UDC 681.3 https://doi.org/10.15407/pp2019.04.003

Dmytro V. Rahozin

A RESOURCE LIMITED PARALLEL PROGRAM MODEL

Modern parallel programs run in a complex, resource-limited environment, and this raises the new requirements for resource consumption and execution stability of long running processes. In order to help with checking resource constraints for such parallel software a resource-limited parallel program formal model was developed. The model expresses the resource and time constraints and is suitable both for fine grained and coarse-grained parallelism in programs. For higher degrees of parallelism (at independent procedure level, bigger loop iterations, large computing blocks for graphics, video and neural network processing) the interpretation of formal model can be done in run-time and avoid dead locks and hangs during resource allocation. We are discussing several modern software frameworks that are able to integrate the functionality to interpret the model and check the feasibility of the set of parallel programs running on hardware simultaneously with resource and time limitations. Real world tasks – neural network inference, video processing, general purpose computing on GPU – which get benefits after enabling such models - are discussed.

Key words: formal model, parallel computing, parallel computing on graphics processing units.

Introduction

Nowadays parallel programming still is an activity, which cannot be planned well in terms of spent time and parallelization quality (i. e. writing a good code, which fits underlying architecture well in limited time and money). This can be easily explained by increasing complexity of underlying computing architecture of the modern hardware. The parallel architectures of 80ies had the minimal number of cache levels, simpler memory hierarchy and quite simple MESI cache protocols, but even in this case the performant implementation of basic BLAS/LAPACK procedures was a tricky thing. Today the state-of-art parallel computing is based on Graphics Processing Units (GPUs) which has quite complex memory hierarchy, complicated data exchange paths between central processors and GPUs and highly parallel SIMD-type computational units. The sophisticated tuning of GPU-based programs is mostly economically ineffective due to high labor cost and longer time-to-market. This involves the use of programming tools, which enable more high-level programming structures than just basic CUDA or OpenCL code.

The good examples of such tool are the neural network descriptions tools. Starting from Caffe tool [1], the inference and deep learning phases in neural networks are based on a high-level description of the network. The network therefore is defined as a pipeline of standard computing blocks (usually more than 50 types of blocks are defined) and data paths between them. This definition format practically enables the development of a neural network from idea to optimized implementation without moving down to hand programming the neural network behavior. The success of Caffe lead other development groups to implement alternate or competitive neural network definition languages (Darknet [2]) or integrate a set of packages into programming languages and frameworks (TensorFlow [3]), sometimes incompatible. Of course, the use of high level definitions is not limited to neural network structure descriptions.

1. Practical expressions of high-level programming structures

Usually the simplified description of a big computational pipeline lacks ability for semi-precise optimization of resource use, as there is no way to estimate possible changes in computation pipeline which can use GPU resources more effectively or estimate how efficiently the neural network pipeline will run on GPU simultaneously with other applications. The optimizations of such computing pipeline are hard and strictly depend on underlying hardware architecture, which is a trade secret for the main hardware market players.

Other example is OpenCV framework [4] extensions for matrix/image operations on

GPUs. Program can simply move matrices or image operations to GPU without critical changes in code, but the GPU memory management is done by hand by moving the image data between central processor unit (CPU) and GPU memory. Such a helper (here helper mean the OpenCV functions set) does not deal with actual computational resource allocation on GPU, so the user cannot reach the best performance on GPU side without additional and expensive platform-dependent optimizations.

An interesting example is Gstreamer [5] media processing pipeline. Started as an eager media processing pipeline more than 30 years ago, now it is enriched with data processing on GPU side. A good example is Gstreamer-based DeepStream framework from Nvidia [6], which provides optimized neural network inference at GPU side. For this case the data processing pipeline may reach more than a hundred components, which are controlled by a dozen of threads. Despite of many efforts applied from Nvidia side, basic Gstreamer functionality still uses only rudimentary and hidden thread control and does not bother with resource allocation (either GPU memory or GPU computational power).

Looking through these three cases the can lighten the following flaws: 1) too high-level model for neural network description, where resource allocation (computational, memory, communication) is out of scope of the model; 2) too low-level model where the user should deal with resources by hand but not by describing them in model description; 3) even ignorance of resource allocation. All these flaws greatly affect the parallel software system which goes from proof of concept stage to customer environment (i. e. productization process) experiencing all computational, memory and communication resource constraints. Due to modern shift of heavy computations to GPU side, the productization process becomes a headache for engineers, full of bug hunting and code tricks to resolve various resource constraints. The problem appears more complex in case if applications are long-running, so the memory and other resource leaks can be fatal.

In order to overcome the difficulties of constraint management process we propose a model, which allows to introduce resource management process to frameworks such as OpenCV, TensorFlow or Gstreamer. It should be noted that these frameworks are different in levels of parallelism expression. OpenCV provides only basic parallelism, Gstreamer – thread-level and Caffe/Tensorflow – coarse-grain. Still we model will fit to all parallelism types.

There are a lot of previous work introducing various concepts for parallel software models, starting from Timed Finite Automatons [7]. This article introduces resource constraints for parallel software models, which are necessary for executing parallel program in modern highly parallel and resource constrained environment.

2. A Model of Resource-Constrained Parallel System

The definition of the system is derived from [8], extending the previous definitions. The word “real-time” is not used intentionally, as the complex parallel system may have both hardware and resource constraints that prevent real-time behavior.

We consider a discrete-time model where the time is represented as a set of non-negative integer values denoted by $\mathbb{N}$. The time progress is measured by clocks, which are non-negative integer variables increased constantly by one for some time. If compared to [8] the definition “synchronously” is dropped, as in real life system usually a common reference clock is used. For the set of clocks $\mathbf{X}$, a *valuation* $v: \boldsymbol{X} \to \mathbb{N}$ is defined – it is a function associating with each clock its value $v(x)$. For a clocks subset $\boldsymbol{X}' \subseteq \boldsymbol{X}$ and a clock value $l \in \mathbb{N}$ we denote by $v(\boldsymbol{X}', l, x)$ the valuation that coincides with v for all clocks $x \in \boldsymbol{X} \backslash \boldsymbol{X}'$, and that associates l to all clocks $x \in \boldsymbol{X}'$. It is defined by:

$$v(\boldsymbol{X}', l, x) = \begin{cases} l \text{ if } x \in \boldsymbol{X}' \\ v(x) \text{ otherwise.} \end{cases}$$

Guards are used to specify when actions are enabled. Simple constrains over clocks **X** are considered. The grammar allows to build general constrains over clocks:

$$c \coloneqq true \mid false \mid x \ \leq k | x < k | x \geq$$
$$\geq k | x > k | c \wedge c | c \vee c | \neg c.$$

The evaluation of a clock constraint c for a valuation v of clocks **X** denoted by $c(v)$ is obtained by replacing each clock x by its value $v(x)$.

A *guard* g is a clock constraint c with an urgency type $\tau \in \{n, d, u\}$, denoted by $g = [c^\tau]$. Here urgency types are used to specify the need of the action in case if action execution is enabled (i. e. when the clocks constraint is true). Non-urgent actions are denoted by **n**, delayable actions (which should be executed during their enable time interval) by **d**, urgent actions (should be executed as soon as they are enabled) are denoted by **u**. The predicate $urg[g]$ that characterizes the valuations of clocks for which the guard $g = [c^\tau]$ is *urgent* is defined by:

$$urg[g](v) \Leftrightarrow$$
$$\Leftrightarrow \begin{cases} \text{false} & \text{if } g \text{ is nonurgent } (\tau = \text{n}) \\ c(v) \wedge \neg c(v + K) & \text{if } g \text{ is delayable } (\tau = \text{d}) \\ c(v) & \text{if } g \text{ is urgent } (\tau = \text{u}) \end{cases}$$

The set of guards over the set of clocks **X** is denoted as **G**(**X**).

For given guards

$$g_1 = [c_1]^{\tau_1} \; and \; g_2 = [c_2]^{\tau_2},$$

the conjunction of g_1 and g_2 is denoted by $g_1 \wedge g_2$ and is defined by $g_1 \wedge g_2 = [c_1 \wedge c_2]^{max\, \tau_1, \tau_2}$, considering that urgency types are ordered as follows: $n < d < u$. So, for given guard g=[c]$^\tau$ and a valuation v, we also write $g(v)$ for the expression $c(v)$.

Additionally, the resource constraints are defined.

Additionally to model in [8] let us define the resource set **P** = { p_i^t}, where t is the resource type, I is the resource index for some number of identical resources in computational system. Also the resource p_i^t may be allocated in part – $p_i^t(k)$, where k is the amount of resource p_i^t.

Let us define availability function for resource: $avl(p_i^t(k))$, is it *true* is case if the resource is available.

3. Abstract model with constraints

Definition 1. Abstract model with constrains.

An *abstract model* is a timed automaton M=(**A**,**Q**,**X**,**P**,→) such that:

- **A** is a finite set of (*observable*) actions. In addition to actions **A** internal actions β. The set of actions **A** ∪ {β} is denoted as $\mathbf{A}^\beta$;
- **Q** is a finite set of control locations;
- **P** is the resource set;
- **X** is finite set of clocks →⊆ **Q**×($\mathbf{A}^\beta$×**G**(**X**)×$2^{\mathbf{X}}$)×**Q** is a finite set of labeled transitions. A transition is a tuple (q, a, g, r, p, q') where a is an action executed by the transition, g is guard over **X**, r is a subset of clocks that are reset by the transition and p is the resource necessary to be allocated during the transaction. We write $q \xrightarrow{a,g,r,p} q'$ for $(q, a, g, r, p, q') \in\to$.

An abstract model describes the platform-independent behavior of the system.

Definition 2. (Abstract model semantics). An abstract model $M =$ $= (\boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{X}, \boldsymbol{P} \to)$ defines a transition system *TS*. States of *TS* are pairs (*q*,*v*), where *q* is a control location of *M* and *v* is a valuation of the clocks *X*.

Actions: We have $(q, v) \overset{a}{\to}$ $\overset{a}{\to} (q', v[r \longmapsto 0]) if \; q \xrightarrow{a,g,r,p}$ $\xrightarrow{a,g,r,p} q' in \; M \; and \; g(v) is \; true$ and *avl*(*p*) is *true*.

Time steps. For a waiting time $\delta \in \mathbb{N}, \delta > 0$, we have $(q, v) \overset{\delta}{\to} (q, v + \delta)$ if for all transitions

$$(q \xrightarrow{a,g,r} q' of \; M \; for \; for \; all \; \delta' \in$$
$$\in [0, \delta[, \neg urg[g](v + \delta').$$

Urgency corresponds to priorities induced by the timing constraints: urgent transitions have priority compared to other possible transitions. We denote by *wait*(*q*,*v*) the maximal waiting time allowed at (*q*,*v*). Always *wait*(*q*,*v*+*δ*) = *wait*(*q*,*v*) – *δ* for all $\delta \in$

$\in [0, wait(q,v)]$, and is formally defined as follows:

$$wait(q,v) =$$
$$= \boldsymbol{min}\left(\left\{\delta \geq 0 \middle| \vee_{q\xrightarrow{a_i,g_i,r_i,p_i}q_i} urg[g_i](v+\delta)\right\} \cup \cup \{+\infty\}\right).$$

For an abstract model $M = =(\mathbf{A},\mathbf{Q},\{x\},\mathbf{P},\rightarrow)$, a finite execution sequence of M from an initial state (q_0,v_0) is a maximal sequence of observable actions and time-steps $(q_i,v_i) \rightsquigarrow^{\sigma_i} (q_{i+1},v_{i+1})$, $\sigma_i \in \boldsymbol{A} \cup N$ and $i \in \{0,1,2,\dots,n\}$ $(i \in \mathbb{N})$, such that $\rightsquigarrow$ is the transitive closure of $\rightarrow$ for β-transitions, that is $(q_i,v_i) \rightsquigarrow^{\sigma_i} (q_{i+1},v_{i+1})$ if $(q_i,v_i) \xrightarrow{\beta^*} \xrightarrow{\beta^*} (q'_i,v'_i) \xrightarrow{\sigma_i} (q''_i,v''_i) \xrightarrow{\beta^*} (q_{i+1},v_{i+1})$.

For example of this model you can refer to Model 1 in [8].

Definition 3. (Composition of abstract models). Let M_i = $(\mathbf{A_i},\mathbf{Q_i},\mathbf{X_i},\mathbf{P_i},\rightarrow_\mathbf{i})$, $1 \leq i \leq \leq n$, be a set of abstract models. We assume that their sets of action and clocks are disjoint, i.e. for all $i \neq j$we have $\mathbf{A_i} \cap \mathbf{A_j}=\emptyset$ and $\mathbf{X_i} \cap \mathbf{X_j}=\emptyset$. A set of interactions γ is a subset of 2^A, where $A = \bigcup_{i=1}^{n} A_i$, such that any interaction $a \in\gamma$ contains at most one action of each component M_i, that is, $a = \{a_i | i \in I\}$ where $a_{i\in}A_i$and $I \subseteq \{1,2,\dots,n\}$. The composition of the abstract models M_i, $1 \leq i \leq n$, by using a set of interactions γ, denoted by $\gamma(M_1,\dots,M_n)$, is the composite abstract model $M=(\gamma,\mathbf{Q},\mathbf{X},\mathbf{P},\rightarrow)$ such that $\mathbf{Q}=\mathbf{Q_1}\times \times\mathbf{Q_2}\times\dots\ \mathbf{Q_n}$, $X = \bigcup_{i=1}^{n} X_i$ and $\rightarrow_\gamma$ is defined by the rules:

$$a = \{a_i\}_{i\in I} \in \gamma$$
$$g = \wedge_{i\in I}\, g_i$$
$$r = \bigcup_{i\in I} r_i$$
$$\forall i \in I, q_i \xrightarrow{a_i,g_i,r_i,p_i} q'_i$$
$$\forall i \notin I. q'_i = q_i$$
$$(q_1,\dots,q_n) \xrightarrow{(a,g,r,p)_\gamma} (q'_1,\dots,q'_n)$$
$$\text{also } \exists i \in \{1,\dots,n\}. q_i \xrightarrow{\beta,g_i,r_i,p_i}_i q'_i$$
$$\forall i \neq j. q'_j = q_j$$
$$(q_1,\dots,q_n) \xrightarrow{\beta,g_i,r_i,p_i}_\gamma (q'_1,\dots,q'_n).$$

A composition $M=\gamma(M_1,\dots,M_n)$ of abstract models M_i, $1 \leq i \leq n$, can execute two types of transitions: interactions $a = \{a_i\}_{i\in I} \in \gamma$ which corresponds to synchronizations of actions a_i of models M_i, $i \in I$, and internal actions β of the modeles M_i. An interaction $a = \{a_i\}_{i\in I} \in \gamma$is enabled from a state of M if all actions a_i are enabled.

In a composite model $M=\gamma(M_1,\dots,M_n)$ many interaction can be enabled to act simultaneously (in the same time) introducing a degree of non-determinism in the behavior of M.

In order to restrict non-determinism, *priorities* are introduced that specify which interaction should be executed among the enabled ones. A priority on $M=\gamma(M_1,\dots,M_n)$ is a relation $\pi \subseteq \gamma \times Q \times \gamma$ such that for all q the relation $\pi_q = \{(a,a') | (a,q,a') \in \pi\}$ is a partial order. We write $a\pi_q a'$ for $(a,q,a') \in \pi$ to express the fact that a has weaker priority than a' at state q. That is if both a and a' are enabled at state q, only the action a' can be executed. Thus, priority $a\pi_q a'$ is applied only when the conjunction of the guards and resources of a and a' is true. Let $q \xrightarrow{a,g,r,p}_\gamma q'$ and $q \xrightarrow{a',g',r',p'}_\gamma q''$ be transitions of M such that $g=[c]^\tau$ and $g'=[c']^{\tau'}$. Applying priority $a\pi_q a'$ boils down to transforming the guard g of a into the guard $g_\pi=[c\wedge\neg c']^\tau$ and leaving the guard g' of a' unchanged.

Furthermore we denote by $en_q(a)$ the predicate characterizing valuations of clocks for which an interaction a is enabled at state q. It is defined by:

$$en_q(a) = \begin{cases} false, if\ \nexists(q,a,g,r,p,g') \in\rightarrow_\gamma \\ \bigvee_{(q,a,[c]^\tau,r,p,q'\in\rightarrow_\gamma} \quad c - otherwise. \end{cases}$$

Definition 4. Priority. Given a composite model $M=(\gamma,\mathbf{Q},\mathbf{X},\mathbf{P},\rightarrow_\gamma)$ the application of priority π to M defines a new model $\pi M=(\gamma,\mathbf{Q},\mathbf{X},\mathbf{P},\rightarrow_\pi)$ such that $\rightarrow_\pi$ is defined by the rule:

$$q \xrightarrow{a,g,r,p}_\gamma q', g = [c]^\tau,$$

$$g_\pi = \left[c \wedge \neg \bigvee_{a \pi_q a'} en_q(a') \right]^\tau$$

$$q \xrightarrow{a, g_\pi, r, p}_\pi q'.$$

Example of an abstract model with priorities is considered in [8].

Abstract models are platform-independent representations of programs with atomic and instantaneous actions execution. Real ("physical") models represent the program behavior on a real platform. It accounts the fact that the action execution takes some non-zero time. So we need to break action execution atomicity and introduce execution times. The transition of an action a of an abstract model is replaced by a sequence of two consecutive transitions of the corresponding physical (real world) model – see figure 1. The first transition marks the beginning of the execution of action a, and the second transition marks its completion. These transitions are separated by a partial state denoted by ⊥. The execution time of the action corresponds to the waiting time at state ⊥

$$q \xrightarrow{a,g,r,p} q' \ \overset{\perp_t}{\rightarrow} \ \mathbf{q} \xrightarrow{a,g,r,p} \perp_t \overset{\beta}{\rightarrow} q'$$

(Corresponding sequence of transitions in $M^\perp$), where

$$t = (q, a, g, r, p, q') in\, M.$$

This denoted the transformation of transitions of the abstract model.

Definiton 5. Physical model. Let M=(**A**,**Q**,**X**,**P**,→) be an abstract model. We define the associated as the timed automaton $M^\perp$=(**A**,**Q** ∪ **Q**$^\perp$,**X**,**P**,$\rightarrow_\perp$) such that:

Q$^\perp$is the set of partial states such that there is one partial state for each transition of M, that is, **Q**$^\perp$={$\perp_t$ $|t$ ∈→}

$\rightarrow_\perp$is defined by the rule:

$$\frac{q \xrightarrow{a,g,r,p} q' \quad t=(q,a,g,r,p,q')}{q \xrightarrow{a,g,r,p}_\perp \perp_t \quad \perp_t \xrightarrow{\beta,[true]^I,\emptyset,true}_\perp q'}.$$

In the physical model $M^\perp$we assume arbitrary execution times for actions, ranging from 0 to +∞, which is modeled by the guard [true]^I for β-transitions. Notice that $M^\perp$can be further constrained if bounds of the execution times of actions are unknown. For instance, if an estimate *WCET*(a) is known for the worst-case execution timeof an action a, the associated timing constraint is [x_a≤*WCET*(a)]^d instead of [true]^I, where x_a is a clock that is reset whenever a is started. This allows us to statically check the correctness of the application running on the platform but this is beyond the paper scope.

In a physical model $M^\perp$, the execution of the action a by a transition t=(q,a,g,r,p,q') is followed by a lapse of time $\delta(a) \in \mathbb{N}$ at the partial state $\perp_t$before a β-transition is executed:

$$(q, v) \rightsquigarrow^a (\perp_t, v[r \longmapsto 0]) \rightsquigarrow^{\delta(a)}$$
$$\rightsquigarrow^{\delta(a)} (q', v[r \longmapsto 0] + \delta(a)). \qquad (1)$$

This corresponds to the following execution sequence in the abstract model M, if such a sequence is feasible:

$$(q, v) \rightsquigarrow^a (q', v[r \longmapsto 0]) \rightsquigarrow^{\delta(a)}$$
$$\rightsquigarrow^{\delta(a)} (q', v[r \longmapsto 0] + \delta(a)). \qquad (2)$$

It should be noticed that the time stamp $\delta(a)$ of $M^\perp$ in (1) may not be the time stamp of M in (2) if δ(a)>*wait*(q',v[$r \longmapsto 0$]), meaning that the physical model violates timing constraints defined in the corresponding abstract model. In this case we say that the considered execution sequence is not *time-safe*. (The execution times of abstract and physical models are compared in [9] – considering that if all execution sequences of $M^\perp$ are time-safe than $M^\perp$ is weakly simulated by M).

Correct model implementation should execute only time-safe sequences, but time-safety violations occur in a physical model when the execution time of an action is larger than what is allowed by the timing and resource constraints of the corresponding abstract model. Correct implementations are obtained for platforms that are sufficiently fast for executing the program without violating time-safety. Here the physical model preserves the semantic of the abstract model as shown in [10]. Otherwise the time-safety violations should be checked in run time.

Physical model composition considerations and correctness considerations can be checked in [8].

So how we can deal with resource limitations? State-of-the-art software packages require simpler solutions, so model interpretation in run-time (such as in [11]) looks complex and superfluous. In case if program is already expressed as a graph of scheduled blocks, it is possible to evaluate model requirements and behavior at the moments between the previous block finish and a the new block start – this also works for moments of spawning new parallel processes and joining parallel processes for one new serial process. This works well for cases enlisted before: LAPACK-based computations [12], neural network inference and media processing in Gstreamer-like pipelines. Even it is possible for interpreted code, such as Java or .Net, where code assemblies can be annotated with resource information.

4. Affected industry cases

Why all these model descriptions are important for us although all they looks to be abstract for real software business?

Before 2005 the parallel programming was the area of academy pundits and small groups of professionals in computer graphics area. After 2005 the integrated circuits making technology enabled so many transistors on single die for hardware engineers so that the single computing units (up to von Neumann definitions) can not effectively use the hardware. The most efficient way was to place several processing units on one silicone crystal, so that the silicone can run multiple processes simultaneously somehow. It looked like "as single CPU can not deal effectively, let's do multiple CPUs and the programmer should do software in right way". At practical side a lot of money was invested into teaching parallel programming in colleges and making specialized computer languages for highly parallel hardware – such as Nvidia CUDA [13] or anti-Nvidia industry standard OpenCL [14]. As the efficient parallel software requires more time to invest, more skilled resources and more money for test teams work the new tendency was appeared – to make frameworks which allow the programmer to make parallel programs with simpler (less or more simpler) descriptions. Such concept looks nice but works hard. Two well-known technology examples – CUDA and OpenCL shows that the main model of writing a parallel program for GPUs is a "producer-consumer" model when a CPU-side program controls GPU threads execution and none of GPU threads are self-sufficient. Practically all memory management employs CPU-GPU memory communications, and memory communications between neighbor GPU cards are exceptions.

Under these conditions any software developer experiences a serious technology limitation, as any parallel program support only its exclusive execution on GPU resource and the resource pool subdivision between different processes are possible only in case if each parallel program allocates some number of GPUs, leaving the other GPUs to counterpart.

However, all technologies similar to CUDA/OpenCL or any specialized parallel programming languages do not decrease the development time significantly. The good ways to decrease programmer effort are infrastructures, which shorten development time for 90 % in simpler cases and for 50 % in hard cases. The important issue is that the most of the infrastructures are upgradable to incorporate the described physical model. Let us check several infrastructure (framework) cases.

1) TensorFlow [3]. At least 60 % of neural network learning and inference market. Tensorflow interprets network model (using Python language) based on a sequence of big code blocks.

2) Caffee[1]. Near 30 % of neural networking segment, written in optimized C++. Caffee (conceptually similar to Tensorflow) interprets an abstract sequence of blocks.

3) Gstreamer [5] and ffmpeg [15] media processing pipelines. Both construct a pipeline using already defined big code blocks (plugins), including spawning parallel processes and internal queue storage. Pipelines also works for GPU-based computations. Both packages are widely used as a base for industry media processing and broadcasting.

4) OpenCV framework [4]. It is a good base for parallel matrix computations

(and 2D/3D processing) at GPU. Still the OpenCV is more low-level library tool than a framework, the packages can benefit from model interpretation.

5) Simulation tools, starting from the old good NS2 (good use example in [16]) and other network simulations. In case of big runs cluster-based modeling benefit from model interpretations.

The proposed abstract and physical models are good universal tools for many frameworks. It can deal with both low level descriptions of CPU-handled subroutines (at 50-100 CPU instructions level) and high-level annotations for framework elements: all timing and resource constraints remain the same.

The next step will be more practical: incorporation of the model interpreter into one of the frameworks and practical test of benefits got because of expressing the software in term of physical model.

Conclusions

In order to meet the modern requirements for software development – less time, more quality, lower expenses – this article proposes the resource constrained model for parallel programs, which allows to run (or model the behavior) of multiple (and different) parallel software runs under resource constraints on real-world hardware systems. In addition, the list of popular frameworks - which can benefit from incorporating the elements of the resource-constrained models interpreter – is presented. The future work includes the extension of one of framework with model interpreter for low-overhead resource checker on the fly (at program run time) and real-world model examples.

References

1. Yangqing J., Shelhamer E., Donahue J., Karayev S., Long J., Girshick R., Guadarrama S., Darrell T. (2014) Caffe: Convolutional Architecture for Fase Feature Embedding. ArXiv preprint: arXiv:1408.5039
2. Redmon J. (2013) [Online]. Darknet: Open Source Neural Networks in C. – Available from https://pjreddie.com/darknet/
3. Abadi M., Barham P., Chen J., Chen Z., Davis A., Dean J., Devin M., Ghemawat S., Irving G., Isard M. & others (2016). TensorFlow: A System for Large-Scale Machine Learning. *OSDI*. P. 265–283.
4. Bradsky G., Kaehler A. Learning OpenCV — O'Reilly, 2008. P. 1.
5. Taymans W., Baker S., Wingo A. (2018) GStreamer Application Development 1.10.1. P. 164. 12th Media Services.
6. DeepStream [Online] – Nvidia DeepStream Software Development Kit – Available at https://developer.nvidia.com/deepstream-sdk
7. Peter Hui and Satish Chikkagoudar. (2012) A Formal Model for Real-time Parallel computation. In Proc of FTSCS-2012. P. 39–53.
8. Ahlem Triki, Jacques Combaz. (2013) Model-Based implementation of Parallel Real-Time Systems. Verimag Research Report TR-2013-11
9. Wilhelm R., Altmeyer S., Burguiere C., Grund D., Herter J., Reineke J., Wachter B., Wilhelm S. Static timing analysis for hard real-time systems. In Barthe G. and Hermenegildo M.V., eds., *WMCAI*. 2010. Vol. 5944 of LNCS. P. 3–22. Springer.
10. Abdellatif T., Combaz J., Sifakis J. Model-based implementation of real-time applications. In Carloni L.P. and Stavros Tripakis, eds. *EMSOFT*. 2010. P. 229–238.
11. Basu A., Bogza M., Sifakis J. Modeling heterogeneous real-time components in BIP. In *SEFM*. 2006. P. 3–12. IEEE Computer Society.
12. Baboulin M., Demmel J., Dongarra J., Tomov S., and Volkov V. Enhancing the Performance of Dense Linear Algebra Solvers on GPUs (in the MAGMA Project) , Austin, TX, The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC08), Nov. 2008.
13. CUDA [Online] – Available at https://developer.nvidia.com/cuda-zone
14. OpenCL [Online] – Available at Khronos Group: https://www.khronos.org/opencl/
15. Xu Y.G. and Cao S.X. Real-Time Video Acquisition and Frame Compression Processing Technology Based on FFmpeg, Applied Mechanics and Materials. 2014. Vols. 631–632. P. 494–497.
16. Michael Welzl. Adaptive Multimedia Communication over Satellite Routed IP". In ICC 2000 (International Conference on Communications – IEEE Communications Society), New Orleans, Louisiana, USA, 18–22 June 2000.

Література

1. Yangqing J., Shelhamer E., Donahue J., Karayev S., Long J., Girshick R., Guadarrama S., Darrell T. (2014) Caffe: Convolutional Architecture for Fase Feature Embedding. ArXiv preprint: arXiv:1408.5039
2. Redmon J. (2013) [Online]. Darknet: Open Source Neural Networks in C. – Available from https://pjreddie.com/darknet/
3. Abadi M., Barham P., Chen J., Chen Z., Davis A., Dean J., Devin M., Ghemawat S., Irving G., Isard M. & others (2016). TensorFlow: A System for Large-Scale Machine Learning. *OSDI*. P. 265–283.
4. Bradsky G., Kaehler A. Learning OpenCV — O'Reilly, 2008. P. 1.
5. Taymans W., Baker S., Wingo A. (2018) GStreamer Application Development 1.10.1. P. 164. 12th Media Services.
6. DeepStream [Online] – Nvidia DeepStream Software Development Kit – Available at https://developer.nvidia.com/deepstream-sdk
7. Peter Hui and Satish Chikkagoudar. (2012) A Formal Model for Real-time Parallel computation. In Proc of FTSCS-2012. P. 39–53.
8. Ahlem Triki, Jacques Combaz. (2013) Model-Based implementation of Parallel Real-Time Systems. Verimag Research Report TR-2013-11
9. Wilhelm R., Altmeyer S., Burguiere C., Grund D., Herter J., Reineke J., Wachter B., Wilhelm S. Static timing analysis for hard real-time systems. In Barthe G. and Hermenegildo M.V., eds., *WMCAI*. 2010. Vol. 5944 of LNCS. P. 3–22. Springer.
10. Abdellatif T., Combaz J., Sifakis J. Model-based implementation of real-time applications. In Carloni L.P. and Stavros Tripakis, eds. *EMSOFT*. 2010. P. 229–238.
11. Basu A., Bogza M., Sifakis J. Modeling heterogeneous real-time components in BIP. In *SEFM*. 2006. P. 3–12. IEEE Computer Society.
12. Baboulin M., Demmel J., Dongarra J., Tomov S., and Volkov V. Enhancing the Performance of Dense Linear Algebra Solvers on GPUs (in the MAGMA Project) , Austin, TX, The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC08), Nov. 2008.
13. CUDA [Online] – Available at https://developer.nvidia.com/cuda-zone
14. OpenCL [Online] – Available at Khronos Group: https://www.khronos.org/opencl/
15. Xu Y.G. and Cao S.X. Real-Time Video Acquisition and Frame Compression Processing Technology Based on FFmpeg, Applied Mechanics and Materials. 2014. Vols. 631–632. P. 494–497.
16. Michael Welzl. Adaptive Multimedia Communication over Satellite Routed IP". In ICC 2000 (International Conference on Communications – IEEE Communications Society), New Orleans, Louisiana, USA, 18–22 June 2000.

Received 01.10.2019

***About the author*:**

Dmytro V. Rahozin,
candidate of tech. sciences (PhD)
More than 10 publication in Ukrainian and foreign journals.
https://orcid.org/0000-0002-8445-9921

Affiliation:

Institute of Software Systems,
NAS of Ukraine
03187, Kyiv-187,
Acad. Hlushkov avenue, 40.
Tel.: +38 068 575 91 25.
E-mail: dmytro.rahozin@gmail.com