The Architecture and the Technology Characterization of an FPGA-based Customizable Application-Specific Vector Processor

Jaroslav Sykora, Lukas Kohout, Roman Bartosinski, Leos Kafka, Martin Danek
Department of Signal Processing
Institute of Information Theory and Automation (UTIA) of the ASCR, v.v.i.
Pod Vodarenskou vezi 4, Prague, Czech Republic
{sykora, kohoutl, bartosr, kafkal, danek}@utia.cz

Petr Honzik
CIP plus s.r.o.
Milinska 130, Pribram, Czech Republic
petr.honzik@cip.cz

***Abstract*—The traditional approach to IP core design is to use simulations with test vectors. This is not feasible when dealing with complex function cores such as the Image Segmentation case-study algorithm in this paper. An algorithm developer needs to carry out experiments on large real-world data sets, with fast turn-around times, and in real time to facilitate performance tuning and incremental development.**

We propose a methodology called Application-Specific Vector Processor (ASVP). The ASVP approach first constructs a programmable architecture customized for a given application, then employs software techniques to develop firmware that implements the algorithm. Our sample implementation that supports the Image Segmentation kernel is capable of 332 MFLOPs, 400 MFLOPs, and 250 MFLOPs per coprocessor core in Virtex 5, Virtex 6 and Spartan 6 technologies, respectively. The core size is roughly 1500 slices, depending on the configuration and technology.

***Keywords*-Custom accelerators, vector processing, FPGA, DSP.**

I. Introduction

The mainstream design-entry languages of contemporary FPGAs are VHDL and Verilog. Synthesis from higher-level languages, such as C [1], OpenCL [2] or CUDA [3] is difficult for two closely related reasons: First, the FPGA design space is nearly infinite and a program in the high-level language can be implemented in many different ways, with wildly varying resource usage, total cycle count, and cycle time (operating frequency). It is very difficult for a compiler to strike the optimal configuration that minimizes the program execution latency (absolute time). Second, even if the optimal configuration is known, a single (re-)compilation process after an application source code modification may take tens of minutes up to several hours to complete, severely impacting the turn-around time of the application development and lowering the human programmer efficiency.

A high-level overview of our approach to designing custom FPGA-based accelerators is shown in Figure 1. We call our approach *Application-Specific Vector Processor* (ASVP); it is loosely based on the previous work [4] where it was called *Basic Computing Element* (BCE). In traditional work-flows (not shown) when the source code is modified by the developer, the accelerator must be recompiled and resynthesized to generate a new FPGA configware (bitstream) that can be verified. The drawback of the traditional approach is the long synthesis time caused mainly by a slow place&route process of the low-level tools. In our approach we abstract the custom accelerator into a specialized firmware-programmable architecture. In the first step the high-level source code is analyzed and domain features are extracted. Based on the required domain features a customized architecture is selected or newly built. The architecture is programmable by firmware to some extent, hence minor source code modifications can be tested quickly for they will trigger only fast

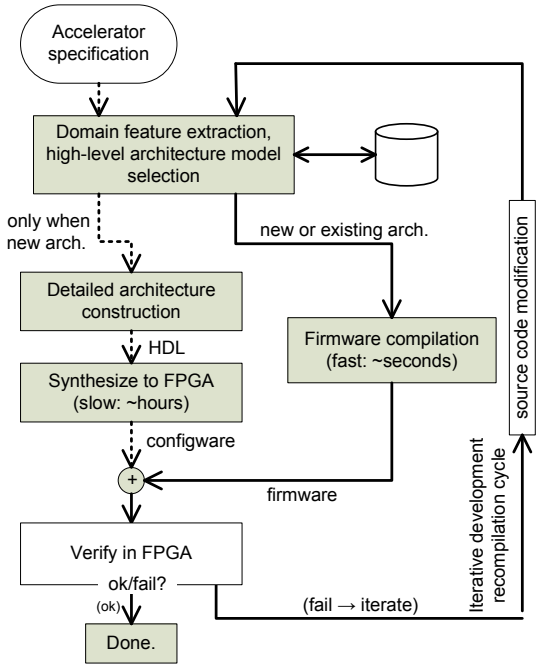

Fig. 1. The proposed custom accelerators development work-flow. Dashed lines indicate processes run once, full lines mark iterative paths.

firmware recompilation. Only when an architectural change is called for after a major source code modification, the costly re-synthesis process must be rerun.

In this work we propose an architecture that satisfies the following requirements:

- The architecture has to be customizable for different applications to take advantage of the FPGA reconfigurability.
- The architecture has to support both floating-point (long-latency) and integer (short-latency) operations found in DSP and video applications. The long-latency FP operations, combined with the common need of DSP applications to support vector reductions, pose some new challenges.
- Several hardware technology nodes of different characteristics should be supported without a need for manual software changes in the firmware. Specifically, the firmware programmer should be shielded from the impact of adapting the latency/frequency ratio of the hardware units to the target technology.

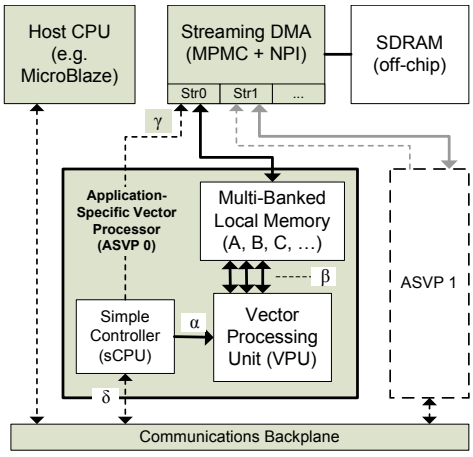

Fig. 2. A system-level organization of an ASVP-based core.

II. Related Work

FCUDA [3] is a source-to-source compiler from NVidia's CUDA language to synthesizable AutoPilot C. Building on FCUDA, a high-level parameter space exploration tool is presented in [5]. The tool first builds a resource and clock-period estimation models by actually synthesizing a small number of sample configurations. The models are used in a binary search heuristic to find the optimal FCUDA compiler/synthesis option setting.

Vector processing was shown to be a good match for embedded applications. In [6] the VIRAM vector processor was evaluated using the EEMBC multimedia benchmark suite, and it was found (for example) to outperform VLIW processors by a factor of 10. The original VIRAM chip is 0.180μm custom design clocked at 200 MHz, 1.6 GFLOP single-precision performance, with an on-chip 13 MB 8-bank embedded DRAM (a crucial feature). The large multi-banked local memory is used as a software-controlled staging buffer that balances the latency/bandwidth ratio of the memory hierarchy.

Following the success of VIRAM, several vector processors were implemented in FPGAs using the VIRAM ISA. VESPA [7] is design-time configurable soft-core implemented in Stratix-I and -III FPGA. The processor allows ISA subsetting, and some of the primary design parameters that affect both performance and resource usage can be configured. VIPERS [8] is similar to VESPA, but it is less strict to VIRAM ISA compliance, and more tailored to the FPGA target technology. It offers a few new instructions to take advantage of the MAC (multiply-accumulate) and BRAM units in FPGAs.

III. Application-Specific Vector Processors (ASVP)

A. System-Level Organization

The system-level view is presented in Figure 2. Similar to streaming architectures (Stanford Imagine [9], IBM Cell [10]), the execution control is hierarchical: **1. Task scheduling** is dedicated to the Host CPU (e.g. MicroBlaze). Optional inter-core synchronization is handled by the Communications Backplane (δ). **2. Scheduling of the vector instructions** is realized in a simple scalar control processor (sCPU) embedded in each ASVP core. The sCPU forms and issues wide instruction words (α) to the Vector Processing Unit (VPU, Figure 3). **3. Data path multiplexing** and vector processing is realized autonomously in the VPU. The unit handles both the vector-linear and vector-reduction operations, as well as local memory banks access scheduling (β).

The memory hierarchy is exposed on two levels:

1) *Global off-chip shared memory* is accessed through the Streaming DMA engine. In the Xilinx technology the engine uses the MPMC (Multi-Port Memory Controller) and NPI (Native Port Interface). The engine is programmed by the sCPU in each ASVP core (γ), and delivers data into the ASVP local memory banks.
2) *Local storage* in each ASVP is realized using multiple memory banks (BlockRAMs in FPGA). The Vector Processing Unit can access all the banks in parallel (β).

The on-chip local storage is used as the working-set staging buffer. A kernel function running in the ASVP accesses data with non-unit strides, and often the same data is reused multiple times in one computation run (temporal locality). In contrast, off-chip memory (DDR DRAM) has high latency, and it delivers high bandwidth only when unit-stride long data arrays are transferred.

B. Vector Processing Unit (VPU)

Figure 3 shows the (simplified) structure of the Vector Processing Unit (VPU). The VPU fetches data vectors from the local memory banks, processes them, and stores the result back. The memory banks are dual-ported: one port of each bank is connected to VPU, the other to the Streaming DMA engine. Hence it is possible to overlap computations and data transfers in the same bank. In the default configuration each bank is a flat array of 1024 32-bit words, suitable for holding single precision floating-point values.

Vectors are extracted from the memory banks by the *Address Generators* (AG). The full hardware configuration of the VPU uses two AGs for each operand channel. The main AG 0-3 handle basic addressing modes: linear or stridden ($increment \neq 1$) access (the increment can be negative), with lower and upper wrap-around bounds (overflowing the upper bound resets the pointer to the lower bound, and vice-versa). The second set of AG 4-7 is for indexed accesses. These AGs have the same configuration registers as the main AGs, but the data stream (η_{0-3}) they read from the bank memories is passed on to the corresponding main AG. There it is used as indices added to the local addresses being sent down to the memory bank.

The architecture does not contain a traditional vector register file as it is notoriously cumbersome and inefficient to implement it in an FPGA. Even the simplest 3-port (2R-1W) register file requires duplicate (data-redundant) dual-ported BlockRAMs. Instead we employ a multi-banked local store with a crossbar. In the default configuration with 4 memory banks this allows to read/write up to 4 values at a time. Further, the local memory banks are not statically partitioned into architectural vector registers. Applications are free to partition the available memory into vector variables: some take advantage of few very long vectors (e.g. FIR, matrix multiplication), other prefer a lot of shorter vectors to implement complex computation (e.g. the image segmentation).

Several vector operands of an instruction *can* be placed in the same memory bank. The crossbar automatically handles the time-domain access scheduling when requests from the Address Generators cannot be satisfied by the switch matrix in the space domain.

C. Data Flow Unit (DFU)

The *Data Flow Unit* (DFU) performs the actual computations on vector streams. A DFU operation is controlled by three fields from the vector instruction word α (Figure 3, see the top right corner): (1) *operation code*, (2) *vector length*, and (3) *number of repetitions*. The 'number of repetitions' field allows to automatically restart the same operation multiple times to create a 'batch' of operations.

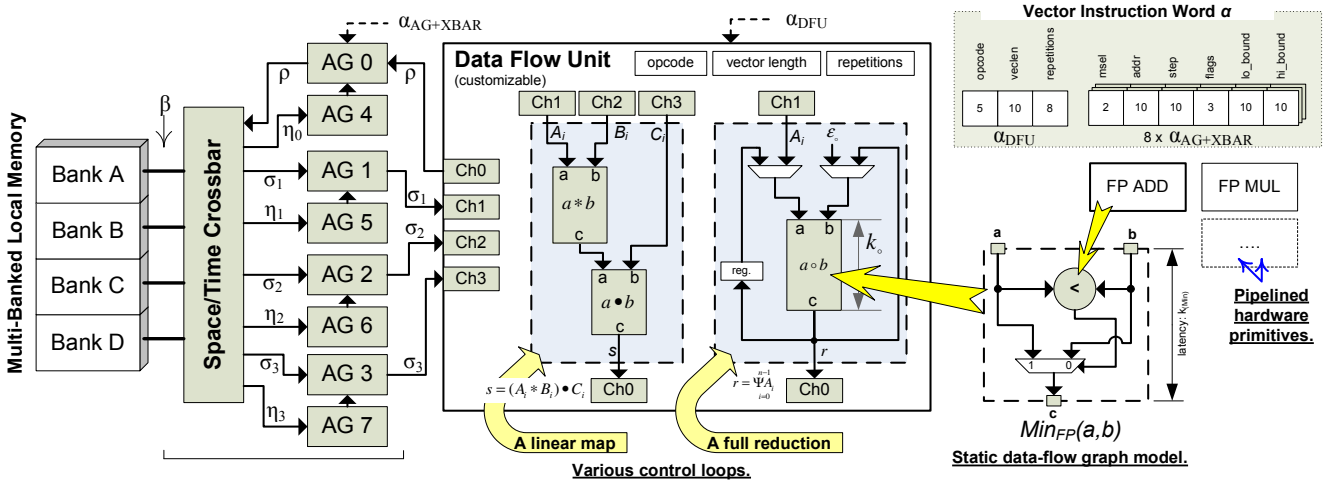

Fig. 3. A functional model of the *Vector Processing Unit* (VPU). Data stored in the local memory banks (A-D) is multiplexed in the *crossbar* and accessed using the *address generators* (AG). It is processed in a *Data Flow Unit* (DFU) that is customized for the application; in the figure one possible internal functional organization of the DFU is shown.

Obviously, this trick lowers the total number of distinct vector instructions that must be issued by the sCPU. More importantly, address generators are *not* rewound in between the operation restarts. Thus by properly setting the AG op-code fields it is possible, for example, to compute one result row in a matrix multiplication using a single DFU instruction.

The Data Flow Unit is the primary target of the application-specific customization effort. First, a set of (vector) operations required by an application algorithm is identified. Based on it an architecture model is constructed, and the application is vectorized.

All application-specific vector instructions are defined by a combination of a *control loop* and an embedded *static data-flow graph*. Three types of control loops are recognized: (a) *element-wise function mapping* (e.g. vector addition), (b) *full reductions* (e.g. vector summation), and (c) *prefix reductions* (e.g. cumulative summation).

Static acyclic data-flow graphs are used to represent a single compute iteration of a customized operation. Graph nodes are implemented in hardware by pipelined compute units to achieve good performance (such as a floating-point adder or multiplier). (For example, depending on the technology, the latency k of the FP-ADDER is between 3 and 6 cycles.)

State machines implementing control loops in a given architecture configuration are adapted to the pipeline latency of the underlying hypothetical compute graphs. This is possible as the pipeline latency k of each data-flow (sub-)graph is known a priori. Pipelining of the **element-wise loops** is trivial; after the first k cycles one result is obtained in each cycle.

Full reduction $\Psi_\circ$ of vector A_i using operator $\circ$ can be defined as:

$$r = A_0 \circ A_1 \circ ... \circ A_{n-1} = \overset{n-1}{\underset{i=0}{\Psi_\circ}} A_i \tag{1}$$

(E.g. summation is: $\Psi_+ \equiv \sum$; $r = \sum_{i=0}^{n-1} A_i$).

The operator denoted $\circ$ represents the embedded data-flow function with latency $k_\circ$, and it has to be associative and commutative so that the reduction can be pipelined. Moreover there has to exist a neutral element $\epsilon_\circ$ (e.g. $\epsilon_+ = 0$). Then we can write (for the case $k_\circ = 2$):

$$r = \epsilon_\circ \circ A_0 \circ A_1 \circ ... \circ A_{n-1} = \tag{2}$$

$$= (\epsilon_\circ \circ A_0 \circ A_2 \circ ...) \circ (\epsilon_\circ \circ A_1 \circ A_3 \circ ...) \tag{3}$$

The last form leads to a hardware sub-circuit model labelled 'full reduction' in Figure 3 in the DFU block. The reduction computation in the circuit goes through three phases: (1) In the first $k_\circ$ cycles the operator output c is invalid, thus the input b must be supplied with the value $\epsilon_\circ$. (2) In the next $n - k_\circ$ cycles the rest of the vector A_i is consumed, and partial results on c are routed to the input b. (3) In the windup phase which lasts w cycles the partial results circling in the pipeline are gradually reduced down to the final value r.

In the windup phase it takes $k_\circ$ cycles (one round) to halve the number of values circling in the pipeline, thus $log_2 k_\circ$ rounds are required: $w = (1 + log_2 k_\circ)k_\circ - 1$. Therefore, the pipelined (parallel) implementation of the full reduction is much faster than the sequential one:

$$t_{seq,\Psi_\circ}(n) = k_\circ \cdot n \tag{4}$$

$$t_{par,\Psi_\circ}(n) = n + w = n + (1 + log_2 k_\circ)k_\circ - 1 \tag{5}$$

The execution times t are given in clock-cycles here. In the pipelined algorithm the windup latency is w cycles, but there is only $k_\circ$ values in the execution pipeline at the beginning. Hence, the operator $\circ$ is used only in $k_\circ - 1$ cycles, and empty cycles are inserted into the pipelined unit otherwise.

Prefix reduction $\Xi_\circ$ of a vector A_i into vector C_j is defined:

$$\overset{n-1}{\underset{j=0}{\forall}} : C_j = \overset{j}{\underset{i=0}{\Psi_\circ}} A_i, \quad \text{hence: } C = \overset{n-1}{\underset{i=0}{\Xi_\circ}} A_i \tag{6}$$

Prefix reductions are much more difficult to implement in a pipelined manner. The pipelined algorithm requires $log_2 n$ passes over the array, and in each pass i roughly $(n - 2^i)$ operations are performed that can be pipelined. Thus:

$$t_{seq,\Xi_\circ}(n) = k_\circ \cdot n \tag{7}$$

$$t_{par,\Xi_\circ}(n) \approx \sum_{i=0}^{log_2 n} (n - 2^i) = n \cdot log_2 n - n + 1 \tag{8}$$

The parallel algorithm is faster for some combinations of $k_\circ$ and n, but it is asymptotically slower (in n) than the sequential algorithm ($O(n) < O(n \cdot log_2 n)$) when only one instance of the reduction operator $\circ$ is available. Thus, when the prefix reduction algorithm is needed, we use the sequential implementation.

TABLE I
SAMPLE OF OPERATIONS IMPLEMENTED IN THE DFU. TYPE: M=ELEMENT-WISE FUNCTION MAP, FR=FULL REDUCTION

Operation	Type	Definition
VCOPY	M	$A_i \leftarrow B_i$
VADD	M	$A_i \leftarrow B_i + C_i$
VMUL	M	$A_i \leftarrow B_i \cdot C_i$
VMAC	M	$A_i \leftarrow B_i \cdot C_i + D_i$
VSUM	FR	$A_0 \leftarrow \Psi_+(B_i)$
DPROD	M+FR	$A_0 \leftarrow \Psi_+(B_i \cdot C_i)$
VMIN	FR	$A_0 \leftarrow \Psi_{\mathrm{Min}}(B_i)$
INDEXMIN	FR	$A_0 \leftarrow Arg\{\Psi_{\mathrm{Min}} B_i\}$
VCMPLT	M	$A_i \leftarrow (B_i < C_i)\ ?\ True : False$
VSELECT	M	$A_i \leftarrow (B_i \neq 0)\ ?\ C_i : D_i$
VCONVR	M	$A_i \leftarrow int2float((B_i >> 16)\&\ 0\text{xFF})$

D. Practical Example

In the *Image Segmentation* (IMGSEG) application there is an operation that locates the minimal (maximal) value in a given vector of floating-point numbers. The operation either returns the value (then it is VMIN, VMAX), or the integer index where the value is located (INDEXMIN, INDEXMAX). The integer index can be used in subsequent operations for indexed accesses in address generators.

These operations can be efficiently implemented as full reductions. First we define a scalar function $\mathrm{Min}(a, b)$ that simply returns the lesser of the two arguments. The function is commutative ($\mathrm{Min}(a, b) = \mathrm{Min}(b, a)$) and associative ($\mathrm{Min}(a, \mathrm{Min}(b, c)) = \mathrm{Min}(\mathrm{Min}(a, b), c)$). The neutral element is $\epsilon_{\mathrm{Min}} = +\infty$ because $\mathrm{Min}(a, +\infty) = a$. Thus we can place $\mathrm{VMIN} \equiv \Psi_{\mathrm{Min}}$. The logical composition is shown in Figure 3.

Another example of application-specific vector instructions are the VCONVR/VCONVG/VCONVB instructions. The instructions take a 32-bit pixel, extract a given 8-bit color (R, G, or B) from it, and convert the colour to a floating-point value. Using a conventional RISC vector ISA (e.g. VIRAM) each operation would be implemented using a sequence of at least 3 instructions (bit mask, shift, float conversion).

Table I lists some other vector instructions we have implemented for the IMGSEG application. The DPROD operation is the dot-product that is very useful for implementing matrix multiplications. The VCMPLT (compare-less-than) operation compares two vectors element-wise and returns a vector of boolean values. The VSELECT operation is a vectorized conditional ternary operator as known in the C language.

Given a set of operations and their high-level specifications as in the table, the hardware implementation of the customized DFU can be generated. Currently this is done mostly manually in VHDL, however, it should be possible to synthesize the DFU automatically in a tool. This is left as a future work.

E. Programming Paradigm

There are two distinct instruction sets in the architecture. The control sCPU executes a classical scalar ISA, and it is programmed in the C language. Currently the architecture uses the 8-bit PicoBlaze processor as the sCPU, and we have developed an optimizing C compiler in the LLVM framework for the PicoBlaze target [11]. The VPU executes vector instructions (denoted α in the figures); they are prepared by sCPU in a so-called 'instruction forming buffer'. The forming buffer is an sCPU I/O periphery, and typically several sCPU instructions are required to setup the next vector instruction for VPU in the buffer. Ideally the preparatory steps are finished before the previous vector instruction has completed so that the sCPU and VPU execution is fully overlapped.

	M2	M3
A3	100	
A4		150
A5		166
A6		

(a) Virtex 5

	M2	M3
A3	100	
A4		150
A5		
A6		200

(b) Virtex 6

	M3	M4
A2	50	
A3		<125
A4		125

(c) Spartan 6

TABLE II
THE MAXIMAL FREQUENCY IN MHZ DEPENDS ON THE PIPELINE DEPTH. Ax = FP-ADDER LATENCY, Mx = FP-MULTIPLIER LATENCY.

IV. EVALUATIONS

A. Technology Scaling

The ASVP architecture was ported to several generations of Xilinx FPGA technology: Virtex 5 (XC5VLX110T-1), Virtex 6 (XC6VLX240T-1), and Spartan 6 (XC6SLX45T-3). The ASVP core is divided in two clock domains: (1) The sCPU and the configuration interfaces are clocked at the base frequency f_0. (2) The Vector Processing Unit is clocked at f_{VPU}. Generally $f_0 \leq f_{VPU}$. As the sCPU is part of a wider ecosystem with which it has to communicate (the γ, δ links in Figure 2), and also because the PicoBlaze processor that implements the sCPU operates in lower frequency ranges, it is clocked at the system base frequency f_0. Contrary, the VPU is coupled only through the command/status link α that carries the vector instruction word to the VPU.

The base frequency f_0 is determined by external factors, such as the System-on-Chip platform and the Host CPU (e.g. MicroBlaze). For simplicity we assume here $f_0(\mathrm{Spartan6}) = 50\mathrm{MHz}$ and $f_0(\mathrm{Virtex5} + 6) = 100\mathrm{MHz}$.

The maximal f_{VPU} is determined by the level of pipelining in data paths and the routing requirements. The separation of the DFU and AG units by FIFO queues allows to pipeline the two VPU parts independently. For example in the high-speed mode additional registers are inserted between the crossbar switch and the memory banks to improve BlockRAM output delays. Ideally, increasing the pipeline latency ($k_\circ$) of a compute unit (such as FP-ADDER) by factor S should also increase the maximal operational frequency by the same factor.

B. FPGA Synthesis Experimental Results

To determine the technology scaling characteristics the ASVP core was instantiated in FPGA in a given configuration, and the maximal target operating clock frequency f_{VPU} was iteratively lowered until the FPGA synthesis process (including place, route, and timing analysis) finally succeeded. Table II summarizes the results for Virtex 5, 6 and Spartan 6 technologies. The ASVP configuration is expressed in $AxMy$ notation: Ax specifies that the FP-Adder has latency x, and similarly My refers to the FP-Multiplier latency y. The maximal operating frequencies of the VPU are 166MHz, 200MHz, and 125MHz, for V5, V6, and S6 technologies, respectively.

Information about area breakdown in flip-flops (FF), combinatorial resources (LUTs), and slices is given in Figure 4 for the Virtex 6 technology; in the other technologies (V5, S6) the results are similar. The single-clock domain (low-speed) and dual-clock domain (high-speed) configurations are compared in the figure.

The control processor itself (sCPU, PicoBlaze 3) consumes very few resources. However, its peripheral circuitry (I/O decoders and

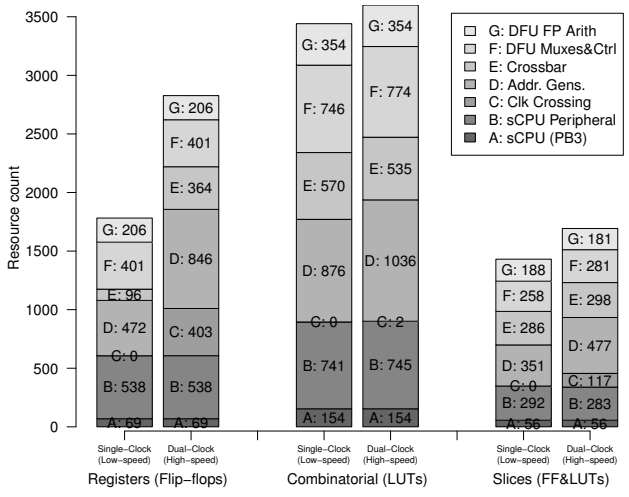

Fig. 4. Breakdown of the area (FF, LUTs and Slices) consumed by the ASVP core in Virtex 6 technology in the single-clock/low-speed and dual-clock/high-speed configurations.

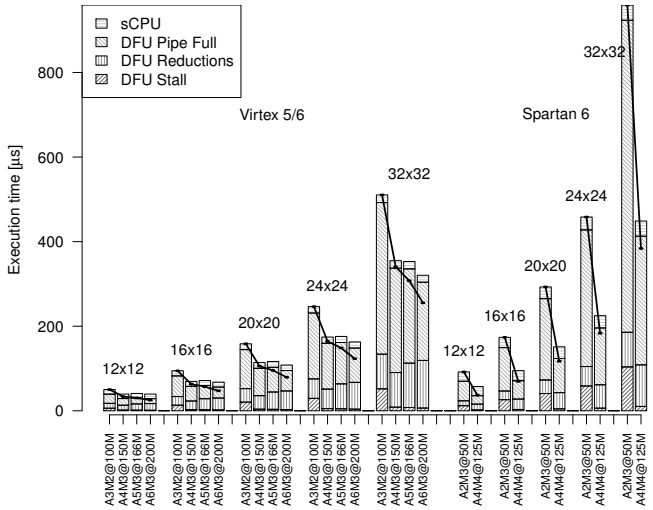

Fig. 5. *Matrix Multiplication* (MATMUL) execution time in microseconds for different matrix sizes, technology (left: Virtex 5/6, right: Spartan 6), and the corresponding frequency/latency ratios.

multiplexers) consumes about 5x more space than the sCPU. The registers for clock domain crossing between f_0 and f_{VPU} constitute about 15% of flip-flops in the dual-clock configuration. The relatively high count is caused by registering the whole vector instruction word α between the sCPU and VPU. The dual-clock configuration is also optimized for a higher frequency, thus additional registers are used in the VPU for deeper pipelining. The crossbar is about 3.8x larger in flip-flops because registers are instantiated on all its inputs. Address generators consume about 1.8x more registers because several more FIFOs are instantiated, and the existing ones are deeper.

All in all, the dual-clock high-speed configuration requires 1.6x more registers, but a roughly equal number of LUTs. In the Virtex 6 technology one slice represents 4 six-input LUTs and 8 storage elements. Thus, after packing the design into slices, the dual-clock high-speed configuration requires roughly 1.2x more slices.

C. Applications Benchmarks

Given the technology performance characteristics determined by the FPGA implementation in Table II, we simulated the cycle-accurate synthesizable VHDL model of the ASVP core in the ModelSim simulator to measure the execution time and dynamic profile of several benchmark kernels. All the benchmark programs compute in the floating-point single precision format. Bar plots in Figures 5, 6, and 7 give the total execution time in microseconds for different technology nodes (latency and frequency) and kernel parameters (e.g. matrix size). The total execution time is split into four parts in the bars: D: execution on sCPU that was not overlapped with computation in DFU; C: DFU has full pipeline (useful computation); B: DFU pipeline bubble due to the reduction windup; A: DFU stall (input data not available).

The bar plots are overlaid with line plots that show the ideal clock frequency scaling for reference: The first column in each group is taken as the baseline, and the subsequent data points are simply scaled by the frequency difference. This ideal scaling is highly optimistic because it assumes that the operating frequency in the whole core is increased, while in fact due to the implementation constraints we keep the sCPU clock frequency f_0 constant and change only f_{VPU}.

In the more complex benchmarks (MANDEL, IMGSEG) the minimal kernel execution time caused by the fixed execution speed (f_0) of the sCPU is presented. The time is measured in simulation by setting f_{VPU} to a very high value (10000MHz), hence fully exposing the sCPU latency.

Matrix Multiplication (MATMUL): The MATMUL program computes a dense matrix multiplication $C = A \times B$. For brevity we present only results measured for square matrices n x n. The computation requires n^2 dot-products that are grouped in n batches. Each batch computes one row (or one column) of the result matrix. Matrix multiplication shows a relatively high reduction cycle count overhead (Figure 5). In the fastest Virtex 6 node A6M3@200M when multiplying 32x32 matrices the reduction cycles contribute to about 35% of the total execution time.

Mandelbrot Set (MANDEL): The MANDEL program computes the Mandelbrot set for a given set of points (a tile). The output of the kernel is an array of the number of iterations executed for each point until the point is known not to be in the set. The tile size is 200 points, and the maximal number of iterations before a forced escape is 50. The kernel speculatively executes all 50 iterations for each point, hence its execution time is independent of the actual part of the set being computed. The body of the iterative loop consists of 18 vector instructions. The kernel (Figure 6) does not contain reduction operations, and given it processes relatively long vectors (tile size 200 elements), it scales quite well because even for higher f_{VPU} the sCPU has enough time to prepare another vector instruction in the forming buffer.

Image Segmentation (IMGSEG): The IMGSEG program implements image foreground/background pixel motion detection (segmentation) using the algorithm presented in [12] without shadow detection and with several modifications. The decision that a pixel from the current frame represents foreground or background depends on statistical models and their mixture. Each pixel is modelled by a mixture of *K* strongest Gaussian models of background (K=4 in our implementation); Gaussian models (mean values and variance) represent a state (colour) of the pixel. Each model also contains the *weight* parameter that specifies how often a particular model successfully described the background in the pixel. The algorithm was vectorized, and the vector length was 50 pixels. Conditional branches were vectorized by speculatively computing both possibilities and

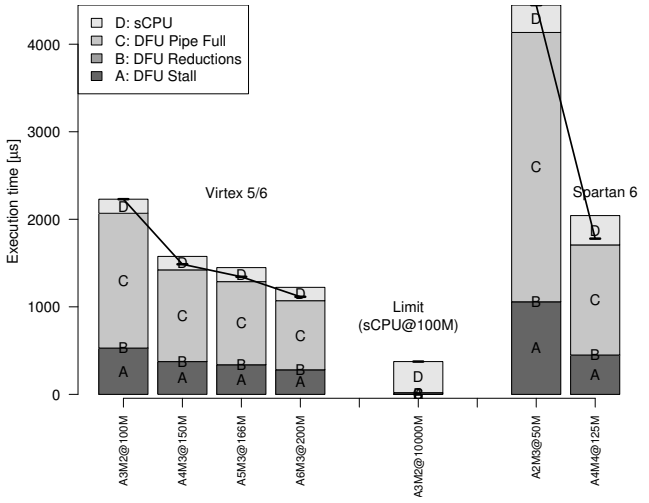

Fig. 6. *Mandelbrot Set* (MANDEL) execution time in microseconds for different technologies (V5/6, S6) and the corresponding frequency/latency ratios. The middle column (labelled 'Limit') shows the speedup limit imposed by the sCPU.

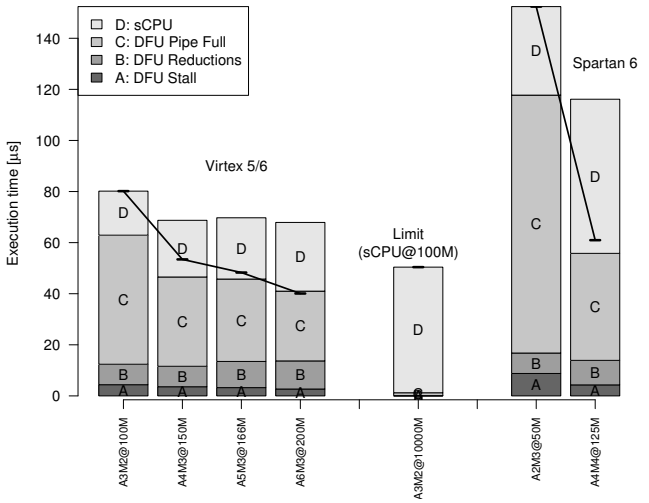

Fig. 7. *Image Segmentation* (IMGSEG) execution time in microseconds for different technologies (V5/6, S6) and the corresponding frequency/latency ratios. The middle column (labelled 'Limit') shows the speedup limit imposed by the sCPU.

then *VSELECT*ing the correct value per each element.

Execution time of the IMGSEG kernel (Figure 7) is largely determined by the sCPU speed (the VPU is idle 52% of time in Spartan 6 A4M4@125M technology). To load a single 16-bit value into the vector-instruction forming buffer (e.g. AG initial addresses are usually changed between two succeeding operations), PicoBlaze sCPU has to execute 4 instructions (2xLOAD, 2xOUTPUT). We plan to amend this by introducing a new configuration table to store the bank memory locations of vector variables. A row from the table could be loaded into the proper fields in the vector instruction forming buffer by a single sCPU I/O write. Our preliminary internal results suggest that this technique can reduce the above mentioned sCPU overhead from 52% to 9% (11x in absolute time), improving the total execution time 1.8x.

V. Conclusion

We presented a new approach to the design of application-specific processors and evaluated it by implementing three algorithms: matrix multiplication, Mandelbrot set, and Image Segmentation. The maximal operating frequency of the sample core in an FPGA is 166 MHz, 200 MHz, and 125 MHz in Virtex 5, Virtex 6 and Spartan 6. When the frequency is downscaled by external factors, the latency of the internal pipelined floating-point units may be decreased, thus lowering overheads in full-reduction operations. The overhead could be lowered further if the DFU interleaved windup execution in one operation with computation in the following one (within an operation batch). The technique is similar to loop unrolling in scalar architectures.

Resource consumption of the proposed architecture is affordable (∼1500 FPGA slices), although there is a room for improvements. Currently, 4 address generators (AG) are instantiated to support indexed addressing modes. Our experience shows, however, that one indexing AG, shared by the existing 4 main AG, would suffice for most applications.

The performance of the embedded simple control processor (sCPU) was found to be the critical factor for the Image Segmentation kernel (52% overhead). However, we are confident this can be overcome by preloading parts of vector instructions from a dedicated table.

Acknowledgement

This work has been supported from project SMECY, project number Artemis JU 100230 and MSMT 7H10001.

References

[1] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in C with ROCCC 2.0," in *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '10. IEEE Computer Society, 2010, pp. 127–134.

[2] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, "Synthesis of Platform Architectures from OpenCL Programs," May 2011, pp. 186–193.

[3] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, july 2009, pp. 35 –42.

[4] M. Danek, J. Kadlec, R. Bartosinski, and L. Kohout, "Increasing the level of abstraction in FPGA-based designs," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, 2008, pp. 5 –10.

[5] A. Papakonstantinou, Y. Liang, J. A. Stratton, K. Gururaj, D. Chen, W.-M. W. Hwu, and J. Cong, "Multilevel Granularity Parallelism Synthesis on FPGAs," in *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '11. IEEE Computer Society, 2011, pp. 178–185.

[6] C. Kozyrakis and D. Patterson, "Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks," in *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 35. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 283–293.

[7] P. Yiannacouras, J. G. Steffan, and J. Rose, "VESPA: portable, scalable, and flexible FPGA-based vector processors," in *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, ser. CASES '08. New York, NY, USA: ACM, 2008, pp. 61–70.

[8] J. Yu, C. Eagleston, C. H.-Y. Chou, M. Perreault, and G. Lemieux, "Vector Processing as a Soft Processor Accelerator," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, pp. 12:1–12:34, June 2009.

[9] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens, "A bandwidth-efficient architecture for media processing," in *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 31. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998, pp. 3–13.

[10] H. P. Hofstee, "Heterogeneous Multi-core Processors: The Cell Broadband Engine," in *Multicore Processors and Systems*, ser. Integrated Circuits and Systems, S. W. Keckler, K. Olukotun, and H. P. Hofstee, Eds. Springer US, 2009, pp. 271–295.

[11] J. Sykora, "Optimizing C Compiler and an ELF-Based Toolchain for the PicoBlaze Processor," 2012. [Online]. Available: http://sp.utia.cz/index.php?ids=pblaze-cc

[12] P. Kaewtrakulpong and R. Bowden, "An Improved Adaptive Background Mixture Model for Realtime Tracking with Shadow Detection," 2001.