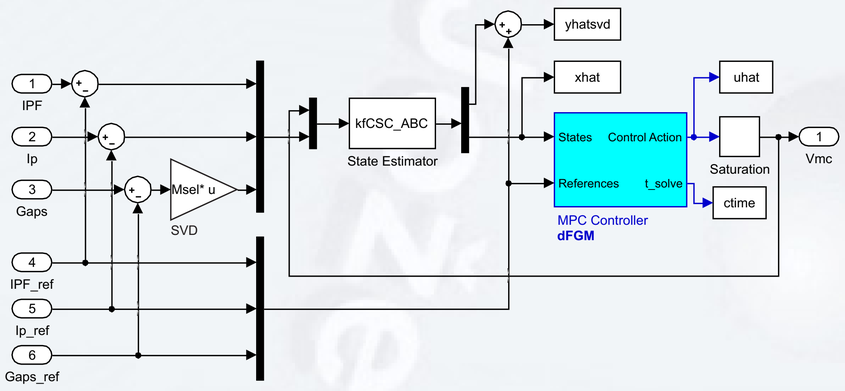
FPGA acceleration of Model Predictive Control for ITER Plasma current and shape control

Samo Gerkšič[a], Boštjan Pregelj[a], Matija Perne[a]

[a] *Jozef Stefan Institute, Jamova cesta 39, 1000 Ljubljana, Slovenia*

Overview

- Plasma Current and Shape Controller (PCSC) for ITER
- Singular Value Decomposition (SVD) -based Model Predictive Control (MPC) using a dual Fast Gradient Method (dFGM) quadratic programming (QP) solver
- FPGA acceleration with a High-Level Synthesis (HLS) approach



SVD-based fast MPC PCSC block scheme

Model Predictive Control (MPC)

MPC is an advanced model-based process control technique, based on on-line optimization of predicted future courses of the process signals. Typically, a simplified control model in the discrete-time linear state space form is used, and the MPC controller is used in conjunction with a Kalman state estimator. It is closely related to LQ (linear quadratic) optimal control, and is convenient for control of multivariable processes such as PCSC. It is efficient in handling of constraints - in the case of ITER PCSC:
- input constraints of the power supply voltages $\mathbf{V}_{PF}$
- output constraints of the superconductive coil currents $\mathbf{I}_{PF}$.

Solving QP problems for MPC using dFGM

In MPC, a QP optimisation problem must be solved in each time step of the algorithm, which is difficult with large-scale multivariable systems with fast dynamics.
Using complexity reduction techniques for MPC and C code optimisation, with the dFGM algorithm the required accuracy was achieved in 500 iterations with peak computation times 3 ms using a standard Intel x86 CPU using a single thread. This is considered sufficiently fast for ITER, but not for experimantation on smaller tokamaks with faster dynamics.
The execution of the dFGM algorithm may be accelerated by parallelisation of matrix-vector numerical operations within iterations, but this is not well-suited for the standard CPU architecture with the thread scheduler timescale 10 μs, thread synchronisation takes more time than computation.

FPGA implementation

FPGA implementation supports small-scale paralellism, but they have other issues regarding the implementation of complex algorithms: relatively low clock frequencies, limited resources for massive parallelism, specific programming.
Aiming to avoid manual recoding as much as possible, the High-Level Synthesis (HLS) approach of Xilinx Vivado HLS was adopted:
- C code (simplifications required: static allocation of variables, direct implementation of functions)
- emulated single float precision (multiple cycles/operation)
- conversion directives (pragma commands) to the VHDL compiler for more optimized implementation:
pipeline, loop unroll, loop merge, array partition
- hardware: Xilinx Zedboard, Zc706

The dFGM main iteration loop must be optimized for speed. Its computational load is dominated by two matrix-vector multiplications (33x99 * 99 and 99x33 * 33), while other vector operations represent less than 10% of the workload.

MPC QP, hard constraints

$$\min_{\mathbf{z}} \frac{1}{2}\mathbf{z}^T\mathbf{H}\mathbf{z} + \mathbf{c}^T\mathbf{z}$$
$$\text{subject to} \quad \mathbf{C}\mathbf{z} \leqslant \mathbf{b}$$

MPC QP, soft state constraints

$$\min_{\mathbf{z},\mathbf{s}} \frac{1}{2}\mathbf{z}^T\mathbf{H}\mathbf{z} + \mathbf{c}^T\mathbf{z} + \frac{1}{2}\mathbf{s}^T\mathbf{W}\mathbf{s} + \mathbf{w}^T\mathbf{s}$$
$$\text{subject to} \quad \begin{matrix}\mathbf{C}_x\mathbf{z} \leqslant \mathbf{b}_x + \mathbf{s} \\ \mathbf{C}_u\mathbf{z} \leqslant \mathbf{b}_u \\ \mathbf{s} \leqslant \mathbf{0}\end{matrix} \qquad \mathbf{C} = \begin{bmatrix}\mathbf{C}_x \\ \mathbf{C}_u\end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix}\mathbf{b}_x \\ \mathbf{b}_u\end{bmatrix}$$

Dual FGM algorithm

$$\mathbf{v}^k = \mathbf{v}^k + \beta^k(\mathbf{v}^k - \mathbf{v}^{k-1})$$
$$\mathbf{y}^k = -\mathbf{H}^{-1}(\mathbf{C}^T\mathbf{v}^k + \mathbf{c})$$
$$\mathbf{v}^{k+1} = \mathbf{v}^k + \mathbf{C}\mathbf{y}^k - \widetilde{\text{prox}}_{h,\mathbf{W},\mathbf{w}}(\mathbf{v}^k + \mathbf{C}\mathbf{y}^k)$$

$$\widetilde{\text{prox}}_{h,\mathbf{W},\mathbf{w}}(\mathbf{t})_i = \begin{cases} t_i & \text{if } t_i \leq b_i \\ b_i & \text{if } t_i > b_i \text{ and } i \text{ hard} \\ b_i & \text{if } b_i + w_i \geq t_i > b_i \text{ and } i \text{ soft} \\ \dfrac{t_i + W_{ii}b_i - w_i}{W_{ii} + 1} & \text{if } b_i + w_i < t_i \text{ and } i \text{ soft} \end{cases}$$

Matrix-vector multiplication implementation

$$\mathbf{r}_{[m\times 1]} = \mathbf{M}_{m\times n} \times \mathbf{v}_{[n\times 1]}$$

Approach 1 (initial)

$$\mathbf{r}_i = \mathbf{r}_i + \mathbf{M}_{i,j} \times \mathbf{v}_j$$

```
// mat_vec_mult_full(d>M, ws->v, ws->tmp_var_n);
mv_mult__M_x_v: for(j = 0; j < n_constraint; j++)
{
    vv_mult__M_x_v: for(i = 0; i < n_opt_var; i++)
    {
        tmp_var_n[i] += M[i][j] * v[j];
    }
}
```

		t=1	t=2	t=3	···	t=7	t=8	t=9	···	t=12	t=13	t=14
i=1	j	RD $M_{i,j}$										
		RD v_j	×	×	×	×						
						RD r_i	+	+	+	WR r_i		
i=2	j		RD $M_{i+1,j}$									
			RD v_{j+1}	×	×	×	×					
							RD r_{i+1}	+	+	+	WR r_{i+1}	
i=3	j			RD $M_{i+2,j}$								
				RD v_{j+2}	×	×	×	×				
								RD r_{i+2}	+	+	+	WR r_{i+2}

Time schedule for parts $\mathbf{M}_{i,j}\times\mathbf{v}_j$ of $\mathbf{M}\times\mathbf{v}$ product.

Approach 2 (inner loop unroll)

Unroll the inner i loop (dimension m)
Partition $\mathbf{M}$ (dimension m)
Partition $\mathbf{r}$ (dimension m)
Pipeline the outer j loop

```
#pragma HLS ARRAY_PARTITION variable=M complete dim=1
#pragma HLS ARRAY_PARTITION variable=n complete
mv_mult__M_x_v: for(j = 0; j < n_constraint; j++)
{
    #pragma HLS PIPELINE
    vv_mult__M_x_v: for(i = 0; i < n_opt_var; i++)
    {
        #pragma HLS UNROLL
        tmp_var_n[i] += M[i][j] * v[j];
    }
}
```

		t=1	···	t=5	t=7	···	t=11	t=9	t=10
j=1	i=1	RD $M_{i,j}$							
		RD v_i	×	×	×				
					RD r_i	+	WR r_i		
	···	···	···	···	···	···	···		
	i=m	RD $M_{i,j}$							
		RD v_j	×	×	×				
					RD r_{i+2}	+	WR r_{i+2}		
	i=1			RD $M_{i,j}$					
				RD v_j	×		×		
							RD r_{i+2}	+	WR r_{i+2}

		t=1	···	t=5	···	t=13	···	···	(n-1)*4 +1	(n-1)*4+12
j=1	i=1									
	···									
	m									
j=2	i=1									
	···							⋮		
	m									
	···	···	···	···	···	···	···	···	···	···
j=n	i=1									
	···							⋮		
	m									

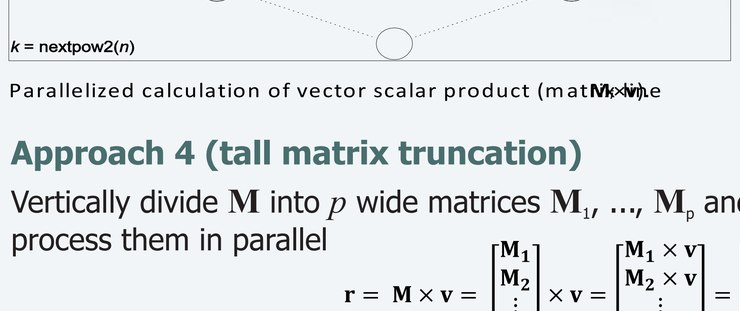
Timing schedule of $\mathbf{M}\times\mathbf{v}$ product, with parallelized inner loop.

Approach 3 (scalar product tree-sum)

Multiplication as series of scalar products:

$$\mathbf{r} = \mathbf{M}\times\mathbf{v} = \begin{bmatrix}\mathbf{M}_{1,1}\mathbf{v}_1 + \mathbf{M}_{1,2}\mathbf{v}_2 + \cdots + \mathbf{M}_{1,n}\mathbf{v}_n \\ \mathbf{M}_{2,1}\mathbf{v}_1 + \mathbf{M}_{2,2}\mathbf{v}_2 + \cdots + \mathbf{M}_{2,n}\mathbf{v}_n \\ \vdots \\ \mathbf{M}_{m,1}\mathbf{v}_1 + \mathbf{M}_{m,2}\mathbf{v}_2 + \cdots + \mathbf{M}_{m,n}\mathbf{v}_n\end{bmatrix} = \begin{bmatrix}\sum_{j=1}^{n}(\mathbf{M}_{1,j}\mathbf{v}_j) \\ \sum_{j=1}^{n}(\mathbf{M}_{2,j}\mathbf{v}_j) \\ \vdots \\ \sum_{j=1}^{n}(\mathbf{M}_{m,j}\mathbf{v}_j)\end{bmatrix} = \begin{bmatrix}\mathbf{r}_1 \\ \mathbf{r}_2 \\ \vdots \\ \mathbf{r}_m\end{bmatrix}$$

Binary-tree-wise parallel summation:



Parallelized calculation of vector scalar product (matrix-vector $\mathbf{M}\times\mathbf{v}$).

Approach 4 (tall matrix truncation)

Vertically divide $\mathbf{M}$ into p wide matrices $\mathbf{M}_1$, ..., $\mathbf{M}_p$ and process them in parallel

$$\mathbf{r} = \mathbf{M}\times\mathbf{v} = \begin{bmatrix}\mathbf{M}_1 \\ \mathbf{M}_2 \\ \vdots \\ \mathbf{M}_p\end{bmatrix} \times \mathbf{v} = \begin{bmatrix}\mathbf{M}_1 \times \mathbf{v} \\ \mathbf{M}_2 \times \mathbf{v} \\ \vdots \\ \mathbf{M}_p \times \mathbf{v}\end{bmatrix} = \begin{bmatrix}\mathbf{r}_1 \\ \mathbf{r}_2 \\ \vdots \\ \mathbf{r}_p\end{bmatrix}$$

```
mv_mult__C_x_x:  for(i = 0; i < 33; i++)  //(i = 0; i < n_constraint; i++)
{
#pragma HLS PIPELINE
vv_mult__C_ x_x: for(j = 0; j < n_opt_var; j++)
    {
        #pragma HLS UNROLL  //factor=33
        //tmp_var_p4[i] += C[i][j] * x[j];
        tmp11[j] = C[i][j] * x[j];
        tmp21[j] = C[i+33][j] * x[j];
        tmp31[j] = C[i+66 ][j] * x[j];
    }
    reset_tmp_rest_m2:  for (j=n_opt_var;j<128;j++)
    {
        #pragma HLS UNROLL
        tmp11[j] = 0.0;
        tmp21[j] = 0.0;
        tmp31[j] = 0.0;
    }
    sum2m2: for (j=0;j<32;j++)
    {
        #pragma HLS UNROLL
        tmp12[j] = tmp11[j]+tmp11[j+32];
        tmp22[j] = tmp21[j]+tmp21[j+32];
        tmp32[j] = tmp31[j]+tmp31[j+32];
    }
    sum3m2: for (j=0;j<16 ;j++)
    {
        #pragma HLS UNROLL
        tmp11[j] = tmp12[j]+tmp12[j+16];
        tmp21[j] = tmp22[j]+tmp22[j+16];
        tmp31[j] = tmp32[j]+tmp32[j+16];
    }
    sum4m2: for (j=0;j<8;j++)
    {
        #pragma HLS UNROLL
        tmp12[j] = tmp11[j]+tmp11[j+8];
        tmp22[j] = tmp21[j]+tmp21[j+8];
        tmp32[j] = tmp31[j]+tmp31[j+8];
    }
    sum5m2: for (j=0;j<4;j++)
    {
        #pragma HLS UNROLL
        tmp11[j] = tmp12[j]+tmp12[j+4];
        tmp21[j] = tmp22[j]+tmp22[j+4];
        tmp31[j] = tmp32[j]+tmp32[j+4];
    }
    tmp12[0] = tmp11[0]+tmp11[2];
    tmp12[1] =  tmp11[1]+tmp11[3];
    tmp_var_p4[i] = tmp12[0] + tmp12[1];
    tmp22[0] = tmp21[0]+tmp21[2];
    tmp22[1] = tmp21[1]+tmp21[3];
    tmp_var_p4[i+33] = tmp22[0] + tmp22[1];
    tmp32[0] = tmp31[0]+tmp31[2];
    tmp32[1] = tmp31[1]+tmp31[3];
    tmp_var_p4[i+66] = tmp32[0] + tmp32[1];
}
```

Matrix-vector multiplication operation count (in clock cycles)

Status	Iteration* latency (Lat)	Initiation* Interval (II)	Iteration* count (It)	Total latency (It*II + Lat)
linitial	*m*	*m* +3	*n*	$n^*(m+3)$ $(+n^{**})$
Inner unroll	12	*4*	*n*	n^* 4 + 12
Scalar product tree-sum	32	1	*m*	*m* + 32
Tall matrix truncation	32	1	*m/p*	*m/p* + 32

* The parameters are stated for the outer loop

** The additional n clocks are required to enter the inner loop *n*-times. This is handled using loop flatten

Conclusions

The automatic conversion does not yield useful results from the original C code, because the code structure prevents automated application of conversion optimization routines, so that the execution time is 45 ms, which is much longer than the CPU implementation, with low use of resources

After a series of manual modifications of the code with pragma directions, the computation time for Xilinx ZC706 is reduced to 1 ms (3x faster than CPU).

Further acceleration is presumably possible by using a FPGA with hardware flow-point multipliers, or by using fixed-point arithmetics (lower dynamic range)

Department of Systems and Control, Jožef Stefan Institute · CREATE PPCC team · COSYLAB · Slovenian Research Agency P2-0001

This work has been carried out within the framework of the EUROfusion Consortium and has received funding from the Euratom research and training programme 2014-2018 under grant agreement No 633053. The views and opinions expressed herein do not necessarily reflect those of the European Commission.