GPU-acceleration of waveform relaxation methods for large differential systems

Dajana Conte, Raffaele D'Ambrosio, Beatrice Paternoster

Dipartimento di Matematica Universitá di Salerno, Fisciano (Sa), 84084 Italy, e-mail: {dajconte,rdambrosio,beapat}@unisa.it.

Abstract

It is the purpose of this paper to provide an acceleration of waveform relaxation (WR) methods for the numerical solution of large systems of ordinary differential equations. The introduced technique is based on the employ of graphic processing units (GPUs) in order to speed-up the numerical integration process. A CUDA solver based on WR-Picard and WR-Jacobi iterations is presented and some numerical experiments realized on a multi-GPU machine are provided.

Key words: Large systems of ordinary differential equations, waveform relaxation methods, Picard and Jacobi iterations, parallel computing, general purpose GPU computing

1. Introduction

Large systems of ordinary differential equations (ODEs) arise as mathematical models in several applications, such as the simulation of large scale integrated circuits and chemical processes [30], cardiac electrical activity [44], atmospheric pollution [43], as well as in the spatial semidiscretization of time-dependent partial differential equations (PDEs), both by finite difference and finite element methods [34, 37, 41].

It is known from the existing literature that the numerical treatment of large systems of ODEs can be more efficient if carried out in a parallel computational environment [1, 27, 28]. Many numerical methods for ODEs suitable for parallel implementations have been introduced in the literature [1, 17, 35, 45]. In the context of multistage methods (such as Runge-Kutta and general linear methods), many formulae are easily parallelizable [11, 12, 14, 15, 18, 19, 20, 29], in particular those depending on structured coefficient matrices [1, 10, 13, 21, 27, 36].

The employ of parallel calculus can significantly increase the speed of computing, but parallel architectures may be quite expensive. A more reasonable balance between efficient solution and sustained cost can be successfully reached by employing modern hardware devices, such as Graphic Processing Units (GPUs). GPUs were initially designed for computer graphics, but nowadays they are employed as general purpose high-performance parallel processors, due to their cheap cost and their great computational capability [25, 26]. In the context of scientific computing, in the last few years several authors have started employing GPUs in many different fields, such as numerical linear algebra, numerical solution of ordinary and partial differential equations.

The purpose of this paper is the employ of GPUs for the numerical solution of large systems of ODEs by means of waveform relaxation (WR) methods, which have widely been used for the parallel numerical solution of functional equations, such as ODEs (see [1, 2, 4, 30] and references therein), Volterra Integral Equations (refer for example to [5, 6, 7] and references therein) and PDEs (see [16, 46, 47] and related bibliography). A GPU acceleration of WR methods has not been treated in the existing literature and, to the best of our knowledge, this paper represents the first attempt in this direction.

2. Waveform relaxation methods

WR methods are iterative methods particularly suited to solve large systems of ODEs. They were introduced in [31], with the aim of efficiently treating large systems of ODEs modeling large scale electrical networks, which are notoriously stiff, overcoming the limits of time-stepping methods. Indeed, the numerical solution of large systems of ODEs via implicit time-stepping methods needs a high computational effort due to the necessity of solving nonlinear

systems of large dimension at each step point. WR iterations are designed in order to decouple the original large system of ODEs in smaller subsystems: in this way, the iteration process can be implemented in a parallel computational environment, since each subsystem can be treated by a single processor. In other words, every processor is responsible for computing the update to a subsystem of differential equations, whose dimension is generally much smaller than that of the given system. This iteration process realizes what is commonly known as *parallelism across the system* [2].

For the initial value problem

$$\begin{cases} y'(t) = f(t, y(t)), \quad t \in [t_0, T], \\ y(t_0) = y_0, \end{cases}$$

with $f : [t_0, T] \times \mathbb{R}^d \to \mathbb{R}^d$, a WR method, starting from an initial function $y^{(0)}(t)$, generates a sequence of continuous-time $\{y^{(\nu+1)}(t)\}_{\nu \geq 0}$ satisfying the differential problem

$$\begin{cases} y^{(\nu+1)'}(t) = F(t, y^{(\nu+1)}, y^{(\nu)}), \\ y^{(\nu+1)}(t_0) = y_0, \end{cases} \tag{2.1}$$

where the splitting function $F : [t_0, T] \times \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^d$ satisfies the consistency condition $F(t, y, y) = f(t, y)$. The convergence of WR iterations has been proved in [38] for the linear case and in [4] for the nonlinear case in an arbitrary norm.

We consider in particular the choice

$$F(t, u, v) = f(t, u),$$

which leads to the WR-Picard method

$$\begin{cases} y_i^{(\nu+1)'}(t) = f_i(t, y^{(\nu)}(t)), \quad i = 1, 2, \ldots, d, \\ y^{(\nu+1)}(t) = y_0, \end{cases} \tag{2.2}$$

decoupling a d-dimensional system of ODEs in d independent quadrature problems, which can be solved in parallel. However, the convergence of this iteration process is very slow (compare [2, 30]). In order to increase the rate of convergence of WR-Picard method, alternative parallelizable iteration schemes can be considered and, in the following, we refer the so-called WR-Jacobi iterations (see, for instance, [2, 4, 30] and references therein)

$$\begin{cases} y_i^{(\nu+1)'}(t) = f_i\left(t,\ y_1^{(\nu)}(t),\ y_2^{(\nu)}(t),\ \ldots,\ y_{i-1}^{(\nu)}(t),\ y_i^{(\nu+1)}(t),\ y_{i+1}^{(\nu)}(t),\ \ldots,\ y_d^{(\nu)}(t)\right), \quad i = 1, 2, \ldots, d, \\ y^{(\nu+1)}(t) = y_0. \end{cases} \tag{2.3}$$

Here the d components of the function F are

$$F_i(t, u, v) = f_i(t, u_1, \ldots, u_{i-1}, v_i, v_{i+1}, v_d), \quad i = 1, 2, \ldots, d.$$

WR discrete-time methods are obtained by employing suitable numerical methods, such as linear multistep formulae (compare [30]), or Runge-Kutta methods (see [1], §8.3), for the solution of the differential problem (2.1) at each iteration.

2.1. WR iterations for linear systems of ODEs

Among large systems of ODEs, linear problems of the form

$$\begin{cases} y'(t) = Ay(t) + g(t), \\ y(t_0) = y_0, \end{cases} \tag{2.4}$$

where $A \in \mathbb{R}^{d \times d}$, $y, g : [t_0, T] \to \mathbb{R}^d$, play an important role, as they derive from the spatial semidiscretization of time-dependent PDEs. For this reason, a separate treatment [1, 2, 4, 38, 30, 39] has been given in the existing literature

concerning WR methods for (2.4). For this class of problems, a WR method is defined in terms of the splitting of the matrix A

$$A = N - M$$

by considering

$$y^{(\nu+1)'}(t) + My^{(\nu+1)}(t) = Ny^{(\nu)}(t) + g(t), \tag{2.5}$$

As in the case of linear systems of algebraic equations, the nature of the splitting determines the type of iterations. For instance, in the case of Picard iterations, we have

$$M = 0, \qquad N = A, \tag{2.6}$$

while, Jacobi iterations are obtained in correspondence of

$$M = -D, \qquad N = L + U, \tag{2.7}$$

where D=diag(A), L and U are the lower and upper triangular part of A, respectively. In the following, we refer to discrete-time WR methods obtained by solving the differential problem (2.5) by a linear multistep method of the form

$$\sum_{j=0}^{k} \alpha_j y_{n+j} = h \sum_{j=0}^{k} \beta_j f(t_{n+j}, y_{n+j}), \tag{2.8}$$

where $\alpha_j, \beta_j \in \mathbb{R}$, $j = 0, 1, \ldots, k$, h is stepsize and $y_n \approx y(t_n)$, being $t_n = t_0 + nh$, and k starting values $y_0, y_1, \ldots, y_{k-1}$ are suitably computed. This choice leads to the iterative scheme

$$\sum_{j=0}^{k} \alpha_j y_{n+j}^{(\nu+1)} + h \sum_{j=0}^{k} \beta_j M y_{n+j}^{(\nu+1)} = h \sum_{j=0}^{k} \beta_j N y_{n+j}^{(\nu-1)} + h \sum_{j=0}^{k} \beta_j g(t_{n+j}), \quad n \geq 0, \tag{2.9}$$

while the iterations do not involve the starting values, i.e. $y_n^{(\nu)} = y_n$, for $n < k$. Equation (2.9) in the unknown $y_{n+k}^{(\nu+1)}$ can be solved uniquely for any n, if and only if $\alpha_k I + h\beta_k M$ is invertible, i.e.

$$\frac{\alpha_k}{\beta_k} \notin \sigma(-hM), \tag{2.10}$$

where σ denotes the spectrum of a matrix.

Convergence results for WR methods (2.5) and (2.9) are reported in [30]. In particular, the continuous-time WR method (2.5) converges in a superlinear way on finite time intervals, while the discrete-time WR method (2.9) satisfying (2.10) converges if and only if

$$\rho\left(\left(\frac{1}{h}\frac{\alpha_k}{\beta_k} I + M\right)^{-1} N\right) < 1, \tag{2.11}$$

where $\rho(A)$ denotes the spectral radius of the matrix A.

3. Implementation issues

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model released by NVIDIA, which provides direct access to the underlying parallel processors in the GPU [40]. It extends C by allowing programmers to define C functions, called *kernels*, that are executed in parallel on the GPU by different CUDA *threads*. Threads are correspondingly grouped into thread *blocks*, which are organized into a *grid*. Each thread is given a unique thread identification number, that is accessible within the kernel through the built-in variables `threadIdx`, `blockIdx`, `blockDim`, which respectively represent the thread identification number within its block, the block identification number within the grid and the dimension of each block.

CUDA kernels are executed on the GPU (*device*), that operates as a coprocessor to the CPU (*host*), running the C program. The host is responsible for allocation of variables in the device global memory and there are some CUDA functions used to copy data between host and device global memory, which is available to all threads and

```
 1   dim3 dimBlock(BLOCK_SIZE);
 2   dim3 dimGrid((int)ceil(float(d)/float(dimBlock.x)));
 3   Niter=0;
 4   repeat
 5          copyVec<<<dimGrid, dimBlock>>>(yold,ynew,d,N);
 6          cudaThreadSynchronize();
 7          lmm<<<dimGrid, dimBlock>>>(yold,ynew,h,t0,f,d,N);
 8          cudaThreadSynchronize();
 9          errComp<<<dimGrid, dimBlock>>>(yold,ynew,errVec,d,N);
10          cudaThreadSynchronize();
11          cudaMemcpy: errVec device ---> host;
12          err=||errVec||_∞;
13          Niter++;
14   until (err>tol and Niter<Nmax)
```

Figure 3.1: Pseudocode for CUDA implementation of WR iterations

persists across kernel invocations. Global synchronization can be achieved between kernel invocations, by the built-in function `cudaThreadSynchronize()`. More details on the programming issues in CUDA are discussed in [40].

We have designed a CUDA program based on discrete time WR-Picard and WR-Jacobi. The implementation strategy of the iterative procedure is described in Fig. 3.1. First of all, we collect the threads in one-dimensional blocks, which are themselves grouped into a one-dimensional grid. The number of threads belonging to each block is chosen by the user and is stored in the integer variable `BLOCK_SIZE`, and the dimension `dimBlock` of each block is defined in line `1` .

The CUDA command of line `2` activates a suitable number of blocks `dimGrid`, in order to have a total number of threads which is at least equal to the dimension d of the system.

Lines `5, 7, 9` contain the typical invocation of CUDA kernels, where inside the symbols `<<< , >>>` the dimension of the grid and of each block is indicated. We employ two dN-dimensional vectors `yold` and `ynew`, allocated on the device, which provide the approximations

$$\texttt{yold[i*N+n]} \approx y_i^{(\nu)}(t_n),$$
$$\texttt{ynew[i*N+n]} \approx y_i^{(\nu+1)}(t_n),$$

with $i = 0, 1, \ldots, d-1$, $n = 0, 1, \ldots, N-1$, $t_n = t_0 + nh$ and $h = (T - t_0)/(N-1)$. By denoting with `i` the identification number of each thread, the kernels are designed so that each thread computes the solution of the `i`-th equation in (2.2) or (2.3), for $\nu \geq 0$, by means of the linear multistep method (2.8), with initial function $y^{(0)}(t) \equiv y_0$. In particular, the kernel `copyVec` updates `yold=ynew`, the kernel `lmm` applies the linear multistep method (2.8) to (2.2) in case of WR-Picard method or to (2.3) in case of WR-Jacobi one for the computation of next iteration `ynew`, and computation of the error estimate vector `errVec=ynew-yold` is carried out in the kernel `errComp`.

In line `11`, we then transfer the vector `errVec` from the device to the host through the built-in function `cudaMemcpy()` and the host finally computes the error estimate as $\|\texttt{errVec}\|_\infty$ in line `12`.

We also observe that, after each kernel, we synchronize the threads through the function `cudaThreadSynchronize()`, in lines `6, 8, 10` . The iteration process stops when the computed error estimate is less than a fixed tolerance *tol* (line `14`).

4. Numerical results

In this section we show the numerical results originated from the application of the discrete time WR method based on both Picard (2.2) and Jacobi (2.3) iterations in GPU environment, by means of a CUDA solver based on the pseudocode described in Fig. 3.1. The computations have been performed on a node with CPU Intel Xeon 6

core X5690 3.46GHz, belonging to the E4 multi-GPU cluster of Mathematics Department of Salerno University. The NVIDIA GPU employed is a 448 cores TESLA C2050 1.15GHz.

We consider the heat equation in one space variable

$$\frac{\partial y}{\partial t} = \frac{\partial^2 y}{\partial x^2}, \quad x \in [0,1], \quad t \in [0,T],$$

with boundary conditions

$$y(0,t) = y(1,t) = 0,$$

and initial condition

$$y(x,0) = y_0(x).$$

A spatial semidiscretization of the problem leads to the following system of ODEs [3]

$$\begin{cases} y'(t) = Ay(t), & t \in [0,T], \\ y(0) = y_0, \end{cases} \tag{4.1}$$

where the matrix $A \in \mathbb{R}^{d\times d}$ has the form

$$A = \begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & 1 & -2 & 1 \\ 0 & \cdots & \cdots & 1 & -2 \end{bmatrix}. \tag{4.2}$$

Such a system is also used in the mathematical modeling of the interconnect resistor-capacitor line (compare [3, 32, 33]).

d	CPUtime	GPUtime	Speed-up
256	250	230	1.09
512	1040	470	2.21
1024	4300	1110	3.87
2048	19670	2340	8.40
4096	87210	12060	7.23
8192	347300	40310	8.61
16384	1375070	167100	8.23

Table 4.1: Numerical results obtained by applying the WR-Picard method to problem (4.1)-(4.2), with $T = 1$, $tol = 10^{-5}$, N=50 and `BLOCK_SIZE`=32. The number of iterations, for each value of d is equal to 18

We consider as underlying linear multistep method (2.8), the one with $k = 1, \beta_1 = \alpha_1 = 1$ and $\alpha_0 = -1$, i.e. the implicit Euler method. For both Picard and Jacobi iterations, condition (2.10) is satisfied for any choice of $h > 0$. As regards the convergence condition (2.11), Picard iterations converge if $h < 1/4$, while Jacobi iterations converge for any $h > 0$, as underlined in the following remark.

Remark 4.1. *Consider the discrete-time WR method* (2.9)*, with the implicit Euler method as underlying linear multistep method* (2.8)*, applied to problem* (4.1)-(4.2)*. Then Picard iterations converge if $h < 1/4$, while Jacobi iterations are convergent for any $h > 0$. As a matter of fact, as regards Picard iterations, from* (2.6) *and $k = 1, \beta_1 = \alpha_1 = 1$ and $\alpha_0 = -1$, convergence condition* (2.11) *reduces to $\rho(hA) < 1$, which, by observing that the eigenvalues of the matrix*

d	CPUtime	GPUtime	Speed-up
256	140	150	0.93
512	580	300	1.93
1024	2390	730	3.27
2048	10420	1800	5.79
4096	48880	8540	5.72
8192	194500	30960	6.28
16384	769150	125750	6.12

Table 4.2: Numerical results obtained by applying the WR-Jacobi method to problem (4.1)-(4.2), with $T = 1$, $tol = 10^{-5}$, N=50 and `BLOCK_SIZE`=32. The number of iterations, for each value of d is equal to 10

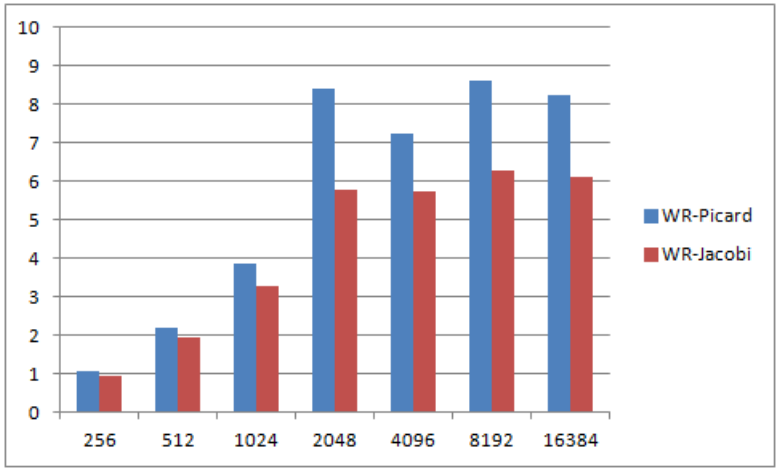

Figure 4.1: Speed-up VS dimension for WR-Picard and WR-Jacobi iterations applied to problem (4.1)-(4.2) with $T = 1$, $tol = 10^{-5}$, N=50 and `BLOCK_SIZE`=32

A in (4.2) *are* $\lambda_k = -2 + 2\cos(k\pi/(d+1)), k = 1, 2, \ldots, d$, *is satisfied if* $h < 1/4$. *Concerning Jacobi iterations, from* (2.7) *and* $k = 1$, $\beta_1 = \alpha_1 = 1$ *and* $\alpha_0 = -1$, *convergence condition* (2.11) *reduces to*

$$\rho\left(\left(\frac{1}{h} + 2\right)^{-1}(L + U)\right) < 1. \tag{4.3}$$

Then, as the eigenvalues of $L + U$ *are* $\lambda_k = 2\cos(k\pi/(d+1)), k = 1, 2, \ldots, d$, *the condition* (4.3) *is satisfied for any* $h > 0$.

Tables 4.1, 4.2 and Fig. 4.1 show the values of speed-up computed as the ratio

$$\text{Speed-up} = \frac{\text{CPUtime}}{\text{GPUtime}},$$

where CPUtime and GPUtime respectively are the serial and CUDA execution times in milliseconds of WR-Picard and WR-Jacobi iterations. We tested the methods for several values of the parameters T, tol, N, `BLOCK_SIZE`, and we report here only the results corresponding to $T = 1$, $tol = 10^{-5}$, $N = 50$, `BLOCK_SIZE`=32, as, for different choices of such parameters, the values of speed-up does not exhibit a substantial change (CPU and GPU times change but their ratio remains almost the same). We observe increasing behaviour of the speed-up when the dimension d of the

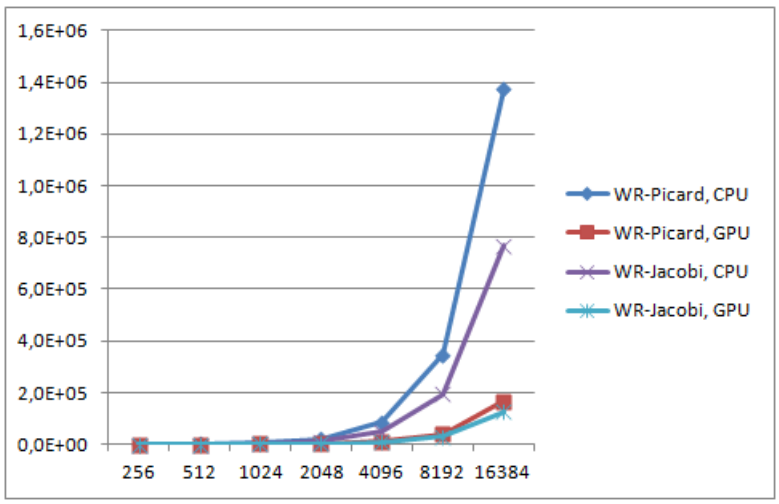

Figure 4.2: Execution time in milliseconds VS dimension for WR-Picard and WR-Jacobi iterations in logarithmic scale applied to problem (4.1)-(4.2) with $T = 1$, $tol = 10^{-5}$, N=50 and BLOCK_SIZE=32

d	CPUtime	GPUtime	Speed-up
32768	1450	410	3.54
65536	3640	780	4.67
131072	7260	1530	4.74
262144	14320	3030	4.73
524288	28500	6000	4.75
1048576	58900	11970	4.92

Table 4.3: Numerical results obtained by applying the WR-Picard method to problem (4.1)-(4.2), with $T = 1$, $tol = 10^{-5}$, N=50 and BLOCK_SIZE=32. The matrix A is stored in tridiagonal format. The number of iterations, for each value of d is equal to 18

system increases and its value stabilizes around 8 for Picard iterations and around 6 for Jacobi ones. Execution times are compared in Fig. 4.2, where we recognize that Jacobi iterations are faster than Picard ones, both as regards the number of iterations and the overall execution time.

In order to deal with higher dimensional problems, we have stored the matrix A in tridiagonal form, i.e. by only storing the three vectors of the diagonal and co-diagonal elements and suitably adapting the algorithm to this memorization strategy. The results are reported in Tables 4.3, 4.4 and Fig.4.3, 4.4. As the dimension increases, the values of the speed-up stabilizes around 5 for Picard iterations and around 7 for Jacobi ones.

Figure 4.5 shows as similar results are also achieved in the case of the heat equation in two space variables that, after a spatial semidiscretization, assumes the form (4.1), with

$$A = \begin{bmatrix} T & I & 0 & \cdots & 0 \\ I & T & I & & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & I & T & I \\ 0 & \cdots & \cdots & I & T \end{bmatrix}, \tag{4.4}$$

d	CPUtime	GPUtime	Speed-up
32768	1010	220	4.59
65536	2840	430	6.60
131072	5730	850	6.74
262144	11580	1670	6.93
524288	22810	3350	6.81
1048576	45550	6610	6.89

Table 4.4: Numerical results obtained by applying the WR-Jacobi method to problem (4.1), with $T = 1$, $tol = 10^{-5}$, N=50 and `BLOCK_SIZE`=32. The matrix A is stored in tridiagonal format. The number of iterations, for each value of d is equal to 10

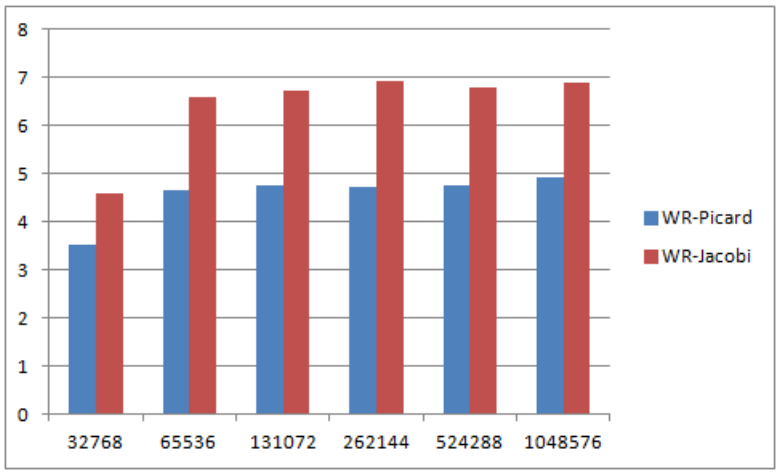

Figure 4.3: Speed-up VS dimension for WR-Picard and WR-Jacobi iterations applied to problem (4.1)-(4.2) with $T = 1$, $tol = 10^{-5}$, N=50 and `BLOCK_SIZE`=32. The matrix A is stored in tridiagonal format

being

$$T = \begin{bmatrix} -4 & 1 & 0 & \cdots & 0 \\ 1 & -4 & 1 & & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & 1 & -4 & 1 \\ 0 & \cdots & \cdots & 1 & -4 \end{bmatrix}.$$

We next consider problem (4.1) of interest in control theory (compare [22, 23]), with $t \in [0, 5]$,

$$a_{ij} = \begin{cases} -(1.5)^{d-i} & i = j,\ i = 1, 2, \ldots, d \\ 0.1 & i = j + 1,\ i = 2, 3, \ldots, d \\ & i = j - 1,\ i = 1, 2, \ldots, d - 1 \\ 0.01 & \text{otherwise} \end{cases} \tag{4.5}$$

and y_0 equal to the d-th vector in the canonical basis of $\mathbb{R}^d$. We observe that for this problem, for any $h > 0$, Picard iterations are not convergent, while Jacobi iterations converge. This follows from the convergence conditions of

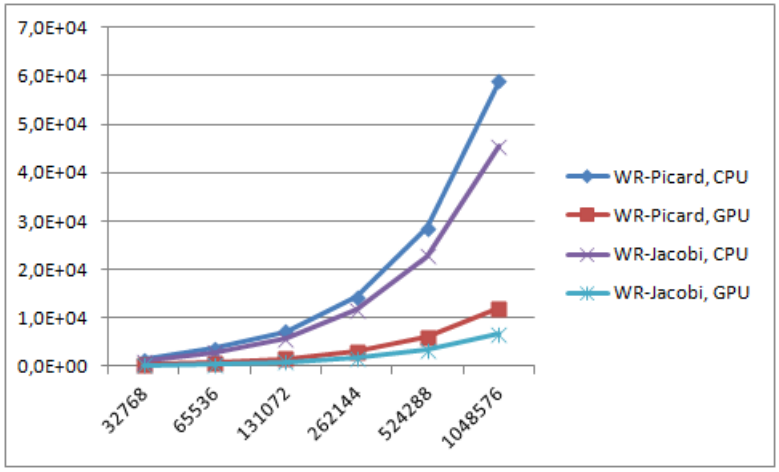

Figure 4.4: Execution time in milliseconds VS dimension for WR-Picard and WR-Jacobi iterations in logarithmic scale applied to problem (4.1)-(4.2) with $T = 1$, $tol = 10^{-5}$, N=50 and `BLOCK_SIZE`=32. The matrix A is stored in tridiagonal format

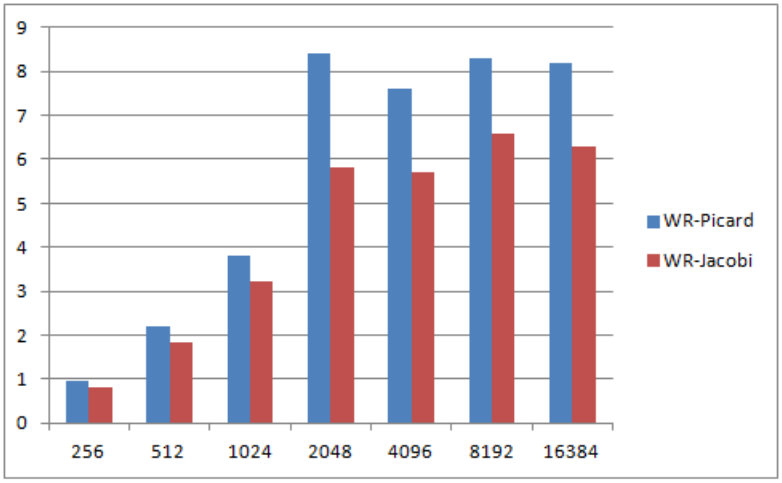

Figure 4.5: Speed-up VS dimension for WR-Picard and WR-Jacobi iterations applied to problem (4.1) with matrix A given by (4.4) with $T = 1$, $tol = 10^{-3}$, N=50 and `BLOCK_SIZE`=32.

Remark 4.1 applied to the matrix A given by (4.5). As a matter of fact, as shown in Figure 4.6, we have $\rho(hA) > 1$ and $\rho\left(\left(\frac{1}{h}+2\right)^{-1}(L+U)\right) < 1$, for any $h > 0$. The values of speed-up for WR Jacobi method are reported in Fig. 4.7. Also in this case we observe an increasing speed-up when the dimension of the problem increases, stabilizing around 7. The execution times are reported in Fig. 4.8.

The speed-up behaviour of Tables 4.1–4.4 and Figures 4.1, 4.3, 4.5, 4.7, can be explained by observing first of all that, when using an n-core CPU with clock frequency f_1 and m-cuda cores on a GPU with clock frequency f_2, the expected ideal speed-up can be measured as $k = \frac{mf_2}{nf_1} = 24.8$ (as in our case $n = 6$, $f_1 = 3.46$, $m = 448$, $f_2 = 1.15$). So the computation time from CPU to GPU should be reduced of the factor $1/k$. As there is always a fraction of the algorithm that must be done in serial, by denoting with p the parallel fraction of the algorithm (i.e. the fraction of the algorithm that is really improved), then the effective obtained speedup is (Ahmdal law)

$$s = \frac{1}{(1-p) + \frac{p}{k}}. \tag{4.6}$$

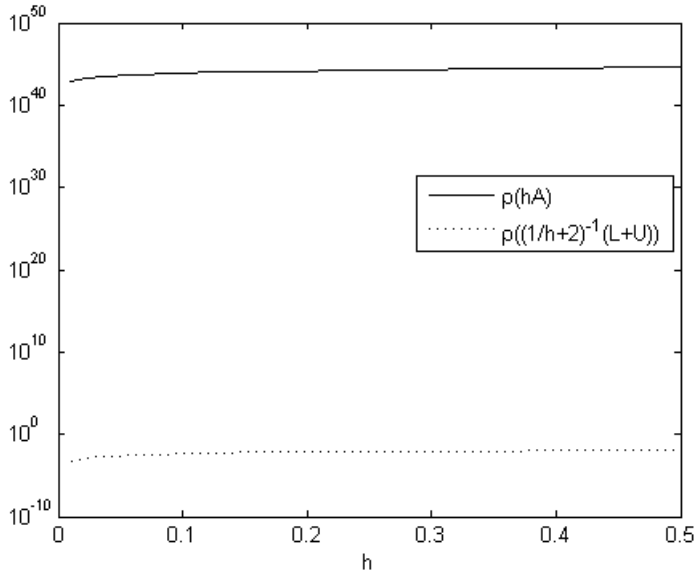

Figure 4.6: Spectral radius for Picard (solid) and Jacobi (dots) WR methods applied to problem (4.1) with A of type (4.5)

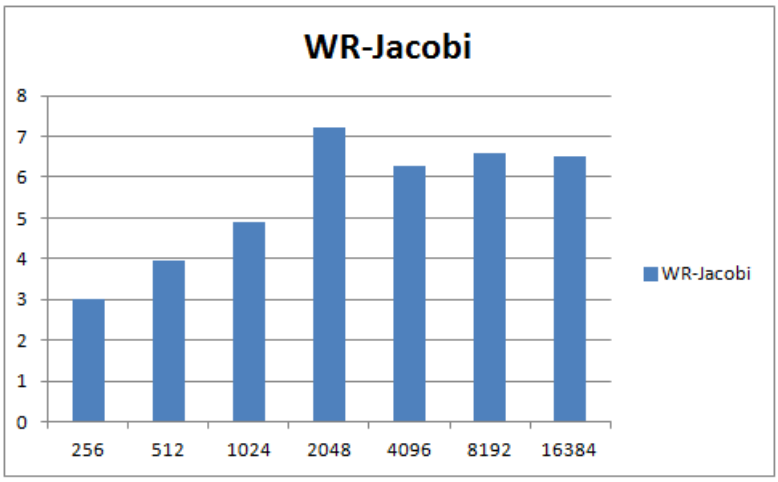

Figure 4.7: Speed-up VS dimension for WR-Jacobi iterations applied to problem (4.1) with A of type (4.5), being $tol = 10^{-5}$, N=50 and `BLOCK_SIZE`=32. The number of iterations, for each value of d is equal to 6

The fraction $1 - p$ has also to take into account also of the computational overhead (e.g. the time due to synchronization, memory transfers between host and GPU device, time to start and terminate a kernel) and depends also on the dimension of the problem and on the number of iterations required by the WR method. The limiting values of the speed-up obtained in Figures 4.1, 4.3 and 4.5 can be motivated by a parallel fraction $p = 0.91$ for WR Picard method and $p = 0.87$ for WR Jacobi method, when the matrix A given by (4.2) or (4.4) is stored as a full matrix, and a parallel fraction $p = 0.83$ for WR Picard method and $p = 0.89$ for WR Jacobi method, when the matrix A in (4.2) is stored in tridiagonal format. As regards Figure 4.5, the limiting value of 7 suggests a parallel fraction $p = 0.89$ for WR Jacobi method when the matrix A is given by (4.5).

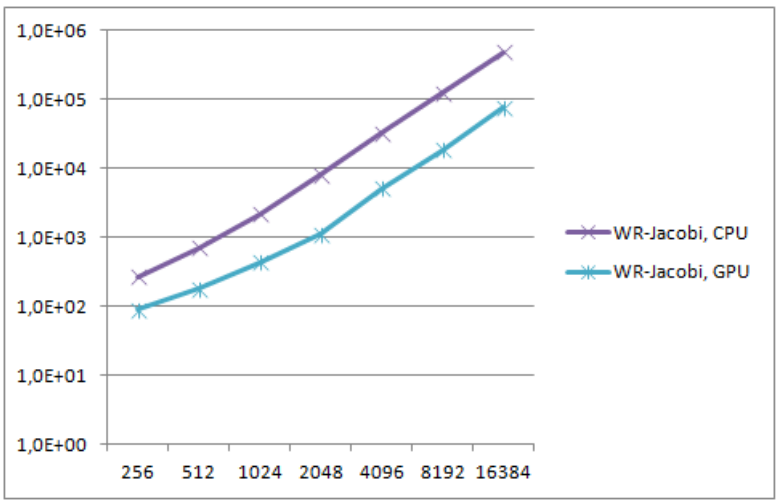

Figure 4.8: Execution time in milliseconds VS dimension for WR-Jacobi iterations in logarithmic scale applied to problem (4.1) with A of type (4.5), being $tol = 10^{-5}$, N=50 and `BLOCK_SIZE`=32

5. Conclusions

We have provided a GPU-acceleration of WR-Picard and Jacobi iterations for the numerical solution of large systems of ODEs. At the best of our knowledge, this paper represents the first contribution regarding the GPU implementation of the WR methods for ODEs. We have provided some numerical results showing the potential of our approach in accelerating the WR methods by employing GPUs. Further developments of this research are regarding the employ of other types of WR iterations in our CUDA solver, as for example the red-black WR Gauss-Seidel or red-black WR SOR methods [42], and for functional equations other than ODEs, following the lines drawn in [8, 9].

6. Acknowledgment

This work was supported by GNCS-INdAM.

References

[1] K. Burrage, Parallel and sequential methods for ordinary differential equations. Numerical Mathematics and Scientific Computation, Oxford University Press, New York (1995).
[2] K. Burrage, C. Dyke, B. Pohl, On the performance of parallel waveform relaxations for differential systems, Appl. Numer. Math. 20 (1-2), 39–55 (1996).
[3] K. Burrage, Z. Jackiewicz, S. P. Norsett, R. A. Renaut, Preconditioning waveform relaxation iterations for differential systems, BIT 36(1), 54–76 (1996).
[4] K. Burrage, J. Sand, A Jacobi Waveform Relaxation Method for ODEs, SIAM J. Sci. Comput. 20 (2), 534–552 (1998).
[5] G. Capobianco, A. Cardone, A parallel algorithm for large systems of Volterra integral equations of Abel type, J. Comput. Appl. Math. 220, 749-758 (2008), dx.doi.org/10.1016/j.cam.2008.05.026 .
[6] G. Capobianco, D. Conte , An efficient and fast parallel method for Volterra integral equations of Abel type, J. Comput. Appl. Math., Vol 189/1-2 pp 481-493 (2006) http://dx.doi.org/10.1016/j.cam.2005.03.056.
[7] G. Capobianco, D. Conte, I. Del Prete, High performance numerical methods for Volterra equations with weakly singular kernels, J. Comput. Appl. Math., 228 (2009) 571579, doi:10.1016/j.cam.2008.03.027.
[8] A. Cardone, E. Messina, E. Russo, A fast iterative method for discretized Volterra-Fredholm integral equations, J. Comput. Appl. Math. 189, 568-579 (2006).
[9] A. Cardone, E. Messina, A. Vecchio, An adaptive method for Volterra-Fredholm integral equations on the half line, J. Comput. Appl. Math. 228(2), 538-547 (2009).
[10] D. Conte, R. D'Ambrosio, Z. Jackiewicz, Two-step Runge-Kutta methods with quadratic stability functions, J. Sci. Comput. 44(2), 191–218 (2010).
[11] D. Conte, R. D'Ambrosio, Z. Jackiewicz, B. Paternoster, A pratical approach for the derivation of algebraically stable two-step Runge-Kutta methods, Math. Model. Anal 17(1), 65-77 (2012).

[12] D. Conte, R. D'Ambrosio, Z. Jackiewicz, B. Paternoster, Numerical search for algebrically stable two-step continuous Runge-Kutta methods, accepted for publication on J. Comput. Appl. Math.
[13] D. Conte, R. D'Ambrosio, B. Paternoster, Two-step diagonally-implicit collocation based methods for Volterra Integral Equations, Appl. Numer. Math. 62(10), 1312-1324 (2012), doi: 10.1016/j.apnum.2012.06.007.
[14] D. Conte, Z. Jackiewicz, B. Paternoster, Two-step almost collocation methods for Volterra integral equations, Appl. Math. Comp. 204, 839–853 (2008).
[15] D. Conte, B. Paternoster, Multistep collocation methods for Volterra Integral Equations, Appl. Numer. Math. 59, 1721–1736 (2009).
[16] Courvoisier, Y., Gander, M.J., Optimization of Schwarz waveform relaxation over short time windows, Numerical Algorithms 64 (2), pp. 221-243, 2013.
[17] M.R.Crisci, B.Paternoster, E. Russo, Fully parallel Runge-Kutta-Nyström methods for ODEs with oscillating solutions, Appl. Numer. Math. 11(13), 143–158 (1993).
[18] R. D'Ambrosio, M. Ferro, Z. Jackiewicz, B. Paternoster, Two-step almost collocation methods for ordinary dierential equations, Numer. Algorithms 53 (2-3), 195-217 (2010).
[19] R. D'Ambrosio, Z. Jackiewicz, Continuous Two-Step Runge-Kutta Methods for Ordinary Differential Equations, Numer. Algorithms 54 (2), 169-193 (2010).
[20] R. D'Ambrosio, Z. Jackiewicz, Construction and implementation of highly stable two-step continuous methods for stiff differential systems, Math.Comput. Simul. 81 (9), 1707-1728 (2011).
[21] R. D'Ambrosio, B. Paternoster, Two-step modified collocation methods with structured coefficient matrices for ordinary differential equations, Appl. Numer. Math. 62(10), 1325-1334 (2012).
[22] E. J. Davison, An Algorithm for the Computer Simulation of Very Large Dynamic Systems, Automatica 9, 665–675 (1973).
[23] W. Enright, On the efficient and reliable numerical solution of large linear systems of O.D.E.'s, IEEE Transactions on Automatic Control AC-24(6), 905–908 (1979).
[24] Everstine, Numerical Solution of PDE, gwu.geverstine.com/pdenum.pdf (2010).
[25] http://gpgpu.org/category/research
[26] http://gpucomputing.net/
[27] E. Hairer, S. P. Norsett, G. Wanner, Solving ordinary differential equations. I. Nonstiff problems. Second edition. Springer Series in Computational Mathematics, 8. Springer-Verlag, Berlin (1993).
[28] E. Hairer, G. Wanner, Solving ordinary differential equations. II. Stiff and differential-algebraic problems. Second revised edition, paperback. Springer Series in Computational Mathematics, 14. Springer-Verlag, Berlin (2010).
[29] K. R. Jackson, S. P. Norsett, The Potential for Parallelism in Runge-Kutta Methods. Part 1: RK Formulas in Standard Form., SIAM J. Numer. Anal. 32, 49–82 (1990).
[30] J. Janssen, Acceleration of waveform relaxation methods for linear ordinary and partial differential equations, PhD-thesis, Katholieke Universiteit Leuven, Belgium (1997).
[31] E. Lelarasmee, The Waveform Relaxation Method for the Time-Domain Analysis of Large-Scale Nonlinear Systems, Ph.D. thesis Department of Electrical Engineering and Computer Science, University of California, Berkeley, USA (1982).
[32] B. Leimkuhler, Estimating Waveform Relaxation Convergence, SIAM J. Sci. Comput 14, 872–889 (1993).
[33] B. Leimkuhler, A. Ruehli, Rapid convergence of waveform relaxation, Appl. Numer. Math. 11(1–3), 211–224 (1993)
[34] L. Lopez, Methods based on boundary value techniques for solving parabolic equations on parallel computers, Parallel Computing 19 (9), 979–991 (1993).
[35] L. Lopez, T. Politi, Parallel methods in the numerical treatment of population dynamic models, Parallel Computing 18 (7), 767–777 (1992).
[36] L. Lopez, T. Politi, Tridiagonal splittings in the conditioning and parallel solution of banded linear systems, Linear Algebra and its Applications 251, 249–265 (1997).
[37] L. Lopez, D. Trigiante, A finite difference scheme for a stiff problem arising in the numerical solution of a population dynamic model with spatial diffusion, Nonlinear Analysis 9 (1), 1–12 (1985).
[38] U. Miekkala, O. Nevanlinna, Convergence of dynamic iterations for initial value problems, SIAM J. Sci. Stat. Comp. 8, 459–482 (1987).
[39] U. Miekkala, O. Nevanlinna, Iterative Solution of systems of linear differential equations, Acta Numerica 5, 259–307 (1996).
[40] NVIDIA Corporation, NVIDIA CUDA compute unified device architecture programming guide, `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf` (2012).
[41] A. Quarteroni, A. Valli, Numerical approximation of partial differential equations. Springer Series in Computational Mathematics, 23. Springer-Verlag, Berlin (1994).
[42] M. Reichelt , J. White and J. Allen, Waveform relaxation for transient simulation of two-dimensional MOS devices,Proc. IEEE Int. Conf. Comp.-Aided Design (1989).
[43] V. Simek, R. Dvorak, F. Zboril, J. Kunovsky, Towards Accelerated Computation of Atmospheric Equations Using CUDA, UKSIM '09 11th International Conference on Computer Modelling and Simulation, 449–454 (2009).
[44] R.J. Spiteri, R. C. Dean, On the Performance of an Implicit-Explicit Runge-Kutta Method in Models of Cardiac Electrical Activity, Biomedical Engineering, IEEE Transactions on 55 (5), 1488–1495 (2008).
[45] P. J. van der Houwen, B. P. Sommeijer, CWI contributions to the development of parallel Runge-Kutta methods, Appl. Numer. Math. 22 (1-3), 327–344 (1996).
[46] Van Lent, J., Vandewalle, S., Multigrid waveform relaxation for anisotropic partial differential equations, Numerical Algorithms 31 (1-4), pp. 361-380, 2002.
[47] Zhang, H., Jiang, Y.-L., A note on the H1-convergence of the overlapping Schwarz waveform relaxation method for the heat equation, Numerical Algorithms 66 (2), pp. 299-307, 2014.