GPU Accelerated Smooth Formation Redeployment in Multiagent Environment

Michal Matuszak & Tomasz Schreiber

Faculty of Mathematics and Computer Science,
Nicolaus Copernicus University,
Toruń, Poland
{gruby, tomeks}@mat.umk.pl

Abstract. The problem of determining optimal formation reorganization in a given time plays important role in many fields from aeronautics like satellites or missiles reconfigurations to entertainment such as controlling agents in computer games. We introduce an algorithm for discovering optimal transition paths between given configurations. Crucial role in presented method plays efficient computation of the matrix exponential. We recalled one of the best algorithms in solving that task - the scaling and squaring method combined with Padé approximants. To test the efficiency of the algorithm we have implemented a simple sandbox environment with use of the NVIDIA CUDA technology.

Keywords: formation redeployment, transition path, CUDA, reconfiguration, matrix exponential

1 Introduction

In this paper, we describe a system for controlling motion and redeployment between unit's formations. We formulate the problem in terms of stochastic diffusion processes. With each unit is associated a particle subject to a small noise. Most likely behaviour of stochastic diffusion processes under conditioning

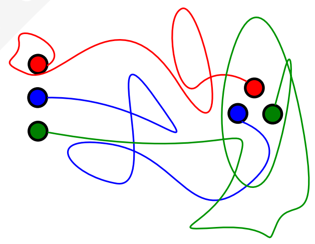


Fig. 1. The noise allow for rare transitions between stable configurations.

is effectively characterised by the large deviation theorems due to Freidlin and

Wentzell [3]. Reformulation of these results on the space of curves in a set-up well suited for our needs was presented in [4]. Analytical solution to the minimisation of the action functional was presented in [8]. The obtained smooth transition paths can be effectively simulated. Capabilities and wide adoption of modern graphics cards encourage us to parallelise our algorithm. A comprehensive survey of formation control is presented in [13, 14].

2 Dynamical system

A physical system in time interval $t \in [a, b]$ moves on trajectory

$$t \to \psi_t,$$

where ψ_t describes configuration at moment t. The Least Action Principle, of fundamental use for our application, indicates that the physical system traverses given path minimising the action functional:

$$\int_a^b L(t, \psi_t, \dot{\psi}_t)dt = S(\psi) \tag{1}$$

Given system is represented by dynamical Gaussian network. $X(t)$ is a system configuration at time t. It is a vector determined by network nodes

$$X(t + dt) = X(t) + b(X(t)) + \sqrt{\epsilon}\Sigma W(t), \tag{2}$$

where:

- b - mandatory translation vector, $b(X) = BX$
- W(t) - Wiener process, $W(t + dt) = W(t) + \mathcal{N}(0, dt)$
- $A = \Sigma\Sigma^T$ - diffusion tensor.

The functional in Eq. 1 can be minimised on the interval $[0, T]$

$$S_T(\psi) = \int_0^T L\left(\psi, \dot{\psi}\right) dt \tag{3}$$

Detailed discussion about that minimisation is out of scope of this paper and can found in [8]. Action functional has the following form:

$$L(\psi, \dot{\psi}) = \frac{1}{2}\left\langle \dot{\psi} - B\psi, A^{-1}\left(\dot{\psi} - B\psi\right)\right\rangle \tag{4}$$

Solving the Euler-Lagrange equations leads to:

$$\dot{w}_{t+dt} = \dot{w}_t + \left(ABA^{-1}\right)\dot{w}_t dt \tag{5}$$

Initial steering configuration w_0, for simplicity we assume that $A = 1$, is given by

$$w_0 = \frac{1}{T}exp(-TB)\left[\psi_T - exp(TB)\psi_0\right]. \tag{6}$$

3 Formation Redeployment

To simulate multiagent formation we introduce dependencies between agents. Figure 2 represent relationship between nodes in defined formations. Simple formations such as *line* or *double line* require only a connection between nearest nodes. In more complex structures we have to extend number of arcs. In *circle* formation each node is connected to adjacent nodes and to their neighbours. Formations *wee* and *wedge* require additional connection to the nodes on the opposite branch. We assume that for each connection exists an edge with opposite direction and value. For each pair of connected nodes x and y a pair of vectors

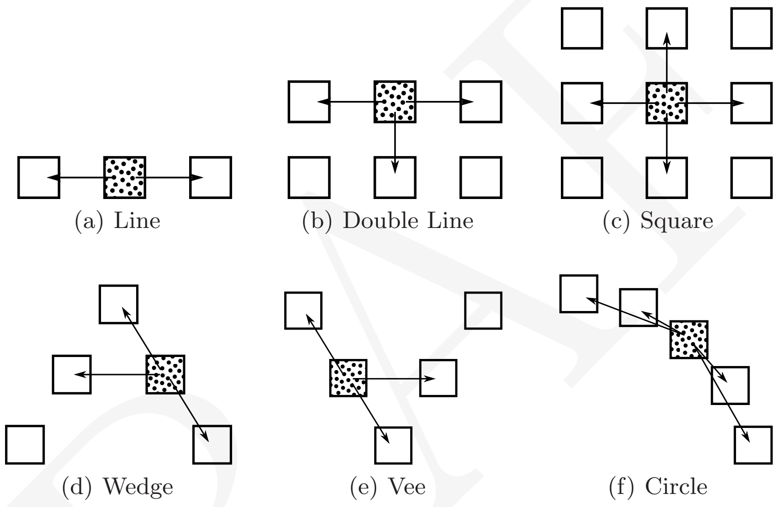


Fig. 2. Dependencies (arcs) in constructed Gaussian network.

is given:

- $\overrightarrow{v}_{x \to y}$
- $\overrightarrow{v}_{y \to x} = -\overrightarrow{v}_{x \to y}$

Let $N(x)$ denote the number of neighbours, $\boldsymbol{v}$ the velocity of the entire formation and $\alpha \in [0,1]$ represents the impact of neighbours on node position then

$$\overrightarrow{x}(t+dt) = \overrightarrow{x}(t)\,(1-\alpha dt) + \overrightarrow{v}dt + \alpha dt \left[\frac{1}{N(x)} \sum_{y \sim x} \left(\overrightarrow{y}(t) + \overrightarrow{v}_{y \to x} \right) \right] \qquad (7)$$

Movements of an agent's set can be fully described by Eq. 2. It can be accomplished with translation of Eq. 7 to the drift transformation b. More precisely, the matrix B can be constructed in the following way:

- Add artificial node $e \equiv 1$
- $B_{xx} = -\alpha$
- $B_{xy} = \begin{cases} \frac{\alpha}{N(x)} & \text{if } y \sim x \\ 0 & \text{if } y \not\sim x \end{cases}$
- $B_{ex} \equiv 0$
- $B_{xe} = \boldsymbol{v} + \frac{\alpha}{N(x)} \sum_{y \sim x} \boldsymbol{v}_{x \to y}$.

4 The Exponential of a Matrix

For a given square matrix A, the exponential of A, denoted by e^A or $exp(A)$, is defined as:

$$e^A = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \cdots = \sum_{n=0}^{\infty} \frac{A^n}{n!}. \tag{8}$$

Computation of the matrix exponent is crucial for solving differential equations. Numerous methods for computing e^A were developed. The straightforward one which uses Euler series (Def. 8) is inefficient even in the scalar case. The survey [9] presented a wide variety of methods and pointed the most powerful ones. Today due to evolution in computer hardware and highly optimised BLAS subroutines the most cogent technique is the scaling and squaring method combined with Padé approximants [5]. Implementation of that algorithm can be found in MATLAB 2009a [7]. We will present only key ideas and main conclusions presented in [9, 5], because the detailed discussion about this algorithm is beyond the scope of this paper.

Relation $e^A = (e^{A/t})^t$, which is unique to the exponential function, plays key role in the scaling and squaring method. To minimise error from Padé algorithm we should ensure that $\|A\|/t \leq 1$. It can be easily accomplished by choosing $t = \lceil log_2 \|A\|/\theta_m \rceil$, where θ_m is the largest value of $\|2^{-k}A\|$, such that the backward error bound does not exceed the unit roundoff in IEEE arithmetic. Values of θ_m for single, double and quadratic arithmetic can be found in [5].

The $[k/m]$ Padé approximants to the exponential function e^A can be written explicitly for all k and m:

$$R_{km}(A) = [Q_{km}(A)]^{-1} P_{km}(A),$$

where

$$P_{km}(A) = \sum_{j=0}^{k} \frac{(k+m-j)!k!}{(k+m)!(k-j)!} \frac{A^j}{j!}, \; Q_{km}(A) = \sum_{j=0}^{k} \frac{(k+m-j)!m!}{(k+m)!(m-j)!} \frac{(-A)^j}{j!}.$$

Diagonal Padé approximants are preferred so for $k = m$ we will write $R_m(A) = [Q_m(A)]^{-1} P_m(A)$. We can notice that $Q_m(A) = P_m(-A)$ then if $\sum_{i=0}^{m} c_i A^i$, we have:

$$P_{2m+1}(A) = A(c_{2m+1}A^{2m} + \cdots + c_3A^2 + c_1 I) + c_{2m}A^{2m} + \cdots + c_2A^2 + c_0 I =: U + V$$

and $Q_{2m+1} = V - U$. For $m \geq 12$ the equations can be simplified:

$$\begin{aligned} P_{13}(A) &= A(A^6(c_{13}A^6 + c_{11}A4 + c_9A^2) + c_7A^6 + c_5A^4 + c_3A^2 + c_1I) \\ &\quad + A^6(c_{12}A^6 + c_{10}A4 + c_8A^2) + c_6A^6 + c_4A^4 + c_2A^2 + c_0I \\ &:= U + V, \end{aligned}$$

and $Q_{13} = V - U$. After computing Q_m and P_m we are ready to solve matrix equation $Q_mR_m = P_m$. Pseudo code for double arithmetic is presented below.

```
if $\|A\| > \theta_{13}$ then
  $ss \leftarrow TRUE$ {scaling and squaring is required}
end if
if $ss = FALSE$ then
  for $k \in \{3, 5, 7, 9, 13\}$ do
    if $\|A\| \leq \theta_k$ then
      $m \leftarrow k${scaling and squaring is not required}
      break
    end if
  end for
else
  $m \leftarrow 13$
  $A \leftarrow A/2^s$ {s - integral power of 2 from breaking $\|A\|/\theta_{13}$ into mantissa and
  exponent}
end if
if $m \leq 13$ then
  $U \leftarrow c_2I + c_4A^2$
  $V \leftarrow c_1I + c_3A^2$
  if $m \geq 3$ then
    $U \leftarrow U + c_6A^4$
    $V \leftarrow V + c_5A^4$
    if $m \geq 5$ then
      $U \leftarrow U + c_8A^6$
      $V \leftarrow V + c_7A^6$
      if $m \geq 7$ then
        $U \leftarrow U + c_{10}A^8$
        $V \leftarrow V + c_9A^8$
      end if
    end if
  end if
else if $m = 13$ then
  $U \leftarrow A^6(c_{13}A^6 + c_{11}A4 + c_9A^2) + c_7A^6 + c_5A^4 + c_3A^2 + c_1I$
  $V \leftarrow A^6(c_{12}A^6 + c_{10}A4 + c_8A^2) + c_6A^6 + c_4A^4 + c_2A^2 + c_0I$
end if
$U \leftarrow A * U$
solve $(V - U)X = (V + U)$
if $ss = TRUE$ then
  $X \leftarrow X^{2^s}$
```

```
end if
return $X$
```

The worst-case cost of the algorithm in double arithmetic is $6+\lceil log_2\|A\|/\theta_m\rceil$ matrix multiplications plus the solution of one matrix equation. To parallelise the algorithm we use NVIDIA CUBLAS library [10] and substitute serial matrix operations with parallel ones.

5 NVIDIA CUDA

Primary purpose of parallel programming is performance, otherwise we would still be writing sequential code. For many years it was difficult to obtain access to truly parallel hardware such as data-centre servers or clusters. From the 2000s researchers, astonished with floating point performance in graphics cards (Fig. 3), started using GPUs for running general purpose computational applications [6]. Early fields of applications were medical imaging and electromagnetics. The main shortcoming for researchers was a very difficult programming model. Developers interested in General-Purpose computation on Graphics Processing Units (GPGPU) had to use graphics APIs, such as OpenGL (with GLSL or Cg) or DirectX (with HLSL). OpenGL tends to be preferred in the open community due to the platform undependability. DirectX, on the other hand, tends to be favoured in the computer game industry, where dependence on Windows is not a particular drawback. In practice, both APIs work as effectively for GPGPU. The data had to be expressed in terms of vertex coordinates, textures and shader programs also random reads and writes to memory were prohibited. Due to complexity in graphics APIs for non-graphics programmers and other listed drawbacks only few people could achieve a masterpiece in efficient use of GPU.

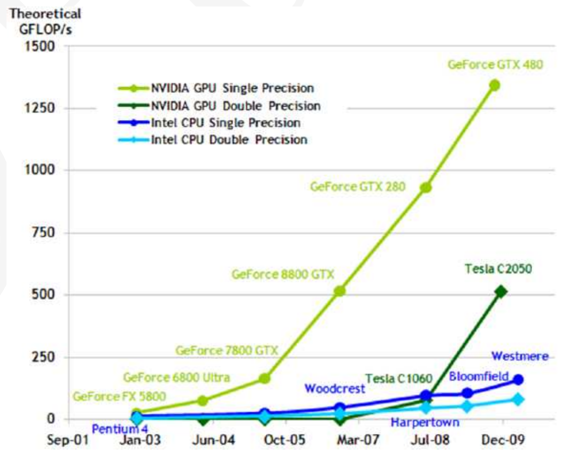
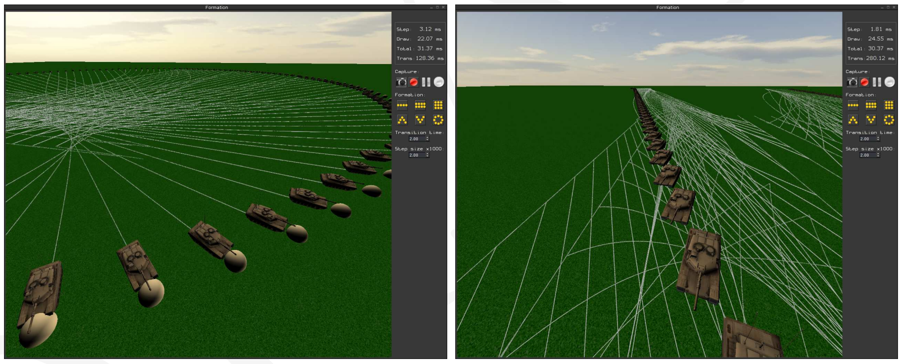


Fig. 3. Floating-Point Operations per Second. Image courtesy of NVIDIA Corporation.

In November 2006, NVIDIA presented a novel massively parallel architecture for GPGPU called CUDA [11] (an acronym for Compute Unified Device Architecture) which addresses GPGPU problems. CUDA provides a set of abstractions that enable expressing data and task parallelism in an easy way with extension to Flynn's taxonomy [2] with a new architecture called SIMT (Single-Instruction, Multiple-Thread). It combines SIMD execution within a block (on an Streaming Multiprocessor) with SPMD execution across blocks (distributed across Streaming Multiprocessors). Developers can chose between high-level languages such as C, C++, Fortran, OpenCL, and DirectCompute [12]. The evolution of programming environment is connected with progression in graphic cards capabilities. In Fig. 3 we can see rapid growth of GPUs performance and also the limitations to random reads/writes were relaxed.

For facile use of GPU power in matrix and vector operations NVIDIA had provided an implementation of BLAS [10] (Basic Linear Algebra Subprograms) for CUDA. CUBLAS is an self-contained library at the API level.

6 Results


Step: 3.12 ms
Draw: 22.07 ms
Total: 31.37 ms
Trans: 128.36 ms
Transition time: 2.00
Step size x1000: 2.00

Step: 1.81 ms
Draw: 24.55 ms
Total: 30.37 ms
Trans: 280.12 ms
Transition time: 2.00
Step size x1000: 2.00

Fig. 4. Screenshots from the sandbox. Image courtesy of [8].

The programme has been implemented in language `D` [1] and C. Some matrix operations incorporate LAPACK, BLAS and CUBLAS [10] subroutines. All test runs were executed on a machine with Intel Core 2 Q9300 2.50 GHz CPU, 4GB RAM and NVIDIA GeForce GTX 480[1]. The CPU application is single

[1] NVIDIA GeForce GTX 480 card is the fastest single GPU video card available for mainstream customers during writing this paper. There exists NVIDIA Tesla C2050/C2070 processors which are dedicated for HPC, with unlocked double precision (DP) units which makes them four times faster in DP computations than GTX 480.

threaded, so is applicable to the core of only one processor. All computations were performed with double precision arithmetic.

The mean square error (MSE) between the target configuration and a simulated one can by controlled by adjusting step size. For *transition time* set on 2.0 and *step size* on 0.002 the MSE was always lower than 0.0001. In Table 1 we can

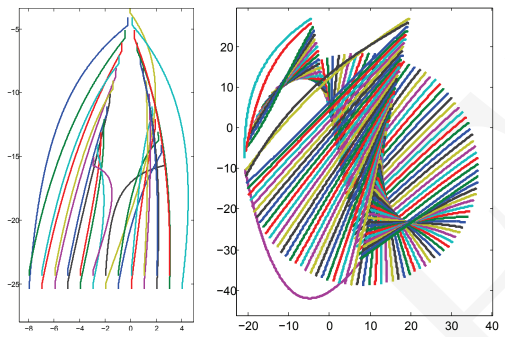


Fig. 5. On the left we can see transitions of 25 units from *vee* formation to *double line*. On the right 121 tanks redeploy from *circle* to *wedge* configuration. Image courtesy of [8].

see dependencies between the number of tanks and the time required for a single step and transition. As we can see, a single step can be computed very fast. The

quantity	step (ms)	CPU (ms)	GPU (ms)	speedup
25	0.04	1.7	40	0.07
100	0.6	24	109	0.22
169	1.6	75	193	0.38
225	3.1	172	275	0.63
400	9	680	637	1.07
625	22	2200	1470	1.50
900	49	12000	3480	3.45
1225	89	54000	6890	7.84
1600	155	253000	12400	17.57
2025	240	785000	26200	**29.96**

Table 1. Step and transition times (see [8] for more information).

most consuming part during w_0 calculation is matrix exponential. GPU accelerated version is up to 30 times faster than sequential one. An analysis of parallel profiler output shows that time is mainly spent (up to 70%) on matrix multiplications (*dgemm*). Transfers of the data from host to device (CPU $\rightarrow$ GPU) and vice versa take up to 30% of the time. The rest of the time is consumed on memory allocations and other matrix operations. We should emphasise that computations are performed for x, y and z coordinates independently.

Acknowledgments

The authors gratefully acknowledge the support from the Polish Ministry of Scientific Research and Higher Education grant N N201 385234 (2008-2010). The work of M. Matuszak has also been supported by the European Social Fund, Government of Poland and Kuyavian-Pomeranian Voivodeship as a part of Integrated Operational Program for Regional Development, Activity 8.2.2.

References

1. Bell, K., Igesund, L.I., Kelly, S. Parker, M. Learn to Tango with D, Apress (2008).
2. Duncan, R., A Survey of Parallel Computer Architectures, *IEEE Computer:* **516** (1990).
3. Freidlin, M.I., Wentzell, A.D. Random perturbations of dynamical systems, Vol. 260, *Grundlehren der Mathematischen Wissenschaften (2nd ed.)* New York: Springer-Verlag (1998).
4. Heymann, M., Vanden-Eijnden, E. The Geometric Minimum Action Method: A Least Action Principle on the Space of Curves, *Comm. Pure Appl. Math.* **61.8**, 1052-1117 (2008).
5. Higham, N. J. The Scaling and Squaring Method for the Matrix Exponential Revisited, *SIAM J. Matrix Anal. Appl.,* **26(4)**, pp. 1179–1193 (2005).
6. Kirk, D.B., Hwu, W.-M.W. Programming Massively Parallel Processors: A Hands-on Approach, *Morgan Kaufmann*, 1 edition (2010).
7. MATLAB version 7.8.0.347 (R2009a). Natick, Massachusetts: The MathWorks Inc., (2009).
8. Matuszak, M., Schreiber, T. Smooth Conditional Transition Paths in Dynamical Gaussian Networks, KI 2011: Advances in Artificial Intelligence, LNAI 7006, pp. 204–215, 2011.
9. Moler, C., Van Loan, C. Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later, *SIAM Rev.* **45, 3** (2003).
10. NVIDIA CUBLAS Library, *NVIDIA Corporation,* (2009).
11. NVIDIA CUDA Architecture: Introduction & Overview, *NVIDIA Corporation,* (2009).
12. NVIDIA Programming Guide 3.2, *NVIDIA Corporation,* (2010).
13. Scharf, D.P., Hadaegh, F.Y. and Ploen, S.R. A Survey of Spacecraft Formation Flying Guidance and Control (Part I): Guidance, *American Control Conference, Vol. 2*, pp. 1733-1739, (June 2003).
14. Scharf, D.P., Hadaegh, F.Y. and Ploen, S.R. A Survey of Spacecraft Formation Flying Guidance and Control (Part II): Control, *American Control Conference, Vol. 4*, pp. 2976-2985, (30 June-2 July 2004).