Reducing Instruction Issue Overheads in Application-Specific Vector Processors

Jaroslav Sykora, Roman Bartosinski, Lukas Kohout, 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, bartosr, kohoutl, 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. Previously we proposed 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.**

In our setting we employ an embedded simple scalar CPU (8-bit PicoBlaze 3) to control a floating-point vector processing unit (VPU) by issuing wide (horizontally encoded) instructions to it. In this work we dramatically reduce the overhead of the wide-instruction issue (in one case by 13x) by implementing a new two-level configuration table. The table stores frequently used vector definitions (in Level 1) and vector instructions (in Level 2), pre-loading them quickly into the issue buffer. A configuration in the issue buffer can be further modified before being sent to the processing unit. This ensures the architecture stays general and fully customizable.

***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

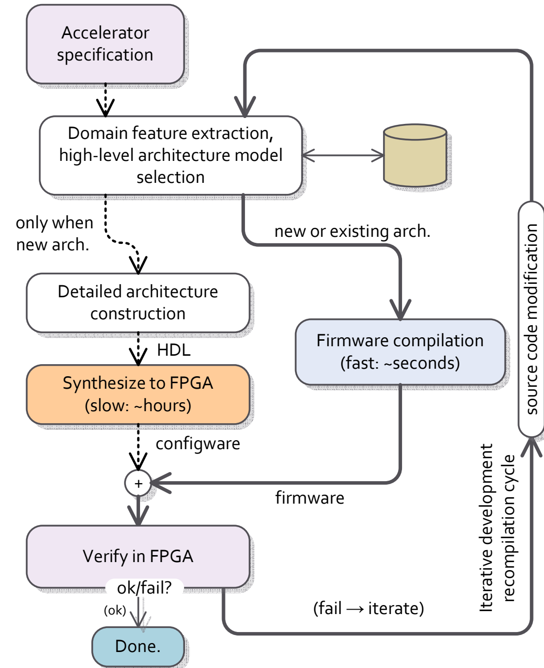


Fig. 1. The proposed custom accelerators development work-flow. Dashed lines indicate processes run once, full lines mark iterative paths.

programmable by firmware to some extent, hence minor source code modifications can be tested quickly for they will trigger only fast 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 our previous work [5] we proposed the ASVP architecture to satisfy 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.

Fig. 3. **Controller in ASVP.** Simple CPU (sCPU = 8-bit PicoBlaze 3) issues vector instructions to the VPU, but with a maximal bandwidth of only 8 bits per 2 clock cycles (‘8b/2cc’). Alternatively, frequent configurations can be preloaded from a new configuration table with a much higher bandwidth of 12 bits per cycle.

In this work we present a new technique that dramatically reduces control overheads in the architecture. In our setting we employ an embedded simple scalar CPU (8-bit PicoBlaze 3) to control a complex vector processing unit (VPU) by issuing wide (horizontally encoded) instructions to it. Although this technique is general, many instructions in the scalar CPU are often required to setup wide vector instructions for VPU. If the setup time in sCPU is not overlapped by a computation in VPU, the efficiency of the architecture is reduced. There are at least two classical solutions to the problem: **A:** increase performance of the simple scalar CPU; or **B:** issue vector instructions directly from the program memory as in traditional vector processors.

Wrt. A: Increasing performance of the scalar CPU would certainly mean a relatively lengthy CPU design or selection process, with an uncertain outcome. Disruptive changes in the architecture would slow down other ongoing development.

Wrt. B: In ASVP vector instructions are very long: in the typical configuration in this paper the length is 203 bits. As it is not feasible to have 203-bit wide program memory, some kind of an encoding or compression technique would have to be developed.

Instead a hybrid scheme was devised: to speed up common cases vector instructions can be loaded from a new configuration table directly into the VPU issue buffer. Two crucial tricks are employed to reduce the hardware size: (1) The new table cannot express all possible configurations, but only those used most often. In that case, the original direct sCPU configuration mechanism is used *in addition* to the new one. (2) To reduce the configuration bit width the new table is organized in two levels. The second level (‘vector instructions’) indirectly references the first level (‘vector definitions’), thus reusing the first-level ‘vector definitions’ in many second-level ‘vector instructions’.

The paper is organized as follows: We begin by related work in section II, and continue with a general description of the ASVP in section III. In section IV the new configuration mechanism is described, and the evaluation is given in section V.

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 [6]. 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 [7] 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 [8] 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 [9] 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 [10], IBM Cell [11]), 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 (Figure 3). The sCPU forms and issues wide instruction words (α) to the Vector Processing Unit (VPU, Figure 4).
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 4 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. Note that the Figure 4 shows a slightly modified architecture with only one shared indexing AG 4 to save resources.

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. 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 4, 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. 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 as a *composition* 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 automata 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 labeled 'full reduction' in Figure 4 in the DFU block.

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 4.

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 colour (R, G, or B) from it, and

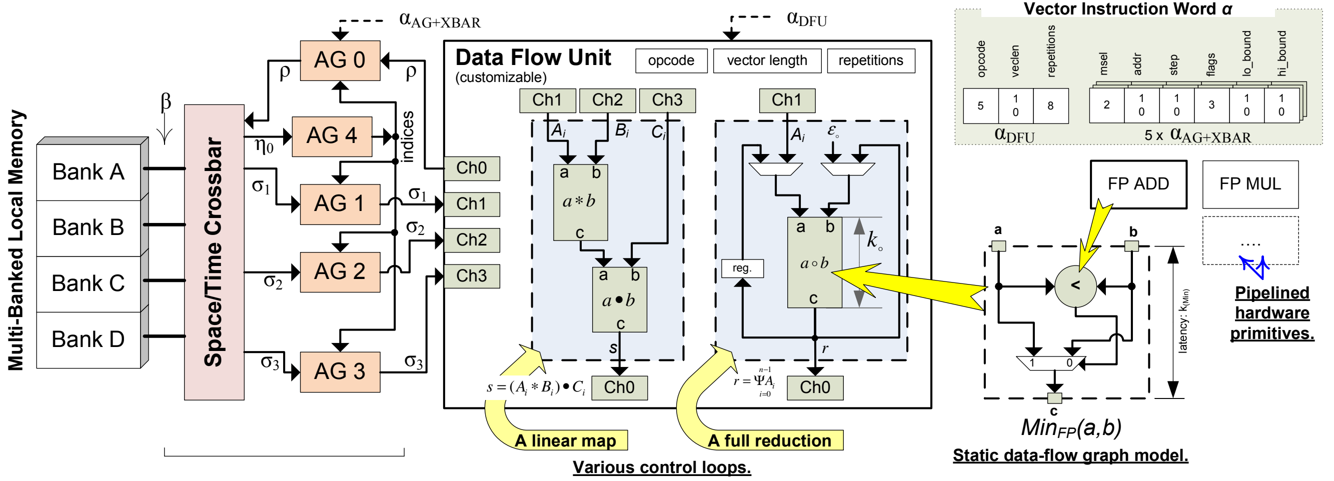


Fig. 4. **A functional model of the *Vector Processing Unit* (VPU).** Data stored in the local memory banks (A-D) is multiplexed in the *crossbar* and accessed using the *address generators* (AG). It is processed in a *Data Flow Unit* (DFU) that is customized for the application; in the figure one possible internal functional organization of the DFU is shown.

TABLE I
A 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	$A_i \leftarrow int2float((B_i >> 16) \& 0\mathrm{xFF})$

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 from 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.

IV. PROGRAMMING INTERFACE

A. Basic Direct-Access API Functions

Figure 3 shows the functional organization of the embedded controller that issues vector instructions to VPU. 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

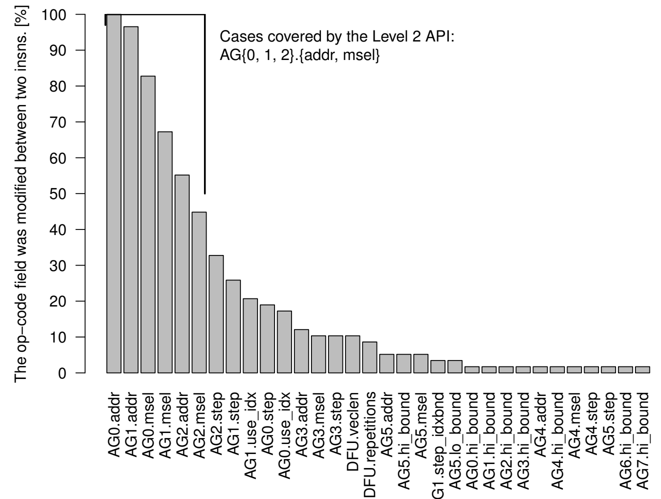


Fig. 5. *Image Segmentation*: Histogram of alterations performed by sCPU to the fields in the vector instruction forming buffer between two succeeding instructions. The *AG0.addr* field is changed in every instruction.

the sCPU, and we have developed an optimizing C compiler in the LLVM framework for the PicoBlaze target [12]. 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.

Table II lists the primary C API functions that are used in the sCPU to access the vector unit. The API functions are divided into three ‘levels’. *Level 0* functions (Figure 7) directly access the vector instruction forming buffer as external registers via the sCPU I/O facilities. For example, the *pb2dfu_set_cnt()* function is defined in C library this way:

```
1 /** Set the (input) vector length. */
2 static inline void pb2dfu_set_cnt(uint16_t cnt)
```

TABLE II
THE PRIMARY C API FUNCTIONS USED TO ACCESS (PROGRAM) THE VECTOR UNIT FROM THE SCPU.

Level	API Function in C in sCPU	Description
0	`void pb2dfu_set_cnt(uint16_t cnt);`	Set the (input) vector length.
0	`void pb2dfu_set_repetitions(uint8_t nrep);`	Set the number of repetitions of a vector operation (batch length).
0	`void pb2dfu_set_addr(uint8_t ag, uint16_t addr);`	Set the base address of the vector operand `ag`.
0	`void pb2dfu_set_bank(uint8_t ag, uint8_t mbank);`	Select the bank (`mbank`) for the specified operand `ag`.
0	`void pb2dfu_set_inc(uint8_t ag, int16_t inc);`	Set the stride of the vector operand `ag`.
0	`void pb2dfu_set_agflags(uint8_t ag, uint8_t flags);`	Set the operation mode (flags) in the address generator (operand).
0	`void pb2dfu_start_op(uint8_t opcode, uint16_t cnt);`	Start the vector operation, with the given vector length.
0	`void pb2dfu_restart_op(uint8_t opcode);`	Start the vector operation.
0	`uint8_t pb2dfu_wait4hw();`	Wait until the vector op. has finished.
1	`void pb2dfu_set_vector(uint8_t ag, uint8_t vdi);`	Load the operand `ag` with the vector defined at the index `vdi` in the Level 1 table (Figure 8)
2	`void pb2dfu_start_insn(uint16_t insn);`	Load AGs 0, 1, and 2 indirectly through the Level 2 and Level 1 tables, and start the operation defined in the Level 2 table (Figure 9).

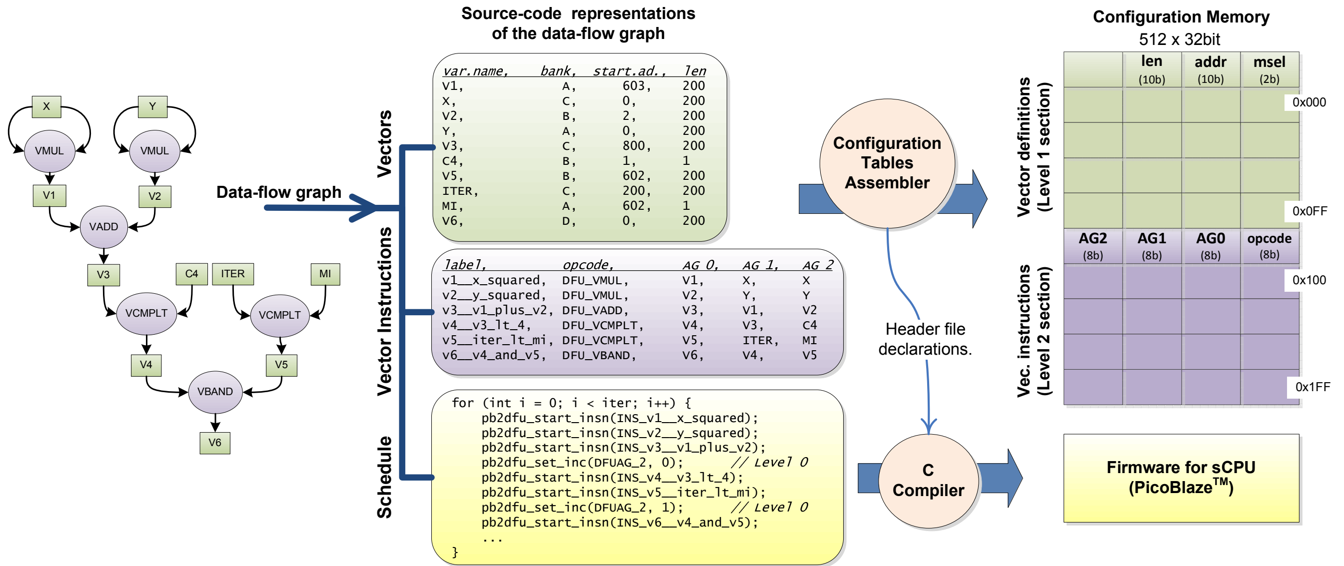


Fig. 6. **Toolchain and the indirect configuration table.** The configuration memory (right side) is 512 x 32 bits with two sections. The first 256-word section contains vector definitions for Level 1 APIs. The second section contains (partial) vector instructions for Level 2 APIs. The contents is specified in a text source code and assembled.

```
{
    /* OUTPUT        value,        (port) */
    __output_port(cnt & 0xff,    REG_DFUVECL_L);
    __output_port(cnt >> 8,      REG_DFUVECL_H);
}
```

Our optimizing C compiler will completely inline the function. For example, to set the vector length of the next vector instruction to 400 elements the program calls *pb2dfu_set_cnt(400);*. This is compiled down to an inlined assembly sequence:

```
load     s4, -112
output   s4, 33        ; 33 = REG_DFUVECL_L
load     s4, 1
output   s4, 34        ; 34 = REG_DFUVECL_H
```

Inlining not only removes the call/return overhead, but (when properly used) it typically also improves the register allocation in the caller function.

The *Level 0* API functions are general as they provide full access to all the hardware functions. However, even with an optimizing compiler, the generated sCPU instruction sequence and the I/O is very elaborate. This was found to be a critical factor for the *Image Segmentation* (IMGSEG) kernel (details below). We implemented the sCPU using the PicoBlaze 3 processor which executes one 8-bit instruction in two cycles. In Spartan 6 the sCPU is running at 50MHz; its performance is only 25MIPS. On the contrary, the VPU's peak performance (in A4M4@125M) is 250MFLOPs, and it can output 125M-elements/s. IMGSEG executes 58 vector operations that collectively take 2790 cycles of the DFU time (counted in the f_0 domain), thus on average there are only 48 cycles of sCPU execution that are fully overlapped between two vector operations. In 48 cycles PicoBlaze 3 executes only 24 instructions. 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), the PicoBlaze processor has to execute 4 instructions (2xLOAD, 2xOUTPUT). Thus, the many instructions needed in sCPU to prepare the next vector instruction cannot be overlapped with useful computation in VPU.

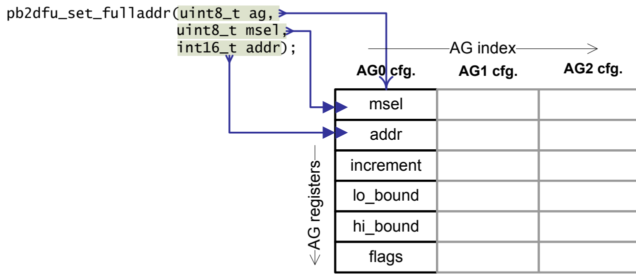


Fig. 7. **Level 0**: Direct access to DFU/AG configuration registers from C API.

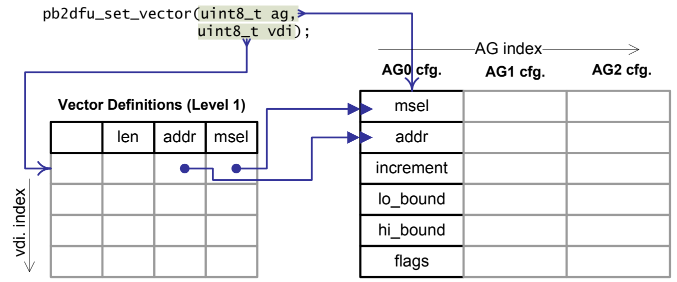


Fig. 8. **Level 1**: Indirect pre-load of a vector definition into an address generator (AG).

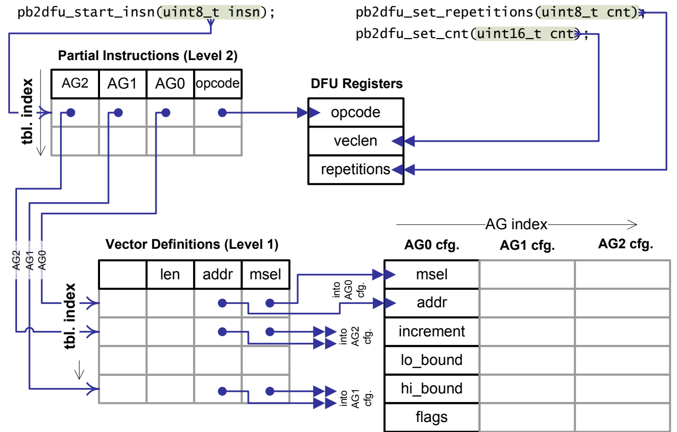


Fig. 9. **Level 2**: Execution of a DFU instruction which preloads AGs 0-2 and the op-code field.

B. New Indirect Configuration Tables

Histogram plot in Figure 5 shows the frequency of alterations of op-code fields between two consecutive vector instructions in the *Image Segmentation* kernel. The ‘AG_n.addr’ and ‘AG_n.msel’ fields are changed very often from one vector instruction to another for they define the starting address (addr) and bank (memory select) of a vector variable in the local memory. Furthermore, address generators (AGs) 0, 1, and 2 are accessed most frequently, while the other configuration parameters are changed only occasionally.

Hence we implemented a new two-level configuration table to speedup the process of setting up the vector instructions in the forming buffer. This new configuration mechanism does not supersede the existing *Level 0* functions as it is *less* general, but rather complements it.

The first level of the new table stores *definitions of vectors* (Figure 8), namely their starting address (10 bits), the memory bank (2 bits), and the vector length (10 bits; currently not used). The second level stores configurations for address generators 0, 1, and 2 as three 8-bit indices into the first-level table, and an 8-bit vector operation op-code (Figure 9). Both levels are implemented as a single 512 x 32-bit local memory (FPGA BlockRAM). The first 256 words are reserved for the Level 1 table (vector definitions), and the second 256 words are for the Level 2 table.

A Level 1 (Figure 8) API function *pb2dfu_set_vector(uint8_t ag, uint8_t vdi)* can be used to load the fields ‘addr’ and ‘msel’ of any address generator *ag* with the vector definitions provided at an index *vdi* (vector definition index) in the table. If other than the two common AG fields have to be loaded (i.e. increment, lo_bound, hi_bound, flags), the Level 0 APIs must be used.

A Level 2 (Figure 9) API function *pb2dfu_start_insn(uint16_t insn)* loads address generators 0, 1, and 2, and then automatically starts a vector instruction. First, the specified row *insn* is fetched from the Level 2 section of the configuration table. It contains three *vdi* indices for AGs 0, 1, and 2. These are sequentially fetched from the Level 1 section and loaded into the corresponding AGs. Finally a vector operation specified in the *insn* row is issued to the hardware. Although the configuration table is indexed sequentially, the whole process is accomplished much faster that by using solely the Level 0 API functions in sCPU.

Figure 6 shows the toolchain flow. At the left an algorithm is expressed as a hypothetical static data-flow graph. Boxes represent vector variables, and ovals represent vector operations. The graph is implemented in a textual source code as shown in the middle of the picture. The source code comprises (a) vector definitions, (b) vec. instructions definitions, and (c) sCPU firmware. The (a) and (b) source is processed by a special assembler that generates the contents of the configuration memory. The firmware source is compiled by the C toolchain. Features of the original data-flow graph that cannot be encoded in the configuration memory are implemented in the C source as direct Level 0 API calls. For example, the fact that variables ‘C4’ and ‘MI’ are scalar (1 element) is implemented by setting the increment to 0 in AG2 using *pb2dfu_set_inc()* call. Should an algorithm frequently change increments in AGs, the field could be implemented in the Level 1 table to speed up the configuration process.

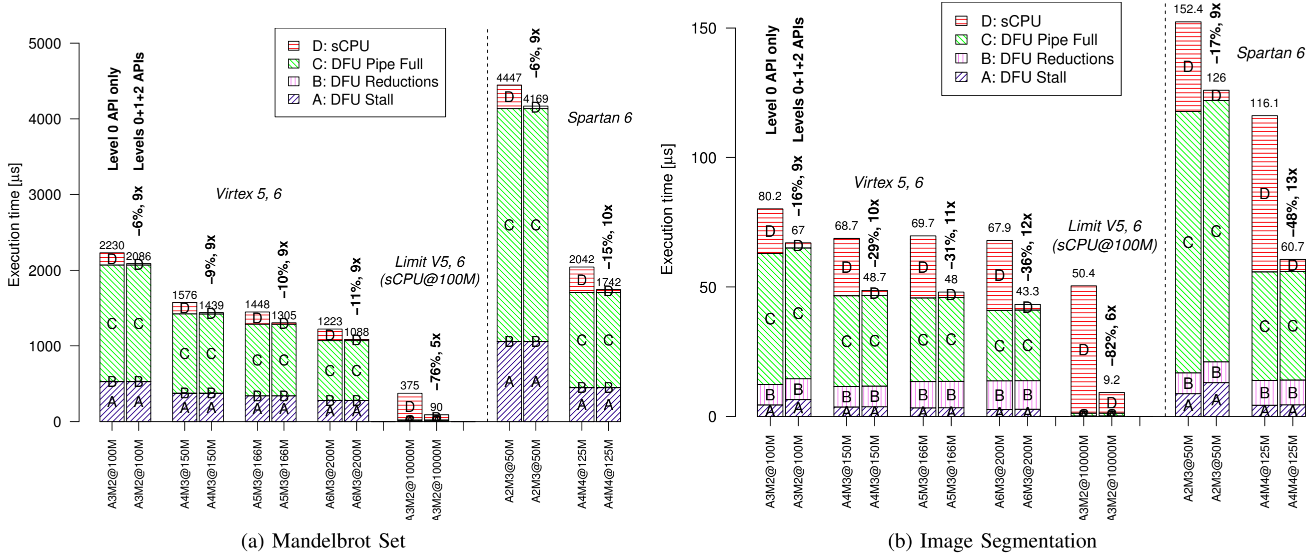
V. Evaluations

A. Technology Scaling

The ASVP architecture was ported to several generations of the 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. On the contrary, the VPU is coupled only through the command/status link α that transfers 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 $f_0(\text{Spartan6}) = 50\text{MHz}$ and $f_0(\text{Virtex5}+6) = 100\text{MHz}$.

The maximal f_{VPU} is determined by the level of pipelining in the data paths and the routing requirements. The separation of the



(a) Mandelbrot Set (b) Image Segmentation

Fig. 10. Execution times in microseconds for the (a) *Mandelbrot Set* and (b) *Image Segmentation* kernels, in Virtex 5, 6, and Spartan 6 technologies and the corresponding frequency/latency ratios. The first bar in each double-bar column represents the situation when only the direct *Level 0 API* is used, the second bar when the indirect configuration tables are used. Bold numbers atop these columns display the absolute speed up in percents and the reduction of the sCPU overhead (in x-notation). The middle double-column labelled 'Limit' shows a hypothetical minimal execution time imposed by the sCPU.

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

Techno.	Kernel	Level 0 API only		Levels 0, 1, 2		**Speed-up**	
		sCPU	Total	sCPU	Total	sCPU	Total
Virtex 5, 6: sCPU at 100MHz							
A5M3 @166M	Mandel	$160\mu s$	$1448\mu s$	$18\mu s$	$1305\mu s$	8.8x	1.1x
	ImgSeg	$24\mu s$	$69.7\mu s$	$2.2\mu s$	$48\mu s$	11x	1.5x
A6M3 @200M	Mandel	$153\mu s$	$1223\mu s$	$18\mu s$	$1088\mu s$	8.5x	1.1x
	ImgSeg	$26.9\mu s$	$67.9\mu s$	$2.2\mu s$	$43.3\mu s$	12x	1.6x
Spartan 6: sCPU at 50MHz							
A4M4 @125M	Mandel	$335\mu s$	$2042\mu s$	$35.1\mu s$	$1742\mu s$	9.5x	1.2x
	ImgSeg	$60.3\mu s$	$116.1\mu s$	$4.6\mu s$	$60.7\mu s$	13x	1.9x

TABLE IV
SPEED-UPS IN THE FASTEST CONFIGURATIONS (SUMMARY).

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 operating 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 operating clock frequency f_{VPU} was iteratively lowered until the FPGA synthesis process (including place, route, and timing analysis) finally succeeded. Table III 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.

Area: In the fastest Virtex 6 A6M3@200MHz node the default configuration of the ASVP without the configuration tables for the Level 1, 2 APIs the core consumes 1525 slices. The configuration tables increase the resource usage by 8% to 1638 slices.

C. Applications Benchmarks

Given the technology performance characteristics determined by the FPGA implementation in Table III, 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. The benchmark programs compute in the floating-point single precision format. Bar plots in Figure 10 give the total execution time in microseconds for different technology nodes (latency and frequency). The total execution time is split into four segments 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).

In each double-bar column the first bars represent a core *without* the proposed indirect configuration tables, i.e. when only Level 0 API is used in kernels. In the second bars all the API levels are used.

The minimal kernel execution time in Virtex 5, 6 caused by the fixed execution speed (f_0) of the sCPU is presented in the middle double-column labeled 'Limit'. The time is measured in simulation by setting f_{VPU} to a very high value (10000MHz), hence fully exposing the sCPU latency.

Mandelbrot Set (MANDEL): The MANDEL program computes the Mandelbrot set for a given set of points (a tile) (Figure 10a).

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 (this is advantageous for benchmarking, albeit suboptimal for real-world use). The body of the iterative loop consists of 18 vector instructions. The kernel does not contain reduction operations (hence segment 'B' is zero), 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. When the Level 1,2 APIs are employed they improve the total execution time by roughly 10% (not counting the hypothetical 'Limit' case).

Image Segmentation (IMGSEG): The IMGSEG program implements image foreground/background pixel motion detection (segmentation) using the algorithm presented in [13] 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 modeled 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 then *VSELECT*ing the correct value per each element.

When only the original Level 0 API is used the execution time of the kernel (Figure 10b, Table IV) is largely determined by the sCPU speed. For example, in Spartan 6 A4M4@125M technology the VPU is idle 52% of time. The proposed indirect configuration tables in Level 1, 2 APIs improve the absolute execution times tremendously: by 30% on average, and almost by 2x (48%) in Spartan 6 A4M4@125M technology. In the later case the sCPU overhead itself is reduced by 13x from 52% of the absolute execution time ($60.34\mu s$ of $116.1\mu s$) down to 8% ($4.58\mu s$ of $60.7\mu s$).

Another positive effect of the new APIs is the sCPU code size reduction. The Level 0 APIs-only kernel had to be written manually in an assembler as the compiler-generated code was too large for a 1024-word program memory of the PicoBlaze 3 processor. Even then only 58 words of the program memory remained free. On the contrary, the new version that uses Level 1, 2 APIs is written entirely in C and still 236 words of the program memory remain free.

VI. Conclusion

The ASVP compute core is composed of a simple scalar CPU and a customizable floating-point vector processing unit (VPU). The scalar CPU prepares and issues wide (horizontally encoded) vector instructions to the VPU. Previous experience showed that in complex compute kernels the scalar CPU is often the bottleneck that limits VPU instruction throughput.

In this paper we noticed that typically only a small subset of the vector-instruction fields are modified between two operations. (This does not mean, however, that the rest of the fields are superfluous.) Hence, to speed up the common case, a new memory (BlockRAM) table was introduced that stores the constant values for the critical (most frequently changed) fields of vector instructions. A simple automaton is commanded by the scalar CPU to fetch a selected row from the table and patch it in the instruction buffer.

The new configuration table is logically organized in two sections, corresponding to two indirection levels. In the first section vector extents in local memories are defined as {bank, start, length} triples. In the second section common three-operand vector instructions are defined; the operands (= address generator parameters) are encoded as indices into the Level 1 section. Hardware functions (configurations) that cannot be invoked via the new mechanism are accessed by direct writes from the scalar CPU. The presented technique can be transferred to similar use cases that employ an embedded CPU that controls complex peripherals.

The technique was implemented and evaluated in FPGAs (Virtex 5, 6, Spartan 6) using the *Mandelbrot Set* and *Image Segmentation* kernels running in the ASVP core. Resource cost of the new hardware function is around 100 slices in Virtex 6. The scalar CPU overhead, i.e. the time *not* overlapped by a useful computation in VPU, is reduced typically 9x (max. 13x). This leads to absolute time reduction (speed-up) of 10% (min 6%, max 15%) for the *Mandelbrot Set* kernel, and 30% (min 16%, max 48%) for the *Image Segmentation* kernel.

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] J. Sykora, L. Kohout, R. Bartosinski, L. Kafka, M. Danek, and P. Honzik, "The Architecture and the Technology Characterization of an FPGA-based Customizable Application-Specific Vector Processor," 2012.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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

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