Energy-Efficient Computation of L1 and L2 Norms on a FPGA SIMD Accelerator, with Applications to Visual Search

Calin Bira*, Radu Hobincu*, Lucian Petrica*, Valeriu Codreanu†, and Sorin Cotofana‡
*Politehnica University of Bucharest
{calin.bira, radu.hobincu, lucian.petrica}@arh.pub.ro
†Eindhoven University of Technology
v.codreanu@tue.nl
‡Delft University of Technology
s.d.cotofana@ewi.tudelft.nl

Abstract—This paper presents a novel accelerator architecture which is SIMD in nature and fully programmable. It provides support in an energy effective manner to a wide range of vector computations, including scalar products and similarity metrics like sum of absolute differences and sum of squared differences. We have evaluated an implementation of the proposed architecture on the Xilinx Zynq-7000 EPP featuring the ARM Cortex-A9 processor, running a SIFT descriptor matching benchmark. Our results indicate that the processor can offload the most intensive computational kernels of the benchmark to the accelerator, thus delivering 4-6x better matching throughput than the ARM processor alone. Moreover, the execution of the SIFT matching benchmark on the accelerated platform consumes 3x less energy than on the ARM Cortex-A9, at a similar power consumption. Our results also suggest that the accelerated ARM system is 40% more energy effective than Intel Core i7 2600K and Nvidia GTX680 when executing the SIFT matching benchmark.

I. Introduction

Object recognition and classification are currently some of the hot topics in computer vision, with applications in image matching [11], robotics [16], and panorama stitching [4]. When matching large databases against each-other, matching speed is the most important performance metric, but power and energy efficiency plays a major role in the economy of the entire process. For robotics and mobile devices in general, energy efficiency is the most important metric since it relates directly to battery drain. Previous work has yet to demonstrate a solution to the image matching problem which is high-speed, low-power, and low-energy. Our research aims to prove that these goals are attainable without sacrificing programmability.

In this paper we propose an energy-efficient solution to the matching problem based on a Single Instruction Multiple Data (SIMD) accelerator architecture. The proposed architecture is well suited for execution of multiply-accumulate operations and for selective execution on large data vectors. It consists of an array of efficient processing elements which are fed instructions and data by the host processor, through the use of Direct Memory Access (DMA) and several FIFO interfaces. The proposed architecture was implemented and evaluated on the Xilinx Zynq-7000 EPP [15] running a SIFT descriptor matching benchmark on a standard image dataset. Our results indicate that the ARM host processor included within the Zynq-7000 can efficiently offload computationally expensive kernels to the accelerator, resulting in 4-6x better matching throughput than when executing alone. Also, the SIFT matching benchmark execution consumes 3x less energy on the proposed platform than on the ARM Cortex-A9 alone, at similar power consumption. Comparisons with desktop parts suggest that the accelerated ARM system is 40% more energy effective than a high-end desktop CPU-GPU system.

This paper is organized as follows. Section II presents details on image matching metrics and the computational requirements involved. Section III introduces the proposed architecture and programming model. In Section IV we present the implementation of the proposed architecture on an off-the-shelf programmable chip. In Section V we introduce the experimental results, with Section VI presenting some concluding remarks.

II. Image Matching

The object recognition process works in several steps. First, the images are split in two sets: the *query* and *search* images. The *search* images are the ones in which the objects are to be detected, while the *query* image contains the objects that we wish to find in the *search* set. The object recognition system does not work on the images themselves, but rather on a set of local features representing interesting characteristics of objects present in the image [17][13]. Therefore, the next step is to extract these local features, called *keypoints*, using algorithms such as Scale Invariant Feature Transform (SIFT) [12]. The third and final step is to find matching keypoints, which are identical or very similar in both the *query* image and at least one *search* image.

The keypoint matching task relies on finding the nearest neighbour of a given query keypoint, in the database of search keypoints, according to a certain distance metric. Formally, the nearest neighbour search problem is finding the element $NN(X)$ in a finite set Q included in a D-dimensional space

ISBN: 978-1-61804-237-8

minimizing the distance to the input vector X. This is described in Equation (1), where $argmin\ d(X,Y)$ is the tuple (X,Y) which minimizes function d.

$$NN(X) = \underset{Y \in Q}{\operatorname{argmin}}\ d(X,Y) \quad ; X \in \mathbb{R}^D, Y \in Q \subset \mathbb{R}^D \quad (1)$$

The distance metric $d(X,Y)$ is the L_p norm computed as in Equation (2). The most usual norma are SAD, i.e, L_1 and SSD, i.e, L_2. For a given matching task with Q keypoints in the query set and S keypoints in the search set, we need a total of $Q * S$ SSD or SAD operations to find the nearest neighbour from the search set for all query keypoints. A match is declared between a query keypoint and its nearest neighbour from the search set if the distance between them is below a given application dependent threshold.

$$L_p = (\sum_{i=1}^{D} |X_i - Y_i|^p)^{\frac{1}{p}} \quad (2)$$

For SIFT and similar algorithms, keypoints are described by large, typically 64 to 128 elements, vectors of parameters. The large size of these vectors increases the computational effort and memory required for the nearest-neighbor calculation. The SSD metric is more precise but makes intensive use of multiplication, which is either slow or requires more hardware resources, while the SAD metric can be implemented with only an adder, but requires a conditional execution based on which of the operands is larger. Both metrics require the accumulation of entire data vectors, which can be implemented on a scalar processor in a time proportional to the vector size, therefore slow for SIFT descriptors.

Research projects, e.g, demoASIFT [14], perform keypoint matching on general-purpose CPUs and support OpenMP [6] and vectorization, where available, to increase performance. There have been several proposals for nearest neighbour search using specialized hardware, such as GPUs. Garcia *et al.* and other research groups have implemented GPU nearest-neighbour search and have achieved as much as 120x speedup when comparing to a similar algorithm running on the CPU [10][5]. However, GPU implementations are extremely power-hungry, even though the high attainable matching speeds make for good energy efficiency.

With regard to specialized accelerators for similarity matching, Wong *et al.* demonstrated FPGA-based SAD implementations [20]. Their analysis indicates that an adder tree is the most efficient implementation, but while the achieved throughput is impressive, the accelerator is not programmable, and therefore is only useful for SAD computation. Flatt *et al.* proposed and evaluated a host-coprocessor scheme based on a RISC CPU coupled to an application-specific FPGA circuit and demonstrated 70x speedup when compared to RISC-only execution for the SSD metric [9]. Their approach also makes use of a fully pipelined adder tree architecture, and while configurable at block level, it is still not programmable.

The architectural solution proposed in the following section aims to increase the computational performance of L_1/L_2 matching algorithms on embedded systems, hence with minimal energy consumption, and also without sacrificing programmability.

III. Accelerator Architecture

This section presents the proposed system from a hardware and programming model perspective. The proposed SIMD architecture as well as its FPGA implementation are introduced below. The software section describes the programming environment and the corresponding software stack.

A. Hardware Architecture

The proposed system follows the host-accelerator paradigm . The accelerator is connected to the same bus as the CPU, the memory, and the Direct Memory Access (DMA) engine, and it is mapped to the processor address space. The processor can transfer data to the accelerator from its internal registers and memory, or it can instruct the DMA engine to transfer data directly from the main memory to the accelerator.

The SIMD accelerator, presented in Figure 1, follows the basic principles of the Connex multimedia processor [18]. The computation core consists of N Processing Elements (PEs), which are simple processors containing several internal registers, an Arithmetic Logical Unit (ALU), and instruction decoding logic. The instruction set follows the RISC principle, whereby all instructions execute in one clock cycle, operate on two registers and write the result to a third register. In order to speed up the computation, each PE benefits from a Local Storage (LS) similar to the shared memory employed by modern GPUs. The transfer between this shared memory and the system bus is done through an Input/Output (IO) network. The PEs are all fed the same instruction at any given time, which minimizes the control path overhead present in traditional architectures. Also, specific PEs can be marked as inactive based on arithmetic flags such as Carry, Less, and Equal. This permits the SIMD engine to selectively execute instructions, which is useful for computing certain types of matching metrics, e.g., SAD. Apart from the number-crunching PEs, the SIMD accelerator is composed out of three separate networks:

- The Input-Output network for controlling I/O data transfers between the main memory and the Local Storage.
- The Distribution network for dispatching the to be executed instructions to the PEs, in the form of a fully pipelined logarithmic tree.
- The Reduction network for gathering the result of global operations like sum reduction, in the form of a fully pipelined adder tree.

Mechanisms, e.g., instruction fetching, branching, and scalar computation, implemented in hardware in the Connex processor, have been moved into software, in order to minimize area and power overhead associated with control functions. A simple loop sequencer is retained to execute loops with a constant number of iterations.

The structure of the SIMD accelerator closely matches the requirements of nearest-neighbour search with the L_1 and L_2

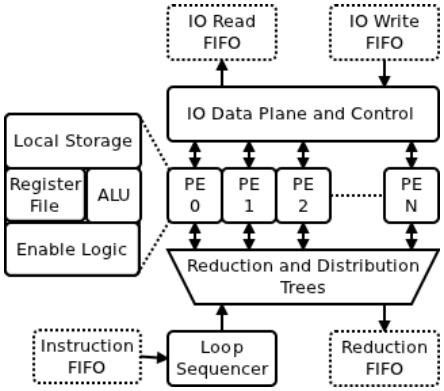

Fig. 1. SIMD Engine Architecture.

norms, which relies on two stages: an element-by-element operation, followed by a reduction operation. The first stage involves performing the same instruction on multiple chunks of data, and hence an SIMD architecture can efficiently do the computation on this level. To speedup the SSD evaluation, each PE includes a hardware multiplier, while the conditional execution feature is targeted at efficient SAD computation. Multiplication is done in two separate instructions: the first launches the multiplication while the second moves the result into the destination register, thus keeping with the RISC principle.

The second computation stage involves the summation of all results computed during the first stage. In our architecture this is implemented with a fully pipelined adder tree, which was found in [20] to be the most efficient way to implement sum reduction for SAD, and was also utilized for SSD in [9].

B. Programming Model

The software architecture and programming model were developed for easy integration with the user code. Instruction fetch as well as most control instructions handled by the host processor, using our accelerator-specific library called OPINCAA (Opcode Injection and Control for Accelerator Architectures).

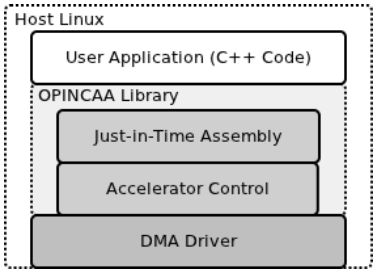

Fig. 2. Accelerator Software Stack.

The overall software architecture is depicted in Figure 2. Accelerator code, on the highest level of the hierarchy, is written in C++ in a specific syntax and can be mixed with ordinary C++ code in order to implement loops and branches. For this purpose, OPINCAA provides a vector data type and operators specific to it, including reduction, cell selection and arithmetic operations. The vectors used in OPINCAA always have the same size as the underlying accelerator implementation.

Kernels of accelerator code are compiled on-demand in a similar fashion to just-in-time (JIT) compilation for Java code. OPINCAA provides the JIT infrastructure as well as other accelerator control functions. Data-dependent accelerator loops are unrolled in software, while loops without data dependencies, and constant number of iterations, are preserved and executed in hardware by the loop sequencer. Compiled kernels are indexed and stored until the user explicitly requests their execution with a call to *executeKernel(index)*, which is a function of the accelerator control driver. In OPINCAA, instruction dispatch is a write to a device file, as are IO writes. Similarly, reductions and IO data are read through file interfaces. The control driver also exposes the functions *readReduction()*, *ioRead()*, and *ioWrite()*, which ensure the correct usage of the file interfaces. These functions abstract the operation of the underlying DMA driver from the user. The DMA driver is implementation-specific, as is the accelerator hardware.

Algorithm 1 SIMD code example.

$$
\begin{array}{l}
for(int\ i = 0;\ i < 10;\ i++)\{ \\
\quad R[1] = LS[i]; \\
\quad R[2] = LS[i + 10]; \\
\quad R[3] = LS[R[1]]; \\
\quad R[0] = (R[1] - R[2]); \\
\quad WHERE\_EQUAL(\\
\quad\quad R[3] = R[1] + R[2]; \\
\quad) \\
\quad REPEAT(5) \\
\quad\quad R[3] = R[3] + R[2]; \\
\quad\quad REDUCE(R[3]); \\
\quad END\_REPEAT \\
\}
\end{array}
$$

Algorithm 1 represents a code snippet to illustrate the syntax used to program the accelerator. The first two lines load the contents of the Local Storage at address i into $R[1]$ and address $i + 10$ into $R[2]$. The third line loads the value at an address stored in a register. The following line computes the difference between $R[1]$ and $R[2]$, places the result in $R[0]$ and sets the appropriate flags. The Carry flag indicates if the sum of $R[1]$ and $R[2]$ overflows, Less indicates if $R[1]$ is less than $R[2]$ and Equal indicates if $R[1]$ and $R[2]$ are equal. The $WHERE$ construct deselects the processing elements where the indicated flag is not set. Execution continues only in selected PEs until an implicit instruction is encountered to re-enable all PEs. In the case of Algorithm 1, the PEs where $R[1]$ and $R[2]$ are equal load their sum into $R[3]$, while all others keep the initial value loaded from the Local Storage. Finally, several summation and reduction operations are launched on register $R[3]$ through the use of the $REPEAT$ statement,

which signals a loop executed in hardware by the loop sequencer. The outer loop, in plain C++ syntax, is unrolled during the JIT assembly, before the program is streamed to the accelerator. Also, any non-vector variable which is used in vector code is replaced by its value during the JIT assembly, and treated as a constant thereafter.

IV. Implementation

This section describes an implementation of the proposed architecture on the Xilinx Zynq-7000 extensible processing platform [15]. The platform consists of two parts, namely the Processing System (PS) and the Programmable Logic (PL). The PS includes a dual-core ARM Cortex-A9 processor running at 667Mhz with NEON instruction support containing also several I/O peripherals. The PS includes a DDR SDRAM memory controller and can boot independently of the programmable logic. The PL consists of a full-fledged Artix-7 FPGA fabric. The PS is linked to the PL through an AMBA AXI bus, and hence the FPGA fabric can accommodate digital circuits that accelerate the computation performed by the ARM cores. For high-speed data transfer between the PS and PL, the Zynq architecture includes a Direct Memory Access (DMA) engine which can be programmed by the ARM processor.

An instance of the architecture was implemented on the Zynq, with the following parameter values: 128 Processing Elements, 16-bit operands, 32 registers, and 2KB Local Storage. This system instance came out of the need to perform fast and efficient nearest-neighbour computation in high-dimensional spaces for SIFT primarily, since it is considered the best-performing matching algorithm. SIMD sizes of 128 processing elements effectively permit calculating the distance between all 128 elements of two SIFT descriptors in one pass. The 16-bit operand dimension was chosen because it has already been used in [14]. Moreover the work in [19] indicates that using a shorter integer representation (short int in our case) instead of full integers or float operands does not result in significant loss of matching accuracy.

The size of the Local Storage was chosen as the size of a Block RAM resource of the Zynq FPGA, while the number of registers allows for an efficient register file in Distributed RAM. Hardware multipliers were implemented in DSP48E1 slices. The resulting accelerator design occupies 90% of the Zynq FPGA and can be clocked at 125Mhz. In our experiment, a 100Mhz frequency was utilized in order to match the AXI bus frequency.

Figure 3 illustrates how the Zynq-7000 is utilized within our approach. The PS connects to the PL through a Xillybus interface core [1], which makes use of the DMA engine to transfer data from the main memory to several FPGA FIFOs. The SIMD accelerator connects to these FIFOs, consuming and producing data from and to the FIFOs as instructed by the PS. On the Zynq, OPINCAA makes use of the Xillybus DMA driver, which abstracts transfers over AXI and exposes the required file interfaces.

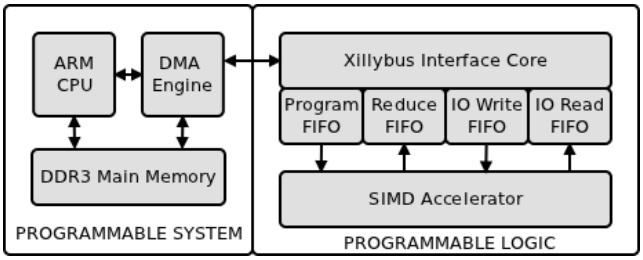

Fig. 3. Zynq-7000 Accelerator.

V. Experimental Results

In order to compare our approach with other matching hardware, we integrated our keypoint matching functions within the demoASIFT project [14]. The evaluation results are obtained from a Zedboard [2] development board housing the Xilinx Zynq-7000 EPP, as presented in Section IV. The benchmark was compiled with gcc version 4.6, which is provided with the Xillinux operating system.

To accurately benchmark our proposal, we made us of images from the standard dataset proposed by Mikolajczyk *et al.* to evaluate feature extractors [13]. The feature extraction was done on the Zynq PS, while the matching was executed either on the PS or on the SIMD accelerator configured on the Zynq PL. We measure the execution time, as well as the energy consumption required to perform image matching for two systems:(i) the baseline ARM system and (ii) the SIMD accelerated system where computation is split between the ARM and the accelerator. Energy consumption for Intel Core i7 2600K quad-core desktop CPU, NVidia GTX680 GPU, and NVidia 8800 Ultra GPU are also presented for the purpose of comparison.

The benchmark code in [14] allows vectorization with Intel SSE [8] by default. We have modified the benchmark in order to make automatic vectorization possible on ARM with NEON [7] instructions, thereby extracting the maximum speed out of the Zynq PS. OpenMP was also used to split the matching workload on all the available processor cores.

A. SIMD Matching Algorithms

By using the previously described SIMD accelerator, we can code the SIFT matching application using Algorithm 2 for SSD or Algorithm 3 for SAD. As discussed in Section II, we need to do $Q*S$ SSD/SAD operations to match a query set of Q keypoints to a search set of S keypoints. For efficient use of the LS and registers, the search and query keypoint sets are broken up into tiles of 308 and 364 keypoints, respectively, in order for one query tile and two search tiles to fit inside the LS simultaneously. One tile from the query set is loaded into the LS and tiles from the search set are sequentially loaded and matched against it. Processing is done on sub-tiles of 28 keypoints from the search set, which allows the register file to be fully used. While processing occurs on one search tile, a second tile is loaded into the LS, thus masking most of the IO time behind the computation.

ISBN: 978-1-61804-237-8

Algorithm 2 SSD Matching Kernel.

```
Require: keypoints to be transferred to LS
BEGIN_KERNEL();
    R29 = 1;
    for(int i = 0; i < 11; i++){
        for(int j = 0; j < 28; j++){
            R[j] = LS[364+ls_off*(28*11)+i*28+j];
        }
        R30 = 0;
        REPEAT(364)
            R[28] = LS[R30];
            R30 = R30 + R29;
            for(int j = 0; j < 28; j++){
                R31 = R[28] − R[j];
                R31 = R31 * R31;
                REDUCE(R31);
            }
        END_REPEAT
    }
END_KERNEL();
```

The actual SSD computation is contained within the innermost loop of Algorithm 2. A search keypoint is loaded from the LS, is subtracted from the currently selected query keypoint, and the difference is squared before being launched in the reduction network, which does the summation. Algorithm 3 presents the innermost loop of the SAD accelerator kernel, which uses conditional execution. In this case R29 is zero and is used to test whether the result of the difference needs to be negated. This code hides some of the instructions actually in use, which re-enable the PEs after the conditional execution. The extra instructions make SAD computation slower than SSD on our accelerator, despite the fact that multiplication takes two cycles to complete.

Algorithm 3 SAD Computation.

```
for(int j = 0; j < 28; j++){
    R30 = R[28] − R[j];
    R31 = R30 < R29;
    WHERE_LT(
        R30 = R[j] − R[28];
    )
    REDUCE(R31);
}
```

B. Performance

Table I presents the performance results in terms of millions of keypoint matches per second (MM/s), where a keypoint match is an SAD or SSD operation on a pair of SIFT keypoints. The SIMD accelerated implementation provides a speedup of 3.94 and 6.35 for SAD and SSD, respectively, over the baseline implementation. We note inhere that during the baseline ARM-only execution, both Cortex cores are fully utilized, while when executing with the use of the SIMD accelerator, a single core is utilized at around 65%. Also, the NEON resources on the Cortex cores are idle during the accelerator enhanced execution.

TABLE I
SSD AND SAD MATCHING.

Platform	ARM Cortex A9	SIMD Accelerator
Frequency [MHz]	667	100
SSD Rate [MM/s]	2.11	13.40
SSD Speedup	1	6.35
SAD Rate [MM/s]	2.34	9.22
SAD Speedup	1	3.94

The SIMD accelerator performs significantly better on the SSD metric, as it does not require conditional execution. Setting up and disabling conditional execution requires extra clock cycles to be used when computing the SAD metric. On the ARM, however, the nature of NEON vector instructions causes SSD execution to be about 10% slower than that of SAD, which is the opposite of what we observe on the accelerator. This happens because of the built-in support for the *VABA* (Vector Absolute Difference and Accumulate) instruction.

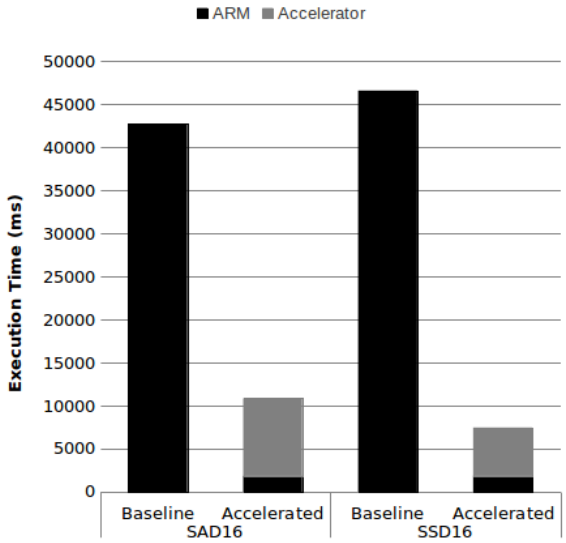

Fig. 4. Profiling of Execution Time.

Figure 4 presents an execution time break-down corresponding to the baseline and accelerated matching. We can observe that even during accelerated matching, a significant proportion of the execution time is occupied by tasks executed on the ARM processor. These are mainly tasks related to decisions on the rejection of matching pairs of keypoints in some circumstances. Further speedup could be attained by moving these decision processes to a separate thread and performing them while matching data are received from the accelerator.

C. Energy

The dual-core ARM CPU consumes, according to Zedboard documentation, a maximum of 1.25 Watts, yielding on average

TABLE II
ENERGY CONSUMPTION PER 100 MMATCHES.

Platform	TDP[W]	SAD energy [J]	SSD energy [J]
Core i7 2600K	95	83.77	76.98
NVidia GTX680	195	24.23	24.37
NVidia 8800 Ultra	175	–	286.88
ARM Cortex A9	1.25	53.41	59.24
SIMD accelerator	1.2	13.01	8.95

2.23 MM/s. The SIMD accelerator consumes 600 mW according to Xilinx power estimation tools. To this we must add the power consumed by the programmable system to control the accelerator. In total, the accelerator consumes approximately 1.2 Watts. To evaluate the implications of our proposal in terms of energy consumption, we calculate and present in Table II the energy per 100 MM for our scheme and equivalent state of the art implementations. This is by no means a comprehensive energy evaluation, but it gives an estimate of the energy-efficiency of the proposed solution.

The Intel CPU results are measured from demoASIFT, with automatic vectorization on SSE and OpenMP used for parallelization. The GPU results for the NVidia 8800 Ultra on SSD matching were extracted from previous work [5]. We have used the cv::gpu::BruteForceMatcher class from OpenCV 2.4 [3] to measure the performance of the GTX680. It must be noted that for the GPU cases, only the device Thermal Design Power (TDP) was taken into account, while the power required by the host for control functions was ignored. In this case we feel that the use of TDP instead of measured power is justified since the chips are fully utilized, including vector resources for the CPU parts, while running the matching benchmarks. Table II suggests that the most energy efficient platform is the accelerated Zynq system, followed by the GTX680, and the baseline Zynq system. The GTX680 performs well because it can exploit the large amount of parallelism in the matching application and delivers 800 MM/s on both metrics, thereby compensating for its high power consumption. The baseline Zynq system is competitive with regard to energy because it consumes very little power. The 8800 Ultra GPU is an older generation chip and its performance is much less than the GTX680, resulting in higher energy consumption. The proposed accelerated system is at least two times more energy efficient than the other platforms.

VI. CONCLUSIONS

We have presented a hardware SIMD accelerator architecture specifically tailored for similarity matching in computer vision algorithms. The accelerator is designed to work in conjunction with an embedded processor and enable high matching throughput for mobile applications energy-constrained applications like robotics. We have implemented this architecture in the Zynq-7000 system-on-chip on the Zedboard development platform, using a Xillybus core for data transfer between the ARM processor and the accelerator. Evaluation has revealed that the SIMD accelerator is able to achieve 4-6x better SIFT descriptor matching throughput than a Cortex A9 processor, despite the FPGA implementation and 100MHz operating frequency. This performance is delivered at roughly 3x less energy consumption and similar power consumption. The accelerated system is 40% more energy effective even than Intel Core i7 2600K and Nvidia GTX680 when executing the SIFT matching benchmark.

ACKNOWLEDGEMENT

Part of this work was carried out with funding and support from POSDRU/159/1.5/132397 ExcelDoc program.

REFERENCES

[1] Xillybus. "http://xillybus.com".
[2] Zedboard. "http://www.zedboard.org/".
[3] Gary Bradski. The OpenCV library. *Doctor Dobbs Journal*, 25(11):120–126, 2000.
[4] M. Brown and D.G. Lowe. Recognising panoramas. In *Proceedings of the Ninth IEEE International Conference on Computer Vision*, volume 2, page 5, 2003.
[5] A. Chariot and R. Keriven. GPU-boosted online image matching. In *19th International Conference on Pattern Recognition*, pages 1–4, 2008.
[6] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.
[7] Pierre Esterie, Mathias Gaunard, Joel Falcou, et al. Exploiting multimedia extensions in C++: A portable approach. *Computing in Science & Engineering*, 14(5):72–77, 2012.
[8] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel AVX: New frontiers in performance improvements and energy efficiency. *Intel White paper*, 2008.
[9] Holger Flatt, Sebastian Hesselbarth, Sebastian Flügel, and Peter Pirsch. A modular coprocessor architecture for embedded real-time image and video signal processing. *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 241–250, 2007.
[10] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using GPU. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–6. IEEE, 2008.
[11] MC Kus, M. Gokmen, and S. Etaner-Uyar. Traffic sign recognition using Scale Invariant Feature Transform and color classification. In *23rd International Symposium on Computer and Information Sciences, ISCIS'08*, pages 1–6. IEEE, 2008.
[12] D.G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.
[13] Krystian Mikolajczyk, Tinne Tuytelaars, Cordelia Schmid, Andrew Zisserman, Jiri Matas, Frederik Schaffalitzky, Timor Kadir, and L Van Gool. A comparison of affine region detectors. *International journal of computer vision*, 65(1):43–72, 2005.
[14] J.M. Morel and G. Yu. ASIFT: A new framework for fully affine invariant image comparison. *SIAM Journal on Imaging Sciences*, 2(2):438–469, 2009.
[15] Mike Santarini. Zynq-7000 EPP sets stage for new era of innovations. *Xcell journal*, 75:8–13, 2011.
[16] S. Se, D. Lowe, and J. Little. Vision-based mobile robot localization and mapping using scale-invariant features. In *IEEE International Conference on Robotics and Automation, Proceedings 2001 ICRA*, volume 2, pages 2051–2058. IEEE, 2001.
[17] Linda Shapiro and George C Stockman. *Computer Vision. 2001*. Prentice Hall, 2001.
[18] Gheorghe Stefan. The CA1024: A massively parallel processor for cost-effective HDTV. In *Spring Processor Forum: Power-Efficient Design*, pages 15–17, 2006.
[19] Tinne Tuytelaars and Cordelia Schmid. Vector quantizing feature space with a regular lattice. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8. IEEE, 2007.
[20] Stephan Wong, Bastiaan Stougie, and Sorin Cotofana. Alternatives in fpga-based sad implementations. In *Proceedings of the IEEE International Conference on Field-Programmable Technology*, pages 449–452, 2002.