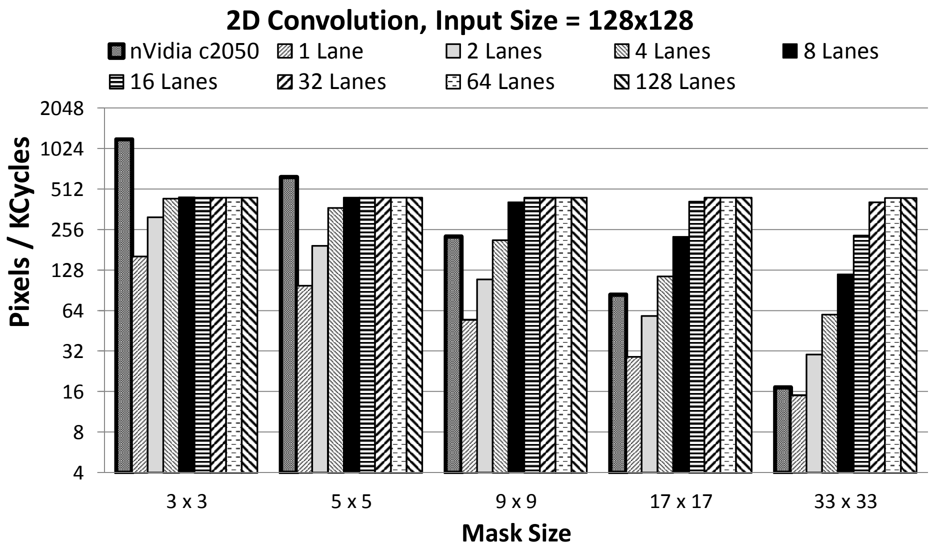


Fig. 3. 2D Convolution Algorithm Throughput

we measure the throughput for both the PRF and the nVidia c2050 in terms of pixels / 1000 cycles. Figure 3 compares the throughput of the c2050 card with multiple PRF configurations, ranging from 1 to 128 vector lanes. The peak throughput for the c2050 was obtained for an image size of 2048×2048 elements, which we will use for the comparison below. The input data for the PRF was set at 128×128 elements, as larger inputs did not improve performance. The mask sizes are varied between 3×3 and 33×33 elements, representing realistic scenarios. For the PRF experiments, we set **HSIZE** = **VSIZE** = 32.

The results suggest that for small masks of 3×3 or 5×5, the GPU is faster than the PRF, which is limited by the bandwidth to the Local Store when using more than 8 lanes.

However, as the masks increase in size, the convolution becomes more expensive in terms of computations, and the throughput of the GPU decreases. However, the PRF scales to a higher number of lanes, and starting from a mask size of 9×9, outperforms the GPU. For the largest mask size of 33×33, all but the slowest PRF configurations gain higher throughput than the GPU.

7 Conclusions and Future Work

We presented a matrix transposition algorithm optimized for PRFs, and a 2D vectorized separable convolution algorithm avoiding strided memory accesses when accesing the input data column-wise. We evaluated the performance of the vectorized algorithm executing on multi-lane PRFs, and compared the throughput with an nVidia Tesla C2050 GPU. The results show that even in a bandwidth constrained system, the PRF is able to outperform the GPU for 9×9 or larger mask sizes. As future work, we will evaluate the performance of the PRF with other computationally intensive workloads.

Acknowledgments. We thank Wen-mei W. Hwu and Nasser Salim Anssari for providing the nVidia Tesla C2050 GPU results.

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

References

1. ITRS: International Technology Roadmap for Semiconductors. Online, 2011 edn., `http://www.itrs.net/`
2. Akdemir, K., et al.: Breakthrough AES Performance with Intel AES New Instructions. White paper, 12 pages (June 2010), `http://communities.intel.com/docs/DOC-5003`
3. Gwennap, L.: Digital, MIPS Add Multimedia Extensions. Microdesign Resources 10(15), 1–5 (1996)
4. Buchholz, W.: The IBM System/370 vector architecture. IBM Systems Journal, 51–62 (1986)
5. Gwennap, L.: AltiVec Vectorizes PowerPC. Microprocessor Report 12(6), 1–5 (1998)
6. IBM. Cell BE Programming Handbook Including the PowerXCell 8i Processor, 1.11 edn. (May 2008)
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); ISSN 0272-1732
8. Ciobanu, C., Kuzmanov, G.K., Ramirez, A., Gaydadjiev, G.N.: A Polymorphic Register File for Matrix Operations. In: Proceedings of the 2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS 2010), pp. 241–249 (July 2010)
9. Ciobanu, C., Kuzmanov, G.K., Gaydadjiev, G.N.: On Implementability of Polymorphic Register Files. In: Proceedings of the 7th Int. Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2012), pp. 1–6 (2012)
10. Ciobanu, C., Kuzmanov, G.K., Gaydadjiev, G.N.: Scalability Study of Polymorphic Register Files. In: Proceedings of the 15th Euromicro Conference on Digital System Design (DSD 2012), pp. 803–808 (2012)
11. Ciobanu, C.B., Martorell, X., Kuzmanov, G.K., Ramirez, A., Gaydadjiev, G.N.: Scalability Evaluation of a Polymorphic Register File: A CG Case Study. In: Berekovic, M., Fornaciari, W., Brinkschulte, U., Silvano, C. (eds.) ARCS 2011. LNCS, vol. 6566, pp. 13–25. Springer, Heidelberg (2011)
12. Asanović, K.: Vector Microprocessors. PhD thesis, University of California at Berkeley (1998)
13. Kuzmanov, G., Gaydadjiev, G., Vassiliadis, S.: Multimedia rectangularly addressable memory. IEEE Transactions on Multimedia, 315–322 (2006)
14. Kuck, D.J., Stokes, R.A.: The Burroughs Scientific Processor (BSP). IEEE Transactions on Computers C-31(5), 363–376 (1982); ISSN 0018-9340

15. Juurlink, B.H.H., Cheresiz, D., Vassiliadis, S., Wijshoff, H.A.G.: Implementation and Evaluation of the Complex Streamed Instruction Set. In: Int. Conf. on Parallel Architectures and Compilation Techniques (PACT), pp. 73–82 (2001)
16. Panda, D.K., Hwang, K.: Reconfigurable Vector Register Windows for Fast Matrix Computation on the Orthogonal Multiprocessor. In: Proc. of the Int. Conference on Application Specific Array Processors, September 5-7, pp. 202–213 (1990)
17. Corbal, J., Espasa, R., Valero, M.: MOM: a Matrix SIMD Instruction Set Architecture for Multimedia Applications. In: Proceedings of the ACM/IEEE SC 1999 Conference, pp. 1–12 (1999)
18. Shahbahrami, A., Juurlink, B.H.H., Vassiliadis, S.: Matrix Register File and Extended Subwords: Two Techniques for Embedded Media Processors. In: Proc. of the 2nd ACM Int. Conf. on Computing Frontiers, pp. 171–180 (May 2005)
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 on Design, Automation and Test in Europe, DATE 2007, San Jose, CA, USA, pp. 978–973. EDA Consortium (2007) ISBN 978-3-9810801-2-4
20. Wong, S., Anjam, F., Nadeem, M.F.: Dynamically Reconfigurable Register File for a Softcore VLIW Processor. In: Proceedings of the Design, Automation and Test in Europe Conference (DATE 2010), pp. 969–972 (March 2010)
21. Wong, S.C., Jasiunas, M., Kearney, D.: Fast 2D Convolution Using Reconfigurable Computing. In: Proceedings of the Eighth International Symposium on Signal Processing and Its Applications, August 28-31, vol. 2, pp. 791–794 (2005)
22. Lee, J.-J., Song, G.-Y.: Super-Systolic Array for 2D Convolution. In: 2006 IEEE Region 10 Conference on TENCON 2006, pp. 1–4 (November 2006)
23. Hecht, V., Ronner, K.: An Advanced Programmable 2D-Convolution Chip for Real Time Image Processing. In: IEEE International Sympoisum on Circuits and Systems, vol. 4, pp. 1897–1900 (June 1991)
24. August, D., Chang, J., et al.: UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. IEEE Comput. Archit. Lett. 6(2), 45–48 (2007); ISSN 1556-6056
25. Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G., Panainte, E.M.: The molen polymorphic processor. IEEE Transactions on Computers 53(11), 1363–1375 (2004); ISSN 0018-9340.
26. Podlozhnyuk, V.: Image Convolution with CUDA. Online (June 2007), `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf`
27. TESLA C2050 / C2070 GPU Computing Processor. Supercomputing at 1/10th of the Cost. Online, `www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf`