



# Ultra low-power visual odometry for nano-scale unmanned aerial vehicles

**Conference Paper****Author(s):**

Palossi, Daniele ; Marongiu, Andrea; Benini, Luca 

**Publication date:**

2017

**Permanent link:**

<https://doi.org/10.3929/ethz-b-000192074>

**Rights / license:**

[In Copyright - Non-Commercial Use Permitted](#)

**Originally published in:**

<https://doi.org/10.23919/DATE.2017.7927257>

**Funding acknowledgement:**

688860 - High-Performance Embedded Real-time Architectures for Low-Power Many-Core Systems (SBFI)

# Ultra Low-Power Visual Odometry for Nano-Scale Unmanned Aerial Vehicles

Daniele Palossi\*, Andrea Marongiu\*<sup>†</sup>, Luca Benini\*<sup>†</sup>

\*IIS, ETH Zürich, <sup>†</sup>DEI, University of Bologna  
{dpalossi, amarongiu, lbenini}@iis.ee.ethz.ch,  
{a.marongiu, luca.benini}@unibo.it

**Abstract**—One of the fundamental functionalities for autonomous navigation of Unmanned Aerial Vehicles (UAVs) is the hovering capability. State-of-the-art techniques for implementing hovering on standard-size UAVs process camera stream to determine position and orientation (visual odometry). Similar techniques are considered unaffordable in the context of nano-scale UAVs (i.e. few centimeters of diameter), where the ultra-constrained power-envelopes of tiny rotor-crafts limit the on-board computational capabilities to those of low-power micro-controllers. In this work we study how the emerging ultra-low-power parallel computing paradigm could enable the execution of complex hovering algorithmic flows onto nano-scale UAVs. We provide insight on the software pipeline, the parallelization opportunities and the impact of several algorithmic enhancements. Results demonstrate that the proposed software flow and architecture can deliver unprecedented GOPS/W, achieving 117 frame-per-second within a power envelope of 10 mW.

## I. INTRODUCTION

Unmanned Aerial Vehicles (UAVs) are increasingly being used for practical applications such as the inspection of industrial facilities or cultivated fields [1], assistance in natural disaster or hazardous areas [2], First-Person-View (FPV) camera-recording or various surveillance tasks.

Like it happened in many other fields, also in robotics the miniaturization of vehicles is one of the major trends of evolution. In this context Commercial-Off-The-Shelf (COTS) quadrotors have already reached the *nano-scale*, featuring only few centimeters in diameter and few tens of grams in weight. The research forefront is focusing on the next generation of *pico-scale* [3]. Such small-scale devices, to date, lack the autonomous navigation capabilities that their larger counterparts boast, since their computational resources, heavily constrained by their tiny power envelopes, are totally inadequate for the execution of sophisticated computer vision workloads.

In this work we study and demonstrate the feasibility of reliable autonomous *hovering* on a *nano-size* quadrotor. Hovering is the capability of the vehicle to maintain a desired altitude of flight over time, autonomously correcting its flight asset; a basic, yet fundamental block for the development of fully autonomous UAVs [4] [5].

State-of-the-art hovering on standard-size UAVs relies on visual algorithmic pipelines to precisely determine position and orientation by analyzing video streams from an on-board camera, i.e. Visual Odometry (VO) [6]. Such techniques are typically not affordable at the nano-scale; most existing COTS systems in this category (e.g. *Bitcraze CrazyFlie 2.0* - <http://www.bitcraze.se>) implement such functionality by means of low-power and noisy sensors like barometers.

Tab. I shows a taxonomy of the most popular and used classes of vehicles. From the numbers reported here, it is undeniable that energy-efficient architectures, highly optimized

Vehicle Class	⊙ : Weight [cm:Kg]	Power [W]	On-board Device
<i>std-size</i> [5]	~ 50 : ≥ 1	≥ 100	Desktop
<i>micro-size</i> [7]	~ 25 : ~ 0.5	~ 50	Embedded
<i>nano-size</i> [4]	~ 10 : ~ 0.05	~ 5	MCU
<i>pico-size</i> [3]	~ 2 : ~ 0.005	~ 0.1	ULP

TABLE I  
ROTORCRAFT UAVS TAXONOMY BY VEHICLE CLASS-SIZE.

software and new classes of algorithms are compulsory, if state-of-the-art navigation capabilities (e.g. hovering, path-planning, object tracking, obstacle avoidance, etc.) are to be brought onto the most challenging *nano-size* and *pico-size* UAVs. Wood *et al.* [3] estimate in 5 mW the power budget for on-board computation on *pico-size* UAVs, showing how this only accounts for 5% of the total, the rest being used by the propellers (86%) and the low-level control parts (9%). Palossi *et al.* [8] show how parallelism is a key asset to achieve energy-efficient path planning using only on-board computational resources for *standard-size* UAVs.

At the *micro-scale* there are several works able to map high level navigation skills in “relatively” low-power architectures, but they are still an order of magnitude away from the *nano-scale* power-budget [6] [7].

Moving to our class of interest (*nano-scale*), the state of the art is represented by solutions that take advantage of commercial low-power MCUs to meet the hard power-budget constraints imposed by this class of vehicles [4]. Their solution requires comparatively much simpler electronics and can rely on the available on-board MCU, but the method does not reach the accuracy of techniques based on feature tracking.

Different from all previous approaches, in this work we propose and demonstrate how the emerging ultra-low-power parallel computing paradigm [9] (and associated software optimizations) satisfies the performance requirement of VO-based *hovering* [5] within the limited power envelopes of *nano-* and *pico-scale* UAVs. Our solution is capable of delivering up to 117 Frames Per Second (FPS) within a power-envelope of 10 mW. We discuss several acceleration opportunities, spanning from coarse-grained (thread-level) parallelization to algorithmic enhancement to make effective use of available hardware features and we ensure there is sufficient compute bandwidth left to execute other typical functionalities required by autonomous UAVs.

## II. SNAPSHOT ALGORITHM

Our software pipeline is based on the monocular hovering technique described in [5], called *Snapshot Algorithm*. This algorithm has been proven to provide a very high accuracy.

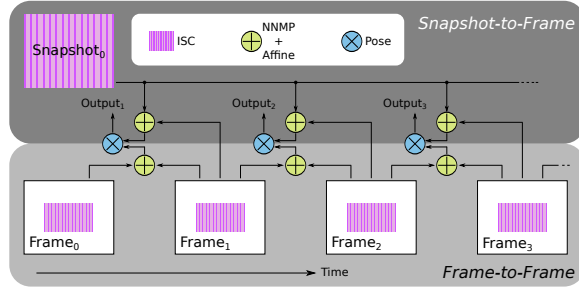


Fig. 1. Algorithmic flows: *Snapshot-to-Frame* and *Frame-to-Frame*.

From an algorithmic point of view, the technique can be summarized in the following four steps: image binarization with *incremental sign correlation* (ISC) [10], template matching between two images using *number of non-matching points* (NNMP) [11], computation of the *affine matrix transformation* (Affine) and the *pose estimation* (Pose). In terms of computational weight the percentage of time spent on each stage can be stated as: NNMP 85%, ISC 10%, Affine 2.5% and Pose 2.5%.

**ISC.** This binarization technique, given in Eq. 1, is aimed to provide an image representation robust to lighting variations. It derives the binary image (i.e.  $I_b$ ) from the intensity image (i.e.  $I$ ):

$$I_b(i, j) = \begin{cases} 1 & \text{if } I(i, j+1) > I(i, j), \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

**NNMP.** Starting from two binary images and a set of *Feature Points* (FPs), in this stage we focus on the *template matching*. We define *template* or *feature* the specific configuration of all pixels inside a window centered on the FP. Then, we look for the same, or the most similar, template in the previous image and within a parametric *search radius*. As a similarity metric we use NNMP, defined as:

$$\text{NNMP}(x, y) = \sum_{i,j=0}^{N-1} I_b(i, j, t) \oplus I_b(i+x, j+y, t-1) \quad (2)$$

Eq. 2 calculates the NNMP between two windows of  $N \times N$  pixels ( $px$ ) computing the *exclusive or* (i.e. XOR  $\oplus$ ), where  $(x, y)$  denotes the radius search. After computing the NNMP for all the search points within the radius search, the pair with the minimum NNMP is considered the matching point of our reference FP, thus reflecting the image motion of the template  $I_b(i, j, t)$ .

**Affine.** Once we have tracked, for each FP, the corresponding point in the previous image, we can initialize the two matrices  $C_1$  and  $C_2$  needed to build our motion model. Let  $(x_1, y_1), \dots, (x_m, y_m)$  be the FP coordinates and  $(\Delta x_1, \Delta y_1), \dots, (\Delta x_m, \Delta y_m)$  the image motion for each point. We can define:

$$C_1 = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ \dots & \dots & \dots \\ x_m & y_m & 1 \end{bmatrix} \quad C_2 = \begin{bmatrix} x_1 + \Delta x_1 & y_1 + \Delta y_1 & 1 \\ x_2 + \Delta x_2 & y_2 + \Delta y_2 & 1 \\ \dots & \dots & \dots \\ x_m + \Delta x_m & y_m + \Delta y_m & 1 \end{bmatrix}$$

Then, we can build the affine model  $C_1 A = C_2$  [5], where  $A$  is the  $3 \times 3$  *Affine Matrix* that can be solved in the least-squares sense:  $A = C_1^+ C_2$ , in which  $C_1^+ = (C_1^T C_1)^{-1} C_1^T$  is the pseudoinverse of  $C_1$ . Thus, all required operations are matrix transformations, whose outputs are i) the *translational*

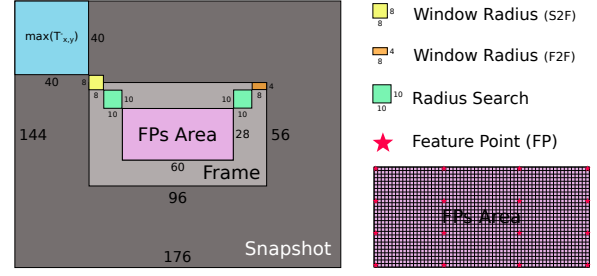


Fig. 2. Configuration used for the: *Snapshot*, *Frame* and *FPs Area*.

*Optic Flow* (OF) ( $Q_x, Q_y$ ), ii) the *translational displacement* ( $T_x, T_y$ ), iii) the *scale* ( $S$ ) and iv) the *rotation angle* ( $\psi$ ).

**Pose.** The first operation of this kernel is a *confidence evaluation* of  $A$ , to reject false estimation. Then, we use the result of the previous stage, performing several arithmetic operations (e.g.  $\sin, \cos, \tan$ , etc.), to estimate *speed* ( $v_x, v_y, v_z$ ) and *position* ( $p_x, p_y, p_z$ ).

Note that the last two kernels operate on floating point data. To achieve the required precision in the retrieved parameters and in computing the *confidence evaluation*, single precision-floating point operations must be adopted. The output of the *pose estimation* is then fed into a cascade of several proportional-integral-derivative (PID) controllers [5] organized as two nested (inner/outer) closed loops.

The *Snapshot Algorithm* tracks movement relying on two distinct visual flows: *snapshot-to-frame* (S2F) and *frame-to-frame* (F2F), as shown in Fig. 1. The former flow is responsible for keeping track of the overall translational displacement ( $T_x, T_y$ ) referred to the snapshot image. To avoid long-term drift effects, a second flow is evaluated for each new frame, getting the translational OF ( $Q_x, Q_y$ ). This OF, based only on the last pair of frames, is used to adjust the translational displacement ( $T_x, T_y$ ), when the flow becomes unreliable.

#### A. Algorithmic Parameters

We retrieved all the key parameters of the algorithms from the reference work [5].

During NNMP computation a sliding window of  $16 \times 16$   $px$  ( $[-8, 8]$  in both directions) is used for the S2F flow and a window of  $16 \times 8$  ( $[-8, 8]$  in  $x$  direction and  $[-4, 4]$  in  $y$  direction) for the F2F flow. The *search radius* for the template matching is stated to be  $10$   $px$  ( $[-10, 10]$  in both directions) and the *maximum translational displacement* ( $T_x^-, T_y^-$ , a limit on the predicted displacement, to avoid going out of the image boundary) is set to  $40$   $px$  ( $[-40, 40]$  in both directions). We inferred the dimension of the *FPs area* (central part of the image where the FPs can lie, shown in Fig. 2) to be  $60 \times 28$   $px$ . Similarly, we can infer the *frame* dimension ( $96 \times 56$   $px$ ).

Concerning *which* and *how many* FPs should be used for tracking, (the paper only states that they consider predetermined FPs, at fixed locations), we select 16 FPs, uniformly distributed across the *FPs area* as depicted in Fig. 2. Considering that the algorithm does not have any “memory” in the F2F flow, it is intuitively reasonable to spatially spread the FPs to gather as much information as possible.

### III. PARALLELIZATION & ALGORITHMIC OPTIMIZATIONS

#### A. PULP Architecture

The platform considered in this work is based on the Parallel Ultra Low Power (PULP v4) architecture (<http://www.pulp->

platform.org) [9]. The PULP cluster has 4 optimized RISC-V cores [12] featuring several enhancements with respect to the reference implementation, including a register-register multiply-accumulate instruction, vectorized instructions for several data types, two hardware loops and support for unaligned load/store operations. Relevant to this work is the *packed-SIMD support* i.e., sub-word parallelism and *bit manipulation support* enhancing the RISC-V ISA with instructions such as `p.extract` (read a register set of bits), `p.insert` (write to a register set of bits), `p.cnt` (count number of bits that are 1), etc. The cores share a L1 multi-banked tightly-coupled data scratchpad memory (*TCDM*). A multi-channel DMA enables fast communication with the L2 memory and external peripherals.

### B. Parallelization

We parallelized ISC and NNMP, the two most computation-intensive stages, with OpenMP. Image binarization within ISC scans the image by rows; different rows are distributed among different cores. The sequential NNMP template matching performs a complete search in a predetermined area of the previous frame (for each FP). Consequently, the simplest parallelization is achieved at the FP granularity. Specifically, every core is responsible for a sub-set of the FPs, proceeding independently and avoiding any conflicts in L1 memory access. This simple parallelization scheme achieves near-ideal speedup.

### C. Algorithmic Optimizations

We focus on the most time-consuming NNMP stage. This operation is straightforwardly implemented using 8 XOR and then summing the 7 output values. However, computation-wise, this is not the optimal solution. The original paper itself suggests that better performance is achieved by using a Look-Up-Table to store the result of the XOR operations.

We improve this initial implementation by exploiting hardware support in the PULP micro-architecture for vectorization. In the ISC kernel we use the vector `v4qu` data type so as to compare four 8-bit elements (i.e. one 32-bit word) in one cycle (i.e., binarizing four pixels in parallel). In the NNMP kernel we use the `p.cnt` extension to retrieve the number of bits set to 1 in a word in one cycle. Since the input data is a binarized image (8-bit per element), and since this operation uses 32-bit operands, we can process four pixels from the template window at the same time.

Then, to hide the cost of L2-to-L1 data transfers (the raw image is stored in the L2), we program the DMA engine for *double buffering*, which enables the transfer of the  $i^{th} + 1$  frame in parallel to the computation of the  $i^{th}$  frame.

As the last two kernels rely on floating point operations, in Sec IV, we evaluate the advantage of using a hardware Floating Point Unit (FPU) as compared to software emulation of floating point operations.

## IV. RESULTS & DISCUSSION

Our experimental evaluation is built considering two main metrics: i) the capability of respecting a given real-time deadline and ii) the capability of performing all the required computations within the allowed power budget. The real-time constraint is imposed by the frame-rate: 30 FPS.

Results are gathered through the official PULP simulator, which accurately models the behavior of the architecture in terms of execution cycles, core stalls, memory conflicts, and

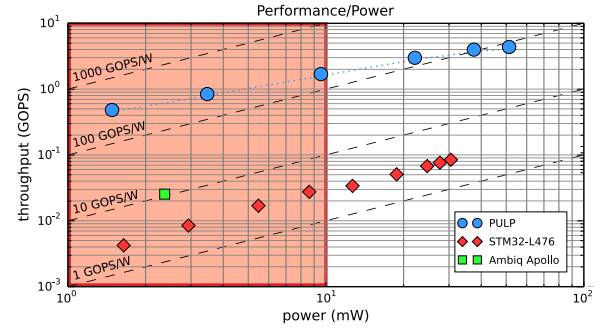


Fig. 3. Energy efficiency of PULP and several MCUs. In red the *pico-size* power-envelope.

cache misses. Architectural parameters such as latencies and energy numbers are extrapolated from an implementation of the platform in 28 nm UTB FD-SOI technology and back-annotated in the simulator. The power-budget is modeled after the reference nano-quadcopter: the *Bitcraze CrazyFlie 2.0*. The power overhead of the FPU is taken from the literature [13] and is estimated in 15%.

### A. Real-Time Performance

In Tab. II, we report execution cycles for the entire application: left and right parts refer to two platform configurations, with and without a hardware FPU, respectively. The various columns refer to SW parallelization (i.e., execution cycles for an increasing number of cores). In addition we report speedup compared to the single-core execution, considering pure SW parallelization (*Parallel SpeedUp*) and SW parallelization plus algorithmic enhancements (*Total SpeedUp*). The last column shows the achieved FPS. Algorithmic enhancements (and their combination) are shown in different rows.

The optimizations named *XOR*, *LUT* and *count* are referred to the second kernel (NNMP), thus only one of them can be enabled at a time. *XOR* and *LUT* are the implementations proposed in the original reference work [5]. In particular, we use *XOR* as the baseline in our experiments. *Count* is our optimization for this second kernel. Concerning the first kernel (ISC), we show the impact of vectorization (*v4qu*), that can be applied orthogonally to the optimizations to the second kernel.

The benefits of parallelization decrease with the amount of algorithmic enhancements due to the Amdahl's law. Both parallelization and algorithmic optimizations are applied only to the first two kernels. Applying the latter reduces the duration of the parallelizable part of the application, thus reducing the benefit of the parallelization overall.

Similar considerations apply for what concerns the impact of the FPU. The floating point operations are only present in the last two kernels, thus if we first apply all the optimizations, the overall performance improvement given by the FPU is more evident. Such behavior can be seen if we compare the percentage boost implied by HW FPU between the first and last line of the table (minimum and maximum optimization level, respectively). The presence of the FPU gives us an overall improvement of 3.5% with 1 core and 4.9% with 4 cores for the minimum level of optimization (first line of the table). For the maximum level of optimization (last line) the same comparison shows a reduction of cycles given by the FPU of 22.3% and 32.0%, respectively.

The last column reports the performance in terms of FPS processed by the entire pipeline (all four kernels) and is visible

		FPU enable								FPU disable							
kernel		# core				SpeedUp				# core				SpeedUp			
ISC	NNMP	1	2	3	4	Parallel	Total	FPS	@200Mhz	1	2	3	4	Parallel	Total	FPS	@200Mhz
—	<i>XOR</i>	85.0	42.5	29.5	21.3	3.9	3.9	9		88.1	44.3	30.4	22.4	3.9	3.9	9	
—	<i>LUT</i>	59.8	30.5	20.5	15.4	3.8	5.5	13		62.0	31.8	21.5	16.3	3.8	5.4	12	
<i>v4qu</i>	<i>LUT</i>	59.8	30.4	20.2	15.4	3.8	5.5	13		62.0	31.7	21.4	16.3	3.8	5.4	12	
<i>v4qu</i>	<i>count</i>	6.6	3.4	2.5	1.7	3.8	50.0	117		8.5	4.5	3.0	2.5	3.4	35.2	80	

TABLE II  
OPENMP PARALLELIZATION + ALGORITHMIC OPTIMIZATION, PERFORMANCE WITH AND WITHOUT FPU [ $10^9$  CYCLE].

how the *count* optimization is boosting the frame-rate up to 117 FPS for the architecture with FPU and up to 80 FPS for the same architecture without the FPU. With this results not only we are guaranteeing the given real-time constraint of 30 FPS, but we are also ensuring there is sufficient room to allocate additional computational power to other typical functionalities required by autonomous UAVs.

### B. Energy Efficiency

As on-board computation accounts for roughly 5% of the overall power consumption of an UAV [3], we can infer that *nano-scale* UAVs can burn around 200-300 *mW* and *pico-size* UAVs around 5-10 *mW* in computation.

Fig. 3 shows the energy efficiency of the PULP platform, compared to two representative ultra low-power commercial MCUs: the STM STM32L47<sup>1</sup> and the Ambiq Apollo<sup>2</sup>.

The shaded box highlights the operating region in which the considered platforms' power consumption match the *pico-size* power envelope. The *CrazyFlie 2.0* is equipped with a 240 *mAh* LiPo Battery (at 3.7 V), has a flight time of 7 minutes, under standard conditions and a total power consumption of 7.6 W. This defines a computational power-budget of 380.6 *mW* (5% of the total). While all the considered platforms nominally operate within this power envelope, only the PULP architecture satisfies the real-time constraint of 30 FPS, as shown in Tab. III. The table shows that several of the evaluated configurations also fit the tighter *pico-scale* compute power envelope, but – again – only PULP delivers the sufficient compute throughput to match the application's real-time requirements (entries in bold in Tab. III). Note that a significant share of computational power is left unused, which allows to implement even more complex navigation capabilities.

## V. CONCLUSION & FUTURE WORK

In this work we demonstrate that *nano-* and *pico-scale* UAV *hovering* can be implemented relying on state-of-the-art visual odometry pipelines, rather than with noisy barometric sensors, as it is commonly done for this energy-constrained class of devices. This is achieved by relying on a parallel, ultra-low-power MCU-class architecture and by optimizing the software implementation to fully exploit the available acceleration opportunities. This solution surpasses by far the required throughput of 30 FPS (we achieve 117 FPS in a power-envelope of 10 *mW*) leaving enough computational bandwidth to execute additional tasks, paving the way towards fully autonomous *pico-scale* UAVs.

<sup>1</sup>STMicroelectronics STM32L476xx Datasheet - [www.st.com](http://www.st.com)

<sup>2</sup>Ambiq Apollo Data Brief - [www.ambiqmicro.com](http://www.ambiqmicro.com)

Device@Frequency [Mhz]	with FPU [mW]	without FPU [mW]	FPS (FPU yes/no)
STM32L47@32	8.9	—	0.5 / —
Apollo@24	11.0	—	0.4 / —
PULP@55	1.3	1.1	<b>32 / 22</b>
PULP@115	3.0	2.6	<b>67 / 46</b>
PULP@175	8.4	7.3	<b>102 / 70</b>
PULP@275	17.3	15.0	<b>160 / 110</b>
PULP@355	28.9	25.1	<b>208 / 142</b>
PULP@435	44.9	39.1	<b>255 / 174</b>

TABLE III  
POWER CONSUMPTION AND FRAME-RATE FOR THE EVALUATED ARCHITECTURES (IN GRAY PICO-SIZE COMPLIANT AND IN BOLD MATCH OF THE REAL-TIME CONSTRAINT).

## ACKNOWLEDGEMENTS

This work has been funded by projects EC H2020 HERCULES (688860) and Nano-Tera.ch YINS.

## REFERENCES

- [1] J. Nikolic *et al.*, "A uav system for inspection of industrial facilities," in *Aerospace Conference, 2013 IEEE*, 2013.
- [2] E. Mueggler *et al.*, "Aerial-guided navigation of a ground robot among movable obstacles," in *Safety, Security, and Rescue Robotics (SSRR), 2014 IEEE International Symposium on*, 2014.
- [3] R. Wood *et al.*, "Progress on 'pico' air vehicles," *Int. J. Rob. Res.*, vol. 31, no. 11, 2012.
- [4] A. Briod *et al.*, "Optic-flow based control of a 46g quadrotor," in *Workshop on Vision-based Closed-Loop Control and Navigation of Micro Helicopters in GPS-denied Environments, IROS 2013*, no. EPFL-CONF-189879, 2013.
- [5] P. Li *et al.*, "Monocular snapshot-based sensing and control of hover, takeoff, and landing for a low-cost quadrotor," *Journal of Field Robotics*, vol. 32, no. 7, 2015.
- [6] C. Forster *et al.*, "Svo: Fast semi-direct monocular visual odometry," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2014.
- [7] J. Conroy *et al.*, "Implementation of wide-field integration of optic flow for autonomous quadrotor navigation," *Autonomous robots*, vol. 27, no. 3, 2009.
- [8] D. Palossi *et al.*, "An energy-efficient parallel algorithm for real-time near-optimal uav path planning," in *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 2016.
- [9] D. Rossi *et al.*, "A -1.8V to 0.9V Body Bias, 60 GOPS/W 4-Core Cluster in Low-Power 28nm UTBB FD-SOI Technology," in *Proceedings of the 2015 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference*, 2015.
- [10] S. Kaneko *et al.*, "Robust image registration by increment sign correlation," *Pattern Recognition*, vol. 35, no. 10, 2002.
- [11] B. Natarajan *et al.*, "Low-complexity block-based motion estimation via one-bit transforms," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 7, no. 4, 1997.
- [12] M. Gautschi *et al.*, "A near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *arXiv preprint arXiv:1608.08376*, 2016.
- [13] —, "4.6 a 65nm cmos 6.4-to-29.2pj/flop@0.8v shared logarithmic floating point unit for acceleration of nonlinear function kernels in a tightly coupled processor cluster," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, 2016.