Online ISSN: 2367-5357
DOI: 10.7546/itc-2019-0014

GPU-In-The-Loop Simulation of Linear Controllers

J. Kralev

Key Words: *GPU-in-the-loop; computation parallelization; state-space controller, GPGPU.*

Abstract. *Many practical computational tasks require processing of big data sets. When calculation cannot be performed fast enough multithread parallelization is most obvious solution. However interdependence of intermediate results can limit the parallelization of the computational problem. In modern control theory a key mathematical model is the state-space representation. The aim of the present paper is to demonstrate how such a model can be implemented on a conventional graphical processing unit without disturbing the response of the respective closed-loop system where the model is participating.*

1. Introduction

Simulations of complex dynamical systems require distributed computational architecture. There are many approaches for distributed computing aiming to extend the capabilities that a single central processing unit (CPU) can offer. Some examples are pear-to-pear (P2P), cluster, grid, cloud or jungle computing. These methods can be implemented over multicore, network or custom FPGA architectures. An easy accessible and relatively cheap multicore architecture is the graphical processing unit (GPU), which is a hardware platform dedicated to processing of 3D video primitives in order to produce realistic video output in real time. GPU is a single instruction multiple data (SIMD) architecture, which allows a single computational algorithm to be executed on several distinct data sets simultaneously (*figure 1*).

Modern GPU produced by nVidia and AMD hosts several hundreds of processing threads. These resources are programmable to a certain level with the help of specialized shader languages primary designed to allow variety of visual effects, such as reflection of light by a soap bubble. Programmable shading is used to achieve cinematic level realism which is the ultimate goal in computer graphics. The parallel array of processors of modern GPU is programmable in C language with the help of application programming interface (API) for graphics. The aim of graphics system is to produce an image from a geometrical description of a scene with fast enough refresh rate. The GPU processes geometric data through a hardware pipeline. The scene is a collection of geometric primitives, lights, object materials, viewpoints with position and orientation.

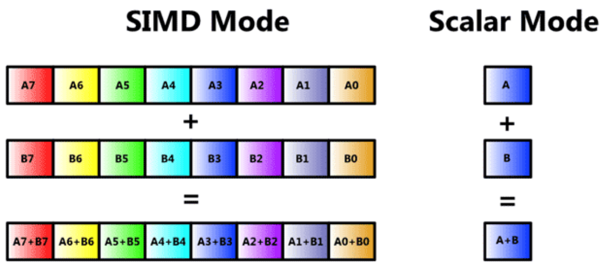


Figure 1. Single instruction multiple data (SIMD) vs. scalar architectures. Courtesy of Intel® Inc.

Recently GPUs begin to find application also as a general-purpose parallel computation engines or GPGPU [4]. They allow extremely high arithmetic throughput and streaming memory bandwidth but with considerable latency between individual computations since final images are only displayed as fast as 16 ms. CPUs are optimized for low latency and GPUs are optimized for high throughput. A single kernel unit from the GPU array achieves a sustained 330 billion floating-point operations per second (Gflops). The GPU's specialized architecture is not well suited to every algorithm [3]. Many applications are inherently serial and are characterized by incoherent and unpredictable memory access. Nonetheless, many important problems require significant computational power.

The article proposes a certain technique to access GPGPU capabilities of a desktop platform by Simulink® environment. Standard Simulink® library does not support bocks for GPU-in-the-loop processing. The present work aims to run a particular state-space realization on GPU by utilizing Microsoft Accelerated Massive Parallelism (AMP) [2, 8]. AMP is C++ wrapper library which goal is to accelerate execution of code by exploiting data-parallel hardware. This library introduces specialized operators for multi-dimensional algorithms. The AMP programming model includes also multidimensional arrays, indexing, memory transfer, tiling, mathematical function library and a control of data transfers between CPU and GPU.

Mathematical models for many physical processes as well as control theory which design the controllers for them are extremely dependent on the ability to solve systems of ordinary differential equations [1, 5]. Every linear time invariant (LTI) system can be represented using auxiliary state variables as

$$\begin{aligned} x(k+1) &= Ax(k)+Bu(k) \\ y(k) &= Cx(k)+Du(k) \end{aligned}, \tag{1}$$

where x is a N-dimensional vector of state-variables, u is N_u -dimensional vector of input variables, y is N_y -dimensional vector of output variables, A, B, C, D are matrices with appropriate dimensions, k is the current time instant and $k + 1$ is the consequent time instant. In order to calculate the expression for one-step the computer must execute $N^2 + NN_u + NN_y + N_uN_y$ multiplications and $N(N+N_u-1)+N_y(N+N_u-1)$ summations. Obviously if these operations are executed in parallel they will be completed faster than a serial execution. The paper is organized as follows. Section II present the Simulink® block design, Section III is about related AMP implementation, Section IV present a practical example for which developed block can be used which concludes with the experimental results.

2. Simulink® block design

The interface between GPU and Simulink model is based on a developed S-function block by the author to support simulation of LTI systems represented in state-space form (*figure 2*). During normal simulation the GPU block behaves only as a placeholder in the diagram by sourcing and sinking just a dummy signals. However after code generation for general real-time (GRT) target the GPU block is replaced with its actual AMP implementation in C++ language. In this case the simulation is running in External simulation mode when the generated code is executed as a standalone application and the Simulink block diagram works just as a frontend presentation of the application runtime.

2.1. Code generation from block diagram

Simulink Coder (formerly Real-Time Workshop) generates C and C++ code from Simulink diagrams, Stateflow charts, and MATLAB functions. The generated source code can be used for real-time and non real-time applications such as simulation acceleration, rapid prototyping, hardware-in-the-loop simulation, embedded algorithm design, etc.

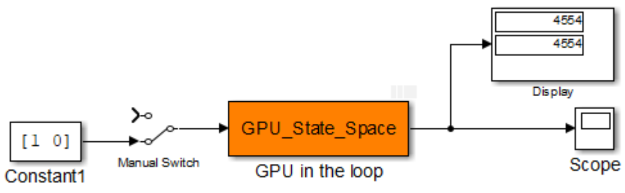


Figure 2. Block diagram with the developed GPU in the loop block for simulation of LTI dynamical systems.

The Simulink model can be translated into suitable description for the target hardware platform. This process generates a sequence of intermediate representations of the primary Simulink functional diagram. A key transition in this process is the conversion from block diagram to C language. Consequently, the generated code is integrated with additional device driver libraries and compiled into a standalone executable. Also the generated code can be integrated with target specific routines to create a module for some operating system. The process finishes with compilation and loading into target platform. These transformations of the original functional description are required because Simulink model is more abstract form of representation than target hardware platform can understand. Code generation is controlled through template-like configuration files. *Figure 3* summarizes the stages of the code generation process.

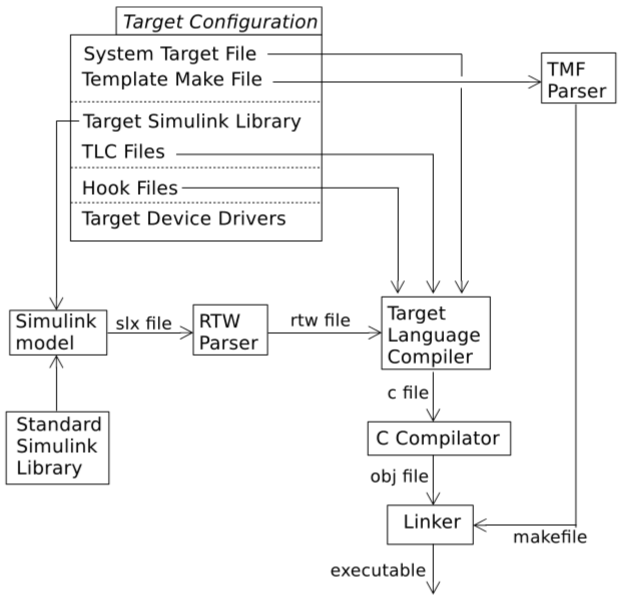


Figure 3. Generation of executable code from Simulink block diagram

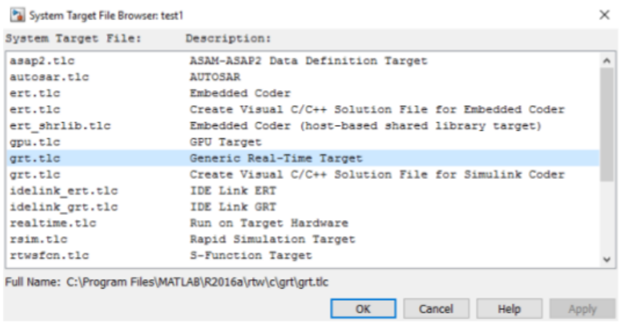
Target Language Compiler (TLC) is an intermediate script language representing the translation from Simulink graphical blocks to the target C code. Therefore each block from the diagram needs a corresponding TLC file. When S-function block is used the designer has to specify also its TLC description in order to inline the block into target. A dedicated RTW file (rtw for real-time workshop) contains the representation of the Simulink model as a hierarchical structure describing also the interconnection between blocks and their parameters. This is the form which TLC parser can read in order to produce C programs. The parser has an access to each block input or output signal. This information is necessary to properly reproduce block function in C language.

The main TLC file for the target is the system target file (STF) which is parsed when build command for the model is invoked. It in turn invokes the block specific TLC files. The STF also defines some target specific interface for user interaction through Model Configuration Setting dialog. For example one can turn on or turn off a particular software future in the generated code. After the code is generated the process of compilation and linking is controlled by a template make file (TMF) also specific to the selected target. The produced makefile from it is used by a dedicated make tool which in turn invokes the target toolchain.

2.2. Design of GPU_State_Space block

The block named GPU_State_Space from *figure 2* is a S-function block which is described by four files: *GPU_State_Space.c* is S-function definition for normal simulation mode, *GPU_State_Space.tlc* is TLC wrapper which works as a template during the C code generation for the block, *GPU_Calc.cpp* is the actual C++ AMP code for SIMD processing of state-space realization, *GPU_Calc.h* is the corresponding header file containing declarations to be included in generated C code.

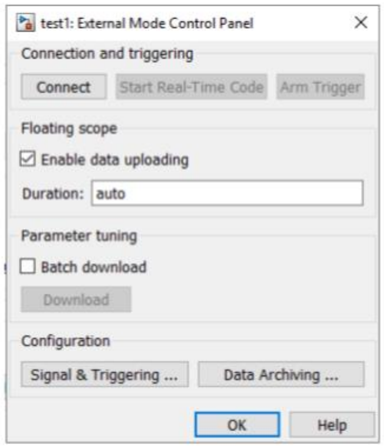
When some model blocks are CPU implemented and others are GPU implemented it is appropriate to use External simulation mode as opposite to Normal simulation mode. In this mode the model is translated to C code and then compiled to selected target which in the case for GPU_state_space block would be generic real-time (GRT) target. *Figure 4* shows the target selection dialog box, which allows for “exporting” the Simulink model to various platforms. New targets can be created as well by developing the respective STL and TMF templates. This target platform used in this article is actually the host PC where the user develops the model. Then the GRT compiled model will be run as a standard console application under MS Windows OS and the source Simulink model serves only as a graphical user interface for data recording and parameter tuning. External mode simulation is started and controlled from dedicated dialog box presented on *figure 5*. This dialog makes the connection between the block diagram and the target and controls the data acquisition options.


System Target File Browser: test1

System Target File:	Description:
asap2.tlc	ASAM-ASAP2 Data Definition Target
autosar.tlc	AUTOSAR
ert.tlc	Embedded Coder
ert.tlc	Create Visual C/C++ Solution File for Embedded Coder
ert_shrlib.tlc	Embedded Coder (host-based shared library target)
gpu.tlc	GPU Target
grt.tlc	Generic Real-Time Target
grt.tlc	Create Visual C/C++ Solution File for Simulink Coder
idelink_ert.tlc	IDE Link ERT
idelink_grt.tlc	IDE Link GRT
realtime.tlc	Run on Target Hardware
rsim.tlc	Rapid Simulation Target
rtwsfcn.tlc	S-Function Target

Full Name: C:\Program Files\MATLAB\R2016a\rtw\c\grt\grt.tlc

Figure 4. Target selection dialog box


test1: External Mode Control Panel

Connection and triggering

Floating scope

☑ Enable data uploading

Duration: auto

Parameter tuning

☐ Batch download

Configuration

Figure 5. External mode control panel

In normal simulation mode the GPU block from *figure 2* is only a placeholder. It has defined input and output ports with appropriate types to allow integration with the data from the rest block in the model. The block is introduced in the model a as S-function GPU_State_Space which is a C program compiled as a MATLAB executable (MEX) binary. *Figure 6* contains the description of the block for simulation mode. This S-function defines only initialization routine about port data types and port size. The input port is defined as a vector signal with N_u elements and the output port is a vector N_y elements which are single precision floating-point numbers. The S-function is compiled with command mex GPU_State_Space.c which generates the GPU_State_Space.*mex* file.

The TLC file for the GPU block is summarized in *figure 7*. TLC syntax is well documented in Mathworks documents. *Figure 7* defines the target implementation of the block with respect to two functions. BlockTypeSetup function includes GPU_Calc.h which declares AMP dependent description of the block. The function Outputs is invoked once in every sample interval during model execution. There the parSS() function is called to calculate the model transition from current state to the next. The inputs and outputs of the AMP algorithm are implements as the array u[] and y[]. On *figure 7* the assigned TLC placeholders are related to their actual values. For example variable y[1] replaces %<y1> placeholder.

```
#define SFUNCTION NAME GPU_State_Space
#define SFUNCTION LEVEL 2

#include " simstruc . h"

static void mdlInitializeSizes (SimStruct *S)
{
...
ssSetNumContStates(S, 0);
ssSetNumDiscStates(S, 0);
if (! ssSetNumInputPorts(S, 1)) return;  // One input  port − port 0
ssSetInputPortWidth(S, 0, Nu); // Dimension of the signal on port 0
if (! ssSetNumOutputPorts(S, 1)) return;  // One output port − port 0
ssSetOutputPortWidth(S, 0, Ny); // Dimension of the signal on port 0
ssSetOutputPortDataType(S, 0, SS_SINGLE);
ssSetNumSampleTimes(S, 1);
ssSetNumRWork(S, 0);
ssSetNumIWork(S, 0); ssSetNumPWork(S, 0);
ssSetNumModes(S, 0); ssSetNumNonsampledZCs(S, 0);
...
}
static void mdlInitializeSampleTimes (SimStruct *S)
{
ssSetSampleTime(S, 0, INHERITED SAMPLE TIME);
ssSetOffsetTime (S, 0, 0.0);
}
```

Figure 6. S-function definition of the GPU in the loop block (GPU_State_Space.c file)

```
%implements GPU_State_Space "C"

%function BlockTypeSetup(block, system) void
  %<LibAddToCommonIncludes("GPU_Calc.h")>
%endfunction %% BlockTypeSetup

%function Outputs(block,system) Output
{
  %assign u1 = LibBlockInputSignal(0,"","",0)
  %assign u2 = LibBlockInputSignal(0,"","",1)
  %assign y1 = LibBlockOutputSignal(0,"","",0)
  %assign y2 = LibBlockOutputSignal(0,"","",1)

  u[0] = %<u1>; u[1] = %<u2>;

  parSS();

  %<y1> = y[0]; %<y2> = y[1];
}
%endfunction
```

Figure 7. TLC description of the GPU in the loop block required for code generation (GPU_State_Space.tlc file)

3. AMP Implementation of LTI system

The key definition in AMP is array class which represents an ordered set of data stored on the target video controller (GPU) opposite to storing it in CPU addressed memory. In AMP programming model multiple threads of execution share a common global memory located at GPU. Also a group of execution threads called a tile can access dedicated tile memory isolated from the global memory. Variables located at GPU belong to array class. Variables located at CPU address space but accessible from GPU belong to array_view class. The tile_static class represents a variable stored in tile memory. *Figure 8* presents the header part of the GPU_Calc.cpp file which includes the library amp.h and the concurrency namespace. There are filled the values of the system matrices *A*, *B*, *C*, *D* which are with predefined dimensions. Also vectors with input, output and state signals are defined.

```
#include "GPU Calc.h"
#include <amp.h>

using namespace concurrency;

#define MAX(X,Y) (((X) < (Y)) ? (Y) : (X))
#define NX 3
#define NU 2
#define NY 2
#define NUM (MAX((NX), MAX((NU), (NY))))
float Amat[NX*NX] = {…};
float Bmat[NX*NU] = {…};
float Cmat[NX*NY] = {…};
float Dmat[NY*NU] = {…};
float u[NU], y[NY], x[NX];
```

Figure 8. State space realization on GPU with C++ AMP library (GPU_Calc.cpp)

The definition of the function parSS() which is called from the TLC wrapper is presented on *figure 9*. Since system *A*, *B*, *C*, *D* are stored in CPU memory a corresponding array views have to be defined. The program instructs parallel execution of multiple threads with parallel_for_each operator from AMP library which defines multiple threads of execution over so called extent of indexes. The parallel for each operator defines the program that will be executed over the data. Its content is given in *figure 10*.

```
void parSS() {
    array_view<float,2> Amat_gpu_view(NX,NX,Amat);
    array_view<float,2>  Bmat_gpu_view(NX,NU,Bmat);
    array_view<float,2> Cmat_gpu_view(NY,NX,Cmat);
    array_view<float,2> Dmat gpu view(NY,NU,Dmat);
    array<float,1> u_mem(NU), x_mem(NX), y_mem(NY);
    array<float,1> x_new_mem(NX);
    array_view<float,1>u_view(NU,u),x_view(NX,x),y_view(NY,y);
    copy(x view,x mem); copy(u view,u mem);

    extent <2> ex(NX,NX);
    parallel_for_each (ex.tile <1,NUM>(),...);

    copy(x new mem,x view); copy(y mem,y view);
}
```

Figure 9. Setting up parallel_for_each operator (GPU_Calc.cpp)

```
[=,&u_mem,&_ mem,&y_mem,&x new_mem]
(tiled_index<1,NUM> idx) restrict(amp) {
    tile_static float buf[NUM];
    tile_static float buf1[NUM];
    if (idx.local [1] < NX) {
        buf[idx.local[1]] = Amat_gpu_view(idx.tile[0], idx.local[1])∗
                            x_mem(idx.local[1]);
        buf1[idx.local[1]] = Cmat_gpu_view(idx.tile[0], idx.loca[1])∗
                            x_mem(idx.local[1]);
    }
    if (idx.local[1] < NU) {
        buf[idx.local[1]] += Bmat_gpu_view(idx.tile[0], idx.local[1])∗
                              u_mem(idx.local[1]);
        buf1[idx.local[1]]+= Dmat_gpu_view(idx.tile[0], idx.local [1])∗
                              u_mem(idx.local_[1]);
    }
    idx.barrier.wait();
    if  (idx.local[1] == 0) {
        int k;
        for (k=0;k<NUM;k++) {
            if (idx.tile[0] < NX)  x_new_mem(idx.tile[0]) += buf[k ];
            if (idx.tile[0] < NY)  y_mem(idx.tile[0]) += buf1[k ];
    }
}
```

Figure 10. State space realization on GPU with C++ AMP library

To parallelize the algorithm (1) the discrete state space equations are represented as a two dimensional calculation problem with the help of row index *i* and column index *j*,

$$x_{k+1,i} = \gamma_i \sum_{j=1}^{N} \alpha_j a_{ij} x_{k,j} + \beta_j b_{ij} u_{k,j} = \gamma_i \sum_{j=1}^{N} p_{ij} \;, \tag{2}$$

$$y_{k,i} = \delta_i \sum_{j=1}^{N} \alpha_j c_{ij} x_{k,j} + \beta_j d_{ij} u_{k,j} = \delta_i \sum_{j=1}^{N} q_{ij} \;, \tag{3}$$

where $N = \max(N_x, N_y, N_u)$. The selection coefficients $\alpha_j, \beta_j, \gamma_j$ and δ_j are defined with

$$\alpha_j = \gamma_j = \begin{cases} 1, \; i < N_x \\ 0, \; i < N_x \end{cases}, \; \beta_j = \begin{cases} 1, \; i < N_u \\ 0, \; i < N_u \end{cases}, \quad \delta_j = \begin{cases} 1, \; i < N_y \\ 0, \; i < N_y \end{cases}. \tag{4}$$

To be able to calculate $x_i(k+1)$ and $y_i(k)$ by summation over j one have to finish with multiplication within each term of the sum. So the form of representation allow only for N threads of parallelization. To reach N^2 parallel threads there have to be synchronization between calculation of (p_{ij}, q_{ij}) terms and $(x_i, y_i) \times N$ terms. C++ AMP supports tile indexing so $N \times N$ threads of execution are divided in N rows of $1 \times N$ tiles (*figure 9*).

The first argument of *parallel_for_each* defines the tiling over the extent *ex* of indexes for the problem. The second argument is a pointer to a function which defines calculation algorithm for each index pair (i, j) (*figure 10*). The operator *idx.barrier.wait*() causes current thread from a tile to be blocked until the rest of the threads from the same $1 \times N$ tile reach that operator. Therefore summation of terms (p_{ij}, q_{ij}) which is after the barrier function won't start before all of the multiplications are updated. This final summation which calculates the values of the next state and the output are executed serially by the leading tread from the $1 \times N$ tile which is identified by the condition *idx.local*[1] == 0. The variable *idx* is an index for the tile threads.

4. Practical example

The following third order state space model in discrete time will be used in the paper as an example system:

$$A = \begin{pmatrix} -0.0357 & 0.4289 & 0.3992 \\ -0.0587 & -0.2242 & 0.4242 \\ -0.0718 & -0.0702 & 0.2281 \end{pmatrix}, \tag{5}$$

$$B = \begin{pmatrix} 0.057 & -1.0278 \\ -0.6234 & 0.4744 \\ 0.2641 & -0.6005 \end{pmatrix}, \tag{6}$$

$$C = \begin{pmatrix} 0.063 & 0.235 & 1.6688 \\ -1.1143 & -0.06 & -0.3196 \end{pmatrix}, \tag{7}$$

$$D = \begin{pmatrix} -2.1295 & 0.5342 \\ -1.0193 & 1.6304 \end{pmatrix}. \tag{8}$$

4.1. Controller Design

The purpose of the controller $K \in R^{N_u \times N_x}$ is to reduce the system sensitivity to external disturbances by making the input signal $u(k)$ linearly dependent on the state estimate $\hat{x}(k)$. The actual state $x(k)$ of the system cannot be directly measurable so linear observer algorithm calculates a convergent estimate $\hat{x}(k) \to x(k)$ (when $k \to \infty$) of the state which can be used by the controller [6, 7]. The desired locations of the poles which are also eigenvalues of the matrix A are set to $\lambda_1 = e^{-T_S} = 0.37$, $\lambda_2 = e^{-2T_S} = 0.14$, $\lambda_3 = e^{-3T_S} = 0.05$, where $T_S = 1$s is the sampling time. After application of the controller *K*, the dynamics of the closed-loop system with the state feedback is determined by the eigenvalues of $A - BK$. Therefore by solving for the elements of the matrix K the equation $eig(A - BK) = diag(\lambda_1, \lambda_2, \lambda_3)$ we get

(9) $$K = \begin{pmatrix} 0.2863 & 0.448 & -1.1498 \\ 0.1944 & -0.2422 & -0.4938 \end{pmatrix}.$$

The state controller calculates control action according to $u = Kx$ but the current state x is not directly observable. Asymptotic estimate $\hat{x}$ of the state is produces from

(10) $$\hat{x}(k+1) = A\hat{x}(k) + Bu(k) + K_{obs}\left(y(k) - C\hat{x}(k)\right),$$

where the observer gain K_{obs} is calculated such that the integral

(11) $$J = \sum_k (\hat{x}-x)^{\mathrm{T}} Q(\hat{x}-x) + (\hat{y}-y)^{\mathrm{T}} R(\hat{y}-y)$$

is minimized. For $Q = I_3$ and $Q = 10^3 I_2$ the observer gain

(12) $$K_{obs} = \begin{pmatrix} 0.9204 & 0.6416 & 0.3609 \\ -0.1908 & -0.0401 & 0.034 \end{pmatrix} \times 10^{-3}.$$

The state feedback gain K regulates the properties of the transient but to achieve unit stationary gain between the reference r and the output signal y a scaling matrix L is calculated as $L = \left((\mathrm{C} - \mathrm{DK})(\mathrm{I}_3 - \mathrm{A} + \mathrm{BK})^{-1}\mathrm{B} + \mathrm{D}\right)^{-1}$ which gives the following result

(13) $$L = \begin{pmatrix} -0.3353 & 0.1058 \\ -0.1685 & 0.3035 \end{pmatrix}.$$

Figure 11 shows the locations of the system poles in the complex plane. After application of the state feedback gain $K \in R^{N_u \times N_x}$, the original locations of the system poles are translated to the prescribed locations (0.05, 0.14, 0.37) of the corrected system. Such change makes the system transients from oscillatory (left half plane pole) to aperiodic. However the locations of the transmission zeros is also changed which can be observed on *figure 12*. One of the zeros of the original system is outside the unit circle which indicates non minimal phase transients. The corresponding zero of the corrected system is also outside the circle but a lot closer. Hence the non minimal phase effects will be attenuated.

4.2. Simulink model

The model from *figure 13* contains two identical closed loop systems controlled with previously synthesized state feedback. Both systems are mathematically equivalent. The only difference is GPU implementation of the plant's model for the upper system. The aim of such simulation is to validate that both systems produce identical response. The controller is implemented as a matrix gain block. The observer Eq. (10) is inserted in the model with the help of Discrete State-Space block.

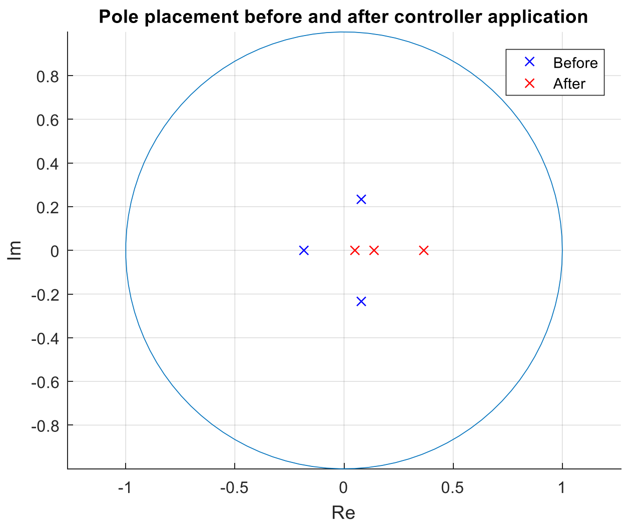


Figure 11. Locations of the closed-loop poles compared to open loop poles

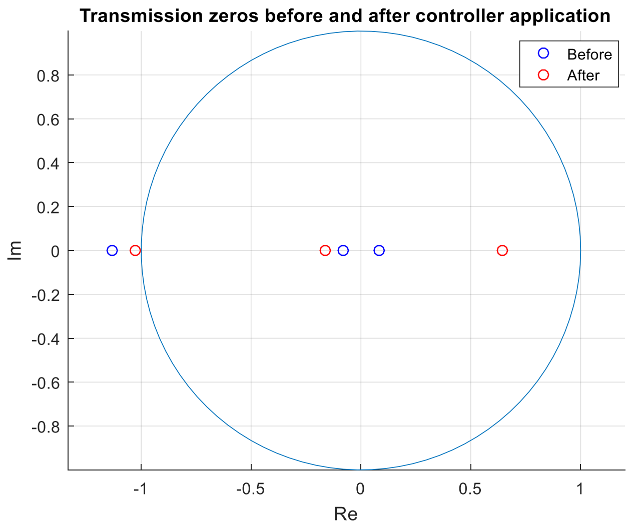


Figure 12. Locations of the closed-loop transmission zeros compared to open loop transmission zeros

Since the outputs of the upper and the lower closed-loop system are identical during simulation the only difference can be seen if calculate the residual $y_{Simulink} - y_{GPU}$ between the Simulink output $y_{Simulink}$ and the GU in the loop output y_{GPU}. This residual is presented on *figure 14* and is result from the rounding errors form floating point arithmetic. It is widely known that for the case of single precision floating point numbers such errors are from the order of 10^{-6}. The figure confirms that. However since the example system is with two outputs the actual plot is the 2-norm of the residual.

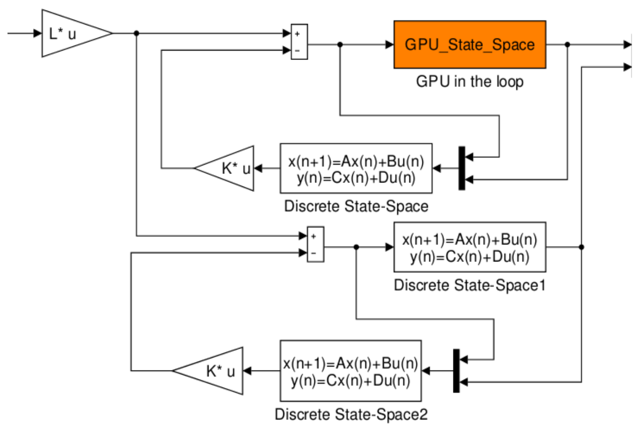


Figure 13. Locations of the closed-loop transmission zeros compared to open loop transmission zeros

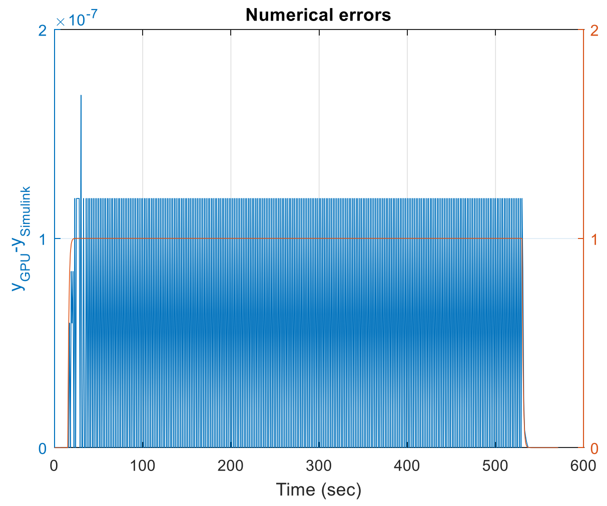


Figure 14. The difference between the outputs of the normally simulated closed loop system and GPU in the loop simulation overlayed with the step response

Conclusion

The article proposes a certain technique to access GPGPU capabilities of a desktop platform by Simulink® environment. Standard Simulink® library does not support bocks for GPU-in-the-loop processing. The target GPU was accessed by utilizing Microsoft AMP – a C++ wrapper library used to accelerate execution of code by exploiting data-parallelism.

The main result of the article is the design of a new Simulink block allowing for distributed computation of linear-state space models. The block is implemented as a parallel SIMD algorithm on a dedicated GPU device and can be applied in many practical contexts. The article examined a practical example for application of such block for the common task in control theory. The selected example was simple enough to illustrate the block usage with the focus on the technique presentation rather than complexity.

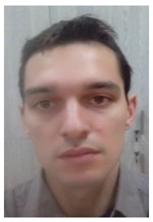
Acknowledgements

This work received seed funding from the National Science Fund of Bulgaria, Grant No KP-06-M27/10.

References

1. Bouvier, D., B. Cohen, W. Fry, et al. Kabini: An AMD Accelerated Processing Unit System on A Chip. – *IEEE Micro*, 34, 2014, No. 2, 22–33, doi: https://doi.org/10.1109/MM.2014.3
2. Gregory, K., A. Miller. C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++. Microsoft Press, 2012.
3. Jing, Y., W. Zeng, N. Wang, et al. GPU-based Parallel Group ICA for Functional Magnetic Resonance Data. – *Computer Methods and Programs in Biomedicine*, 119, 2015, No. 1, 9–16, https://doi.org/10.1016/j.cmpb.2015.02.002
4. Ko, Y., Y. Yi, J. Kim, et al. Fast GPU-in-the-loop Simulation Technique at OpenGL ES API Level for Android Graphics Applications. Proceeding of International Symposium on Rapid Sys Prototyping (RSP), IEEE, Amsterdam, 2015, doi: 10.1109/RSP.2015.7416546
5. Ohshima, S., K. Kise, et al. Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment, VECPAR 2006, 4395, 305–318, doi: 10.1007/978-3-540-71351-7_24
6. Puleva, T., G. Rouzhekov, T. Slavov, B. Rakov. Hardware in the Loop (HIL) Simulation of Wind Turbine Power control. IET Conference Publications, vol. 2016 (CP711), 2016.
7. Puleva, T., G. Ruzhekov, E. Garipov. Modeling and Control of Power Limited Energy system, IET Conference Publications, vol. 2010 (CP572), 2010.
8. Zhu, R. Speedup of Micromagnetic Simulations with C++ AMP on Graphics Processing Units. – *Computing in Science & Engineering*, 18, 2016, No. 4, 53–59, https://doi.org/10.1109/MCSE.2015.132

Manuscript received on 16.01.2019



***Jordan Kralev**, Ph.D., is currently working as an Assistant Professor in the Department of Systems and Control at Technical University of Sofia. He received his M.Sc. and Ph.D. degrees in Control Systems Theory from the Technical University of Sofia, Bulgaria. His main research interests include: system identification; embedded control systems; robotics; robust control systems.*

Contacts:
Department of System and Control
Technical University of Sofia
8 Kliment Ohridski Bulv., Sofia, Bulgaria
e-mail: jkralev@tu-sofia.bg