Tactics Overview for Implementing High-Performance Computing on Embedded Platforms

Tactics Overview for Implementing High-Performance Computing on Embedded Platforms

A Elshazly [1], A Elliethy [1] and M A Elshafey [1]

[1]Department of Computer Engineering and Artificial Intelligence, Military Technical College, Cairo, Egypt

E-mail: ahmedelshazly@mtc.edu.eg

Abstract. Future space missions will rely on novel high-performance computing to support advanced intelligent on-board algorithms with substantial workloads that mandates firm real-time and power constraints requirements. Consequently, these advanced algorithms require significantly faster processing beyond the conventional space-grade central processing unit capabilities. Moreover, they require careful selection of the target embedded platform from a diverse set of available architectures along with several implementation tactics to map the algorithms to the target architecture to fully unlock its capabilities. In this paper, we present a study of different architectures and embedded computing platforms for the satellite on-board computers. Moreover, we present a comprehensive overview of recent implementation tactics such as source code mapping and transformations. Additionally, we highlight some optimization techniques such as partitioning and co-designing using hardware accelerators. Finally, we discuss several implementation analysis methodologies to derive optimized code implementations. The top ranked YOLO-v3, as a deep learning based object detection algorithm, is selected as a case study model to be optimized using OpenVINO toolkit. The experimental results show an improvement ratios up to 73%, 41%, and 34% in terms of frames per second, CPU utilization, and cache memory, respectively. The study presented in this paper aims to guide the researchers in the field of high performance embedded computing in terms of different hardware architectures along with several implementation tactics.

1. Introduction

Recently, Cubesat satellites attracted attention on both commercial and academic levels. These satellites have restricted image processing capabilities due to their limited resources. Thus, they may restrain the exploitation of automated and intelligent image on-board processing tasks that can be considered as the future trends for satellite missions. Additionally, these tasks are crucial in some applications that require automated decision support or low latency conclusions [1, 2, 3, 4, 5].

One of the challenging automated task is oriented toward onboard deep learning based applications such as: telemetry compression [6, 7] and image processing [1, 2]. However, deep learning based models are characterized by a larger number of parameters with high degree of precision accuracy. Due to hardware and power limitations of the on-board computer (OBC) of the satellites, in general and Cubesats in particular, efficient optimization techniques should

be applied on the target deep learning based approach taking into consideration the hardware specification of the current OBCs [8, 9, 10, 11, 12].

In this paper, we provide a survey of recommended tactics for efficient optimizations that can be utilized for performing complex computations on the OBC of the satellites. Our survey is followed by a case study of optimizing the pre-trained YOLO-v3 deep learning based model for object detection in order to be accommodated on the OBC of the satellite. In our case study, we provide an experimental comparison between the typical YOLO-v3 deep learning model and the optimized one, in terms of the processing time (frame per second). Moreover, the paper provides a survey about the OBCs that are commonly used in satellite systems.

The paper is organized as follows. Section 2 presents an overview of OBCs. In section 3, an overview of the deep learning model of the YOLO-v3 structure is demonstrated. General tactics for implementing efficient optimization are described in details in Section 4. In Section 5, we present our experimental results in which we apply the optimization tactics presented in the paper on the YOLO-v3 model [13] using the OpenVINO™ [14] toolkit. Conclusions and future work are suggested in section 6.

2. Satellites on-board computers

The satellite OBC is the heart subsystem of the satellite that connects, manages and controls other subsystems. It also handles payload and housekeeping data to accomplish the targeted mission of the satellite [15, 16, 17, 18, 19].

The main tasks of the OBC includes the span [20] of:

- Attitude Determination and Control (ADCS), such that the attitude of the satellite is measured by sensors. i.e. magnetic field and solar sensors, and corrected by magnetorquers.
- Platform monitoring and control (housekeeping functions) through periodically checking measured sensors values (voltage, temperature, attitude, and orbit) for safe operation and monitoring telemetry data sent to the ground station.
- Experimental measurement and analysis, in which operational data and experimental measurements of new sensors, actuators and control techniques are collected, processed and analysed.

The architecture of the OBC belongs mainly to one of the following three architectures showed in Fig. 1 and listed in Table. 1 associated with their advantages and disadvantages.

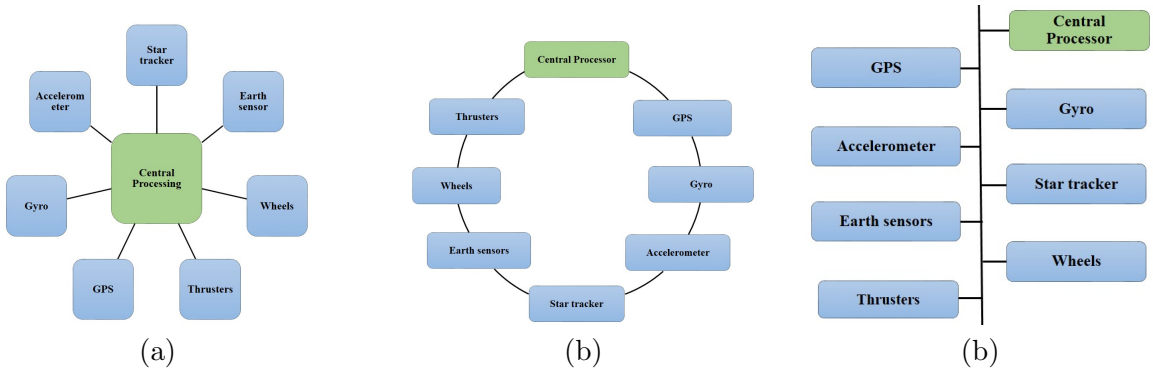

Figure 1. Different OBC architectures: (a) centralized architecture, (b) distributed architecture, and (c) bus architecture.

The hardware of the satellite OBC computing platforms [15, 16, 17, 18, 19, 20] has limited resources. Examples of such platforms are: (1) Microprocessor based, (2) Micro-controllers, (3)

Field Programmable Gate Array (FPGA), (4) Application Specific Integrated Circuit (ASIC) , and (5) System-on-Chip (SOC). The differences of these platforms are summarized below

- Microprocessor based platform. It focus on computation, requires peripheral ICs, supports fastest generic computation, and has typical power consumption higher than 10 Watt.
- Micro-controllers based platform. It focus on embedded systems, has memory integrated, supports integrated peripheral function: ADC, DAC, PWM, I2C, UART, etc, and has a typical power consumption less than 1 Watt.
- FPGA based platform. It is a Re-programmable logic, has IP-cores for specific and complete micro-controllers, provides fast and low power for specific functions, has higher power than micro-controllers.
- ASIC based platform. It is a complete hardware solution for specific application, supports IP-cores for specific function, and has fast and power efficiency.
- SOC based platform. It provides a complete hardware and generic application. It has the smallest complete systems.

Table 1. General OBC architectures presented in Fig. 1.

	Advantages	Disadvantages
Centralized	– Adequate for a small number of well-defined sub-systems with a direction interfaces to the central computer. – High reliability that is achieved by isolating sudden failure at any interface and preventing influence on others.	– Required changes in both hardware and software of the central node in order to connect to a new one. – Large wiring harnesses is needed for transmitting data to multiple recipients.
Distributed	– Small wiring harnesses with the capability of being distributed along the structure of the space-craft . – Limited effect on the central node when adding new nodes.	– Reliability reduction is be-cause each node is required to achieve data transmission to the next node.
Bus	– High processing capacity with high reliability can be achieved through the ability of adding multiple processing units.	– More complex testing proce-dures are relatively required for these systems.

3. YOLO-v3 deep learning model

Our case study in this paper is the optimization of the pre-trained YOLO-v3 deep learning based object detection model. It is a deep neural network that contains a set of 53 convolutional

layers each followed by Batch Normalization (BN) layer and Leaky ReLU (LRELU) activation function. The YOLO-v3 deep network model, shown in Fig. 2, can recognize up to 80 different shapes in images and videos [21].

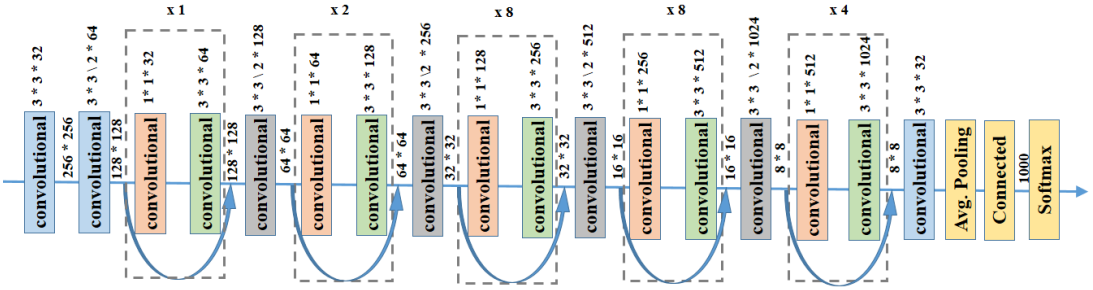

Figure 2. YOLO-v3 fully convolutional network.

The detection phase of YOLO-v3 model is achieved at three different scales by three strides of values 32, 16, and 8. At each scale, a count of 3 bounding boxes are detected by 3 anchors, different anchors for different scales, resulting a total number of 9 anchors is used. For such deep learning based detection model, detections at scales of 13 x 13, 26 x 26 and 52 x 52 are achieved for 416 x 416 input size [21].

4. Efficient optimization tactics for complex computations

Optimizations is used to mitigate the cost the expensive instructions, which results in reducing the execution time. It includes several components that can be either hardware or software. We will summarize them below.

4.1. Hardware accelerators

Hardware accelerators are used to offload the computations from a host processor to an accelerator element in order to enhance the performance and meet the nonfunctional requirements (such as strict execution time limit). There are many types of the H/W accelerators such as ASICs, GPUs and FPGAs. The programming models of these devices exhibit different levels of trade-offs between accuracy and the nonfunctional requirements.

GPU accelerators consists of lightweight cores and on chip memories are used to increase the degree of parallelism for a single program. Originally, GPUs are used for the acceleration of graphical applications but it has been evolved to support many intensive applications. Recently, GPUs are used in embedded systems as hardware accelerators, thanks to their increasing capabilities and capacities [22] [23].

The enhances introduced in CPU cores nowadays, allow them to further support parallelism which includes the support of multi-threading. Single Instruction Multiple Data units (SIMD) is one example, in which the CPU can concurrently operate on multiple data operands and perform the same operation. Another example is the Fused Multiple-add units (FMA) that are used to perform multiple-add or multiple-subtract operations [24].

To enhance the performance of a complex algorithm, it is common to develop the code on different architectures. The code in Fig. 3 shows a code that uses different directives to speak to different targets to measure the execution time for a PC or an embedded system using a Xilinx MicroBlaze [25, 26, 27, 28, 29]. To get the maximum performance from every computing element in the computing platform, it is required to split the code into different sections and map these sections of the code to distinct processing elements in the computing platform [30, 31], in order to meet the nonfunctional requirements. This process can be performed in two ways:

#ifdef embedded_timing
XTime_GetTime(&timeStamp_206983496);
printf("TS-before 'gridsInit': %llu\n", timeStamp_206983496);
#endif
	Timing for MicroBlaze

#ifdef pc_timing { hTimer t ("t1"); #endif	Timing for PC

#ifdef embedded_timing
XTime_GetTime(&timeStamp_206983496);
printf("TS-after 'gridsInit': %llu\n", timeStamp_206983496);
#endif
	Timing for MicroBlaze

#ifdef pc_timing
}
printf("TS-after 'gridsInit': %llu\n", hTimer::elapsed("t1");
#endif
	Timing for PC

Figure 3. Code developing for different targets.

static partitioning and dynamic partitioning. The static partitioning mapping is determined at compile time and remains unchanged at run-time. The static partitioning follows the following steps

(i) Mapping: in which the code is inspected and a static call graph is generated. The graph is G=(N, E), where nodes N represent the function calls while edges E represent the data flow and control that relates the two nodes connected by the edge E.

(ii) Mapping to the target heterogeneous architecture.

(iii) Finding a configuration in which the overall cost

$$\sum_{task} cost(PE_t, task), \tag{1}$$

is minimized. This minimization is required to meet a given performance goal.

Dynamic partitioning [32] on the other hand determines the mapping during run-time according to existing workload and device availability. We use dynamic partitioning because the input size is determined only during the run-time. The efficient processing element is determined based on an estimation model for a particular job.

4.2. Code transformations and optimizations

Developers need to perform the code optimizations in order to meet the nonfunctional requirements. Some basic code optimizations are internally done by the compilers [33] such as the strength reduction. There are many examples of strength reduction such as the replacement of multiplication and division by left and right shifts, respectively. Another example of the strength reduction is when we need to negate a value, as shown in Fig. 4.

unsigned int p2 = * (unsigned int *) &a;
p2 = p2 ^ (1 << 31);
float neg_a = * (float *) &p2;
	B = 105 * A;
	can be substituted by:
	A1 = (A<<4)-A;
	B = (A1<<3)-A1;

Negate a value.	The replacement of multiplication and division by left and right shifts.

Figure 4. Basic code optimizations.

However, compilers do not have all code transformation so some optimizations must be done manually. Example of such manual optimization is the Loops optimizations. Loops are

considered as very important hot-spots as they have a very high impact on performance. There are many types of loop optimizations such as

- Loop splitting: It simply partitions a single loop into many other loops [34] which is very important when considering data parallelism, as shown in Fig. 5 (a).
- Loop Strip-mining: In this transformation we transform a loop into a nested loop[35], as shown in Fig. 5 (b).
- Loop Perforation: In this loop transformation, we only consider a part of the iterations to give approximate results but better performance [36] [37]. An example is shown in Fig. 5 (c).
- Loop Unrolling: To unroll a loop by a factor K is to repeat the body of the loop K times which is very important when considering the usage of SIMD units [38]. An example is shown in Fig. 5 (d).
- Loop UN-switching: In this transformation, the invariant conditions are moved outside the loop to avoid unnecessary processing[39]. An example is shown in Fig. 5 (e).

To perform the code optimization, first we need to perform source code analysis[40]. Source code analysis is used in understanding the structure of the code. In this phase, the procedures are being analyzed to determine which procedures invoke other procedures or which procedures allocate storage and understanding where the execution spends the bulk of its time. But before we speak about the source code analysis we need to understand the data dependencies first.

Data dependency is the relationship between the instructions. We must check for it in order identify the legal optimizations that can be done within the code[41]. There are four types of data dependencies. The first type is True-data dependency which happens when an operand is modified and then read in a later instruction. The second type is Anti-data dependency which happens when an operand is read and then modified in a later instruction. The third type is Output-data dependency which happens when an operand is modified by two instructions. The last type is Input-data dependency which happens when an operand is read by two instructions.

For a compiler to check for data dependency in a loop, the compiler first generates code to check for data dependency at run-time during the first iteration. If no dependencies are detected, then the rest of the iterations can run concurrently. Another optimization is called Loop Permutations, in which the order of execution is changed between two loops in a nested loop. This change reduces the dependency which enables us to use data parallelism as shown in Fig. 6.

Source code analysis can be classified into two types: Static and dynamic code analysis. Static code analysis aims at understanding the properties of the code without running the app. To perform the static analysis we must follow the following steps. First, we need to parse the code to generate a flow graph. Then, we need to identify the data dependencies to decide the legal optimizations that can be done within the code in order to exploit the characteristics of the target computing platform [42] [43]. In dynamic analysis, the goal is to uncover the metrics of performance during execution time. Specifically, the dynamic analysis tries to find the hot-spots which are the prime candidates for optimization. These spots can be found by profiling [44] and instrumenting the code by adding primitives to the code to inspect the code during the run-time. This can by done manually or by using the LARA language [45].

5. Experimental work

In this section, we provide an experimental comparison between the typical YOLO-v3 deep learning model and the optimized one using the optimization tactics presented in Section 4. We utilized the OpenVINO™ [14] toolkit to perform the optimization. First, we give an overview about the OpenVINO toolkit, then we demonstrate the performance metrics used for evaluating the optimization gain, and finally our results are figured out.

Original loop	Splitted loop

`for(i = 0; i < N; i++)`
`A[i] = B[i] + C;`
	`for(i = 0; i < N/2; i++)`
	`A[i] = B[i] + C;`
	`for(i = N/2; i < N; i++)`
	`A[i] = B[i] + C;`

(a) Loop splitting

Original loop	Loop after strip-mining

`for(i=0; i<N; i++) {`
`sum += A[i];`
`}`
	`for(is=0; is<N; is+=S) { // S is the size of the 1-`
	`dimension block`
	`for(i=is; i<min(N,is+S-1); i++) {`
	`sum += A[i];`
	`}`
	`}`

(b) Loop strip-mining

Original loop	Loop after Perforation

`for(i=0; i<N; i++) {`
`sum += A[i];`
`}`
	`for(i=0; i<N; i+=k) {`
	`sum += A[i];`
	`}`

(c) Loop perforation

Original loop	Loop after fully unrolled

`for(i=0; i<4;i++) {`
`c[i] = a[i] + b[i];`
`}`
	`c[0] = a[0] + b[0];`
	`c[1] = a[1] + b[1];`
	`c[2] = a[2] + b[2];`
	`c[3] = a[3] + b[3];`

(d) Loop unrolling

Original loop	Loop after Un-switching

`for(i=0;i<M;i++) {`
`if(op==1)`
`sum += A[i];`
`else`
`sum *= A[i];`
`}`
	`if(op==1)`
	`for(i=0;i<M;i++) {`
	`sum+=A[i];`
	`}`
	`else`
	`for(i=0;i<M;i++) {`
	`sum *= A[i];`
	`}`

(e) Loop un-switching

Figure 5. Different loop optimizations.

D = (1,0) , Original loop	D=(0,1), Permutated loop

`for (i=1; i < N; i++){`
`for (j=1; j < N; j++){`
`C[i][j] = C[i-1][j] + 1;`
`}`
`}`
	`for (j=1; j < N; j++){`
	`for (i=1; i < N; i++){`
	`C[i][j] = C[i-1][j] + 1;`
	`}`
	`}`

Figure 6. Loop Permutations

5.1. OpenVINO™ toolkit

OpenVINO toolkit is an Intel toolkit used to perform tasks such as automatic speech recognition and emulation of human vision using latest artificial neural networks [46, 47, 48]. The model

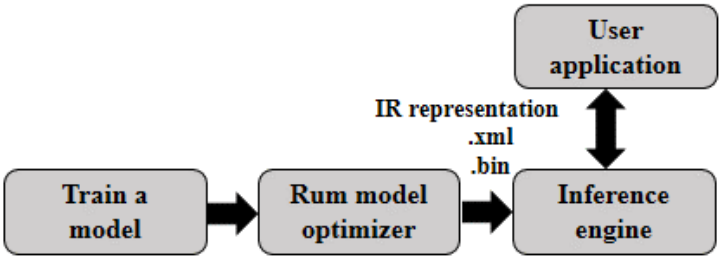

Figure 7. Overview of OpenVINO™ toolkit [14].

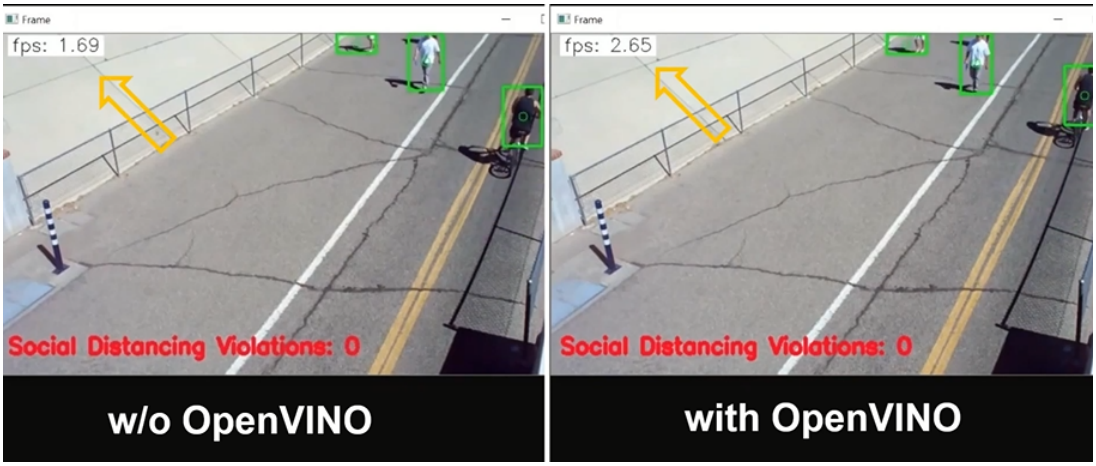

Figure 8. Visual result that shows two frames from the video used in our experiments.

optimizer which is the most important component in the OpenVINO™ toolkit converts the trained models to an Intermediate Representation (IR) (.xml and .bin) to be used in inference operations, as shown in Fig. 7. The Model Optimizer performs a group of code optimizations to produce simpler and faster models and it is applicable for models trained in popular frameworks.

5.2. Performance evaluation metrics

For evaluation the performance gain due to applying optimization techniques, we implement the following three metrics:

- Frames per second: The higher the better. It is the average count of frames accessed per second.
- CPU utilization: The lower the better. It measures the amount of CPU resource requirements for handling the task.
- Cache memory: The higher the better. It indicates the size of data cached, in repeated access cases, to avoid the delay in going back to access the storage memory.

5.3. Results

We have benchmarked the YOLO-v3 on CPU as a processing element without any optimizations and with the optimizations applied by the model optimizer from the OpenVINO toolkit. The application used here is to calculate the social distancing between persons using YOLO-v3. Fig. 8 shows two frames from the video used in our experiments.

Table. 2 represents the performance enhancement in frames per second (FPS), cache memory, and CPU utilization after implementing the optimization techniques using the OpenVINO

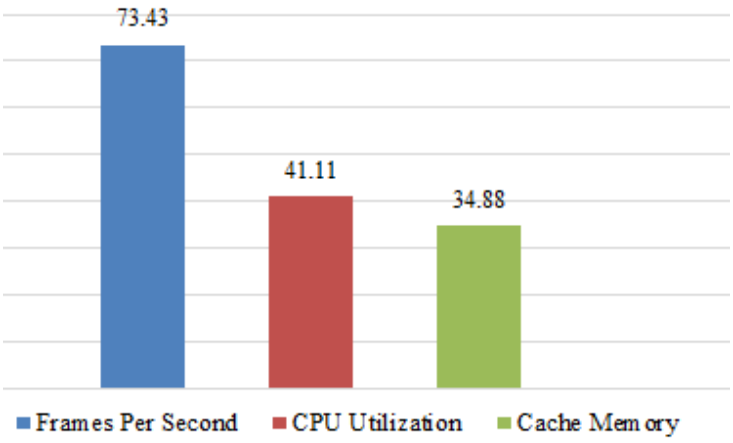

Figure 9. Enhancement ratios (%) due to optimization.

toolkit. The listed results, in table 2, show that the performance gain is about 73%, 41%,

Table 2. Performance enhancement after optimization.

	Without optimization	With optimization
Average FPS	1.28	2.22
Cache memory	911 MB	1.2 GB
CPU utilization	90%	53%

and 34% in terms of frames per second, CPU utilization, and cache memory, respectively, as shown in Fig. 9.

6. Conclusion

This paper demonstrates how we can improve the performance of complex and intensive algorithms, especially intelligent ones, using optimizations such as offloading the computations from the host processor to a co-processor and analysing the code to find the hot-spots and apply code optimizations to them. The paper showed experimentally that code optimization techniques can lead to a better performance in terms of frames per second, CPU utilization, and cache memory, that helps to implement the high-performance computing algorithms on embedded platforms with limited resources as in case of satellite on-board computers.

References

[1] Lim S and Xiaofang C 2014 *2014 IEEE International Conference on Aerospace Electronics and Remote Sensing Technology* (IEEE) pp 137–141
[2] Nzeugaing G N and Biermann E 2016 *Journal of Engineering, Design and Technology*
[3] Liu S and Yang J 2019 *Symmetry* **11** 1373
[4] Tsog N, Behnam M, Sjödin M and Bruhn F 2018 *2018 IEEE Aerospace Conference* (IEEE) pp 1–8
[5] Meß J G, Dannemann F and Greif F 2019 *European Workshop on On-Board Data Processing (OBDP2019)* (European Space Agency)
[6] Mahmoud T A, Shehab A F and Elshafey M A 2020 *Journal of Aerospace Information Systems* 1–8
[7] Shehab A F, Elshafey M A and Mahmoud T A 2020 *2020 IEEE Aerospace Conference* (IEEE) pp 1–11

[8] Huq R, Islam M and Siddique S 2018 *1st China Microsatellite Symposium*
[9] Ibrahim S K, Ahmed A, Zeidan M A E and Ziedan I E 2020 *Ain Shams Engineering Journal* **11** 45–56
[10] Li D, Wang M, Dong Z, Shen X and Shi L 2017 *Geo-spatial information science* **20** 134–140
[11] Xu X, Dong X and Xie Y 2020 *Remote Sensing* **12** 1216
[12] Jeon M J, Kim E and Lim S B 2014 *Aerospace Engineering and Technology* **13** 37–43
[13] Redmon J, Divvala S, Girshick R and Farhadi A 2016 *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* pp 779–788
[14] Abhishek Nandy M B 2019 *Intel Deep Learning Frameworks: Applications of Open Vino and Other Intel Tools* (APress) ISBN 978-1484239810
[15] Tyugashev A and Sygurov Y M 2019 *Journal of Physics: Conference Series* **1368** 042032
[16] Ahmadi A, Kosari A and Malaek S 2018 *IEEE Aerospace and Electronic Systems Magazine* **33** 34–51
[17] Kim Y V 2020 *Satellite Control System: Part I-Architecture and Main Components* (IntechOpen)
[18] Tyugashev Andrey O and Sergei 2019 *International Symposium on Intelligent and Distributed Computing* (Springer) pp 404–413
[19] Tyugashev A and Sergei 2019 *Procedia Computer Science* **159** 1019–1026
[20] Schwenk K, Ulmer M and Peng T 2018 *High-Performance Computing in Geoscience and Remote Sensing VIII* vol 10792 (International Society for Optics and Photonics) p 1079205
[21] Redmon J and Farhadi A 2018 *arXiv preprint arXiv:1804.02767*
[22] Wu R, Zhang B and Hsu M 2009 *Proceedings of the Combined Workshops on UnConventional High Performance Computing Workshop plus Memory Access Workshop* UCHPC-MAW '09 (New York, NY, USA: Association for Computing Machinery) p 1–6 ISBN 9781605585574
[23] Reuther A, Michaleas P, Jones M, Gadepally V, Samsi S and Kepner J 2019 *2019 IEEE High Performance Extreme Computing Conference (HPEC)* pp 1–9
[24] Corp I 2016 *Intel® 64 and IA-32 architectures optimization reference manual* (Intel Corp) order number: 248966-033
[25] Cardoso J M and Weinhardt M 2016 *FPGAs for Software Programmers* vol 1 (Springer) ISBN 978-3-319-26408-0
[26] Pilato C, Mantovani P, Di Guglielmo G and Carloni L P 2016 *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **36** 435–448
[27] Josipović L, Ghosal R and Ienne P 2018 *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* pp 127–136
[28] Mantovani P, Di Guglielmo G and Carloni L P 2016 *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)* (IEEE) pp 204–211
[29] Carloni L P, Cota E G, Guglielmo G D, Giri D, Kwon J, Mantovani P, Piccolboni L and Petracca M 2019 *Proceedings of the Workshop on Computer Architecture Education* pp 1–8
[30] Yan X H, He F Z and Chen Y L 2017 *Journal of Computer Science and Technology* **32** 340–355
[31] Hou N, Yan X and He F 2019 *Design Automation for Embedded Systems* **23** 57–77
[32] Stitt G, Lysecky R and Vahid F 2003 *Proceedings of the 40th Annual Design Automation Conference* DAC '03 (New York, NY, USA: Association for Computing Machinery) p 250–255 ISBN 1581136889
[33] Triantafyllis S, Vachharajani M, Vachharajani N and August D I 2003 *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* pp 204–215
[34] Liu J, Wickerson J and Constantinides G A 2016 *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* pp 72–79
[35] Weiss M 1991 *Proceedings of the 5th International Conference on Supercomputing* ICS '91 (New York, NY, USA: Association for Computing Machinery) p 234–243 ISBN 0897914341
[36] Misailovic S, Sidiroglou S, Hoffmann H and Rinard M 2010 *2010 ACM/IEEE 32nd International Conference on Software Engineering* vol 1 pp 25–34
[37] Sidiroglou-Douskos S, Misailovic S, Hoffmann H and Rinard M 2011 *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* ESEC/FSE '11 (New York, NY, USA: Association for Computing Machinery) p 124–134 ISBN 9781450304436
[38] Murthy G S, Ravishankar M, Baskaran M M and Sadayappan P 2010 *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)* pp 1–11
[39] Lokuciejewski P, Gedikli F and Marwedel P 2009 *Proceedings of Th 12th International Workshop on Software and Compilers for Embedded Systems* SCOPES '09 (New York, NY, USA: Association for Computing Machinery) p 11–20 ISBN 9781605586960
[40] Binkley D 2007 *Future of Software Engineering (FOSE '07)* pp 104–119
[41] Wu Y, Sun J, Liu Y and Dong J S 2013 *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* pp 323–333
[42] Keith Cooper L T 2011 *Engineering: A Compiler 2nd Edition* (Morgan Kaufmann) ISBN 978-0120884780

[43] Louridas P 2006 *IEEE Software* **23** 58–61
[44] Dang K N, Meyer M, Okuyama Y and Abdallah A B 2017 *The Journal of Supercomputing* **73** 2705–2729
[45] Cardoso J a M, Carvalho T, Coutinho J G, Luk W, Nobre R, Diniz P and Petrov Z 2012 *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development* AOSD '12 (New York, NY, USA: Association for Computing Machinery) p 179–190 ISBN 9781450310925
[46] Kustikova V, Vasiliev E, Khvatov A, Kumbrasiev P, Vikhrev I, Utkin K, Dudchenko A and Gladilov G 2019 *International Conference on Analysis of Images, Social Networks and Texts* (Springer) pp 11–23
[47] Mathew G, Ramachandran S S and Suchithra V 2019 *TENCON 2019-2019 IEEE Region 10 Conference (TENCON)* (IEEE) pp 1783–1788
[48] Andriyanov N 2020 *2020 Systems of Signal Synchronization, Generating and Processing in Telecommunications (SYNCHROINFO)* (IEEE) pp 1–5