# Exponential integrators on graphic processing units

Lukas Einkemmer, Alexander Ostermann

**Abstract**—In this paper we revisit stencil methods on GPUs in the context of exponential integrators. We further discuss boundary conditions, in the same context, and show that simple boundary conditions (for example, homogeneous Dirichlet or homogeneous Neumann boundary conditions) do not affect performance if implemented directly into the CUDA kernel. In addition, we show that stencil methods with position-dependent coefficients can be implemented efficiently as well. As an application, we discuss the implementation of exponential integrators for different classes of problems in a single and multi GPU setup (up to 4 GPUs). We will show that for stencil based methods such parallelization can be done very efficiently while for some unstructured matrices the parallelization to multiple GPUs is severely limited by the throughput of the PCIe bus.

**Index Terms**—GPGPU, exponential integrators, time integration of differential equations

## 1 Introduction

The step size for the time integration of stiff ordinary differential equations (or the semidiscretization of partial differential equations) is usually limited by a stability condition. In order to overcome this difficulty implicit schemes are employed that are usually stable for much larger step sizes; however, such schemes have to solve a nonlinear system of equations in each time step and are thus costly in terms of performance. In many instances the stiffness of the differential equation is located in the linear part only. In this instance, we can write our differential equation as a semilinear problem

$$\frac{\mathrm{d}}{\mathrm{d}t}u(t) + Au(t) = g(u(t)), \tag{1}$$

where, for example, $A$ is a matrix with large negative eigenvalues and $g$ is a nonlinear function of $u(t)$; it is further assumed that appropriate initial conditions are given. The boundary conditions are incorporated into the matrix $A$. Since the linear part can be solved exactly, a first-order method, the exponential Euler method, is given by

$$u_{n+1} = \mathrm{e}^{-hA}u_n + h\varphi_1(-hA)\,g(u_n), \tag{2}$$

where $\varphi_1$ is an entire function. In [8] a review of such methods, called exponential integrators, is given and various methods of higher order are discussed. The main advantage, compared to Runge–Kutta methods, is that an explicit method is given for which the step size is only limited by the nonlinearity. It has long been believed that the computation of the matrix functions in (2) can not be carried out efficiently. However, if a bound of the field of values of $A$ is known a priori, for example, polynomial interpolation is a viable option. In this case the application of Horner's scheme reduces the problem to repeated matrix-vector products of the form

$$(\alpha A + \beta I)x, \tag{3}$$

where $A \in \mathbb{K}^{n\times n}$ is a sparse matrix, $I$ is the identity matrix, $x \in \mathbb{K}^n$, and $\alpha, \beta \in \mathbb{K}$ with $\mathbb{K} \in \{\mathbb{R}, \mathbb{C}\}$. That such a product can be parallelized to small clusters has been shown in [4] (for an advection-diffusion equation that is discretized in space by finite differences).

The parallelization of sparse matrix-vector products to graphic processing units (GPUs), similar in form to (3), has been studied in some detail. Much research effort in improving the performance of sparse matrix-vector multiplication on GPUs has focused on developing more efficient data structures (see e.g. [3] or [1]). This is especially important on GPUs as coalesced memory access is of

• *L. Einkemmer and A. Ostermann are with the Department of Mathematics, University of Innsbruck, Austria.*
*E-mail: lukas.einkemmer@uibk.ac.at*

paramount importance if optimal performance is to be achieved. Data structures, such as ELLRT, facilitate coalesced memory access but require additional memory. This is somewhat problematic as on a GPU system memory is limited to a greater extend than on traditional clusters. To remedy this situation a more memory efficient data structure has been proposed, for example, in [7]. Nevertheless, all such methods are extremely memory bound. On the other hand, the parallelization of finite difference computations (called stencil methods in this context) to GPU's have been studied, for example, in [11] and [5]. Even though such methods do not have to store the matrix in memory they are memory bound; for example, in [5] the flops per byte ratio is computed to be $0.5$ for a seven-point stencil (for double precision computations) which is still far from the theoretical rate of $3.5$ that a C0275 can achieve. Both papers mentioned above do not consider boundary conditions in any detail. More recently, in [9], an algorithm is introduced that generates CUDA kernels from a high level description of the finite difference scheme. The performance of the generated code, however, is still a factor of at least $1.5$ away from hand tuned implementations (see [5]).

This paper is structured as follows. In section 2 we will revisit stencil methods in the context of (3) and discuss the implementation of boundary conditions as well as non-constant diffusion coefficients and their performance characteristics. In addition, we will study the performance characteristics of CUDA devices with compute capability 1.1 (C1060) and 2.0 (C2075). In section 3 we will discuss the second part of the implementation of an exponential integrator for a differential equation as given in (1), namely the computation of the non-linearity. We will investigate both a non-linearity drawn from a combustion problem as well as a class of non-linearities that can be implemented by using stencil methods. In section 4 we generalize the discussion of sections 2 and 3 to a multiple GPUs setup where (in our case) $4$ C1060 GPUs are connected via the PCIe bus. We will investigate which class of problems can be efficiently parallelized on such a system. Finally, we conclude in section 5.

## 2 Stencil methods

Let us focus our attention first on the standard seven-point stencil resulting from a discretization of the Laplacian in three dimensions, i.e.

$$\begin{aligned}(\Delta x)^2 (Au)_{i_x,i_y,i_z} = &-6u_{i_x,i_y,i_z} \\ &+ u_{i_x+1,i_y,i_z} + u_{i_x-1,i_y,i_z} \\ &+ u_{i_x,i_y+1,i_z} + u_{i_x,i_y-1,i_z} \\ &+ u_{i_x,i_y,i_z+1} + u_{i_x,i_y,i_z-1},\end{aligned}$$

where $\Delta x$ is the spacing of the grid points. The corresponding matrix-vector product given in (3) can then be computed without storing the matrix in memory. For each grid point we have to perform at least $2$ memory operations (a single read and a single store) as well as $10$ floating point operations (6 additions and 2 multiplication for the matrix-vector product as well as single addition and multiplication for the second part of (3)).

One could implement a stencil method that employs $8$ memory transactions for every grid point. Following [5] we call this the *naive* method. On the other hand we can try to minimize memory access by storing values in shared memory or the cache (note that the C1060 does not feature a cache but the C2075 does). Since no significant 3D slice fits into the relatively limited shared memory/cache of both the C1060 and C2075 we take only a 2D slice and iterate over the remaining index. Similar methods have been implemented, for example, in [5] and [11]. We will call this the *optimized* method.

To implement boundary conditions we have two options. First, a stencil method can be implemented that considers only grid points that lie strictly in the interior of the domain. Second, we can implement the boundary conditions directly into the CUDA kernel. The approach has the advantage that all computations can be done in a single kernel launch. However, conditional statements have to be inserted into the kernel. Since the kernel is memory bound we do not expect a significant performance decrease at least for boundary conditions that do not involve the evaluation of complicated functions.

The results of our numerical experiments (for both the naive and optimized method) are given in Table 1.

Before we discuss the results let us note that on a dual socket Intel Xeon E5355 system the aggressively hand optimized stencil method implemented in [5] gives a mere $2.5$ Gflops/s; that is only slightly better than the $1.7$ Gflops/s we get for a CSR based implementation on a dual socket Intel Xenon E5620.

In [5] a double precision performance of $36.5$ Gflops/s is reported for a GTX280 of compute

TABLE 1
Timing of a single stencil based matrix-vector computation for a number of implementations and boundary conditions. The corresponding Gflops/s is shown in parentheses. All computations are performed with $n = 256^3$.

| Device | Boundary | Method | Double | Single |
|---|---|---|---|---|
| C1060 | None | Stencil (naive) | 13.8 ms (12) | 8.2 ms (20.5) |
| | | Stencil (optimized) | 7.6 ms (22) | 8.0 ms (21) |
| | Homogeneous Dirichlet | Stencil (naive) | 13.4 ms (12.5) | 7.6 ms (22) |
| | | Stencil (optimized) | 8.8 ms (19) | 9.2 ms (18) |
| | $z(1-z)xy$ | Stencil (optimized) | 36 ms (4.5) | 39 ms (4.5) |
| | $\sin(\pi z)\exp(-xy)$ | Stencil (optimized) | 54 ms (3) | 56 ms (3) |
| C2075 | None | Stencil (naive) | 5.5 ms (30.5) | 3.1 ms (54) |
| | | Stencil (optimized) | 4.3 ms (39) | 2.9 ms (58) |
| | Homogeneous Dirichlet | Stencil (naive) | 6 ms (28) | 3.5 ms (48) |
| | | Stencil (optimized) | 5 ms (33.5) | 3.9 ms (43) |
| | $z(1-z)xy$ | Stencil (naive) | 12.3 ms (13.5) | 6.9 ms (24) |
| | | Stencil (optimized) | 7 ms (24) | 6.0 ms (28) |
| | $\sin(\pi z)\exp(-xy)$ | Stencil (naive) | 14.3 ms (11.5) | 13.8 ms (12) |
| | | Stencil (optimized) | 9.7 ms (17.5) | 6.8 ms (24.5) |

capability 1.3. However, the theoretical memory bandwidth of the GTX280 is 141.7 GB/s and thus, as we have a memory bound problem, it has to be compared mainly to the C2075 (which has the same memory bandwidth as the GTX280). Note that the C1060 (compute capability 1.1) has only a memory bandwidth of 102.4 GB/s. In our case we get 39 Gflops/s for no boundary conditions as well as homogeneous Dirichlet boundary conditions. Since we do not solve exactly the same problem, a direct comparison is difficult. However, it is clear that the implemented method is competitive especially since we do not employ any tuning of the kernel parameters.

We found it interesting that for the C2075 (compute capability 2.0) there is only a maximum of 30% performance decrease if the naive method is used instead of the optimized method for none or homogeneous boundary conditions (both in the single and double precision case). Thus, the cache implemented on a C2075 works quite efficiently in this case. However, we can get a significant increase in performance for more complicated boundary conditions by using the optimized method. In the single precision case the expected gain is offset, in some instances, by the additional operations that have to be performed in the kernel (see Table 1).

Finally, we observe that the performance for homogeneous Dirichlet boundary conditions is at most 10% worse than the same computation which includes no boundary conditions at all. This difference completely vanishes if one considers the optimized implementation. This is no longer true if more complicated boundary conditions are prescribed. For example, if we set

$$f(x, y, z) = z(1 - z)xy,$$

for $(x, y, z) \in \partial([0, 1]^3)$ or

$$f(x, y, z) = \sin(\pi z)\exp(-xy),$$

for $(x, y, z) \in \partial([0, 1]^3)$, the performance is decreased by a factor of about 2 for the C2075 and by a factor of 5-7 for the C1060. Thus, in this case it is clearly warranted to perform the computation of the boundary conditions in a separate kernel launch. Note, however, that the direct implementation is still faster by a factor of 3 as compared to CUSPARSE and about 40 % better than the ELL format (see [2]). The memory requirement are an even bigger factor in favor of stencil methods; a grid of dimension $512^3$ would already require 10 GB in the storage friendly CSR format. Furthermore, the implementation of such a kernel is straight forward and requires no division of the domain into the interior and the boundary.

Let us now discuss the addition of a position-dependent diffusion coefficient, i.e. we implement the discretization of $D(x, y, z)\Delta u$ as a stencil method (this is the diffusive part of $\nabla \cdot (D\nabla u)$. For the particular choice of $D(x, y, z) =$

$1/\sqrt{1+x^2+y^2}$, taken from [12], the results are shown in Table 2.

TABLE 2
Timing of a single stencil based matrix-vector computation for a position dependent diffusion coefficient given by $D(x,y,z) = 1/\sqrt{1+x^2+y^2}$. All computations are performed with $n = 256^3$.

**Double precision**

| Device | Method | Time |
|---|---|---|
| C1060 | Stencil (naive) | 37 ms (4.5 Gflops/s) |
| | Stencil (optimized) | 42 ms (4 Gflops/s) |
| C2075 | Stencil (naive) | 10.7 ms (15.5 Gflops/s) |
| | Stencil (optimized) | 10.5 ms (16 Gflops/s) |

**Single precision**

| Device | Method | Time |
|---|---|---|
| C1060 | Stencil (naive) | 37 ms (4.5 Gflops/s) |
| | Stencil (optimized) | 45 ms (3.5 Gflops/s) |
| C2075 | Stencil (naive) | 10.2 ms (16.5 Gflops/s) |
| | Stencil (optimized) | 10.3 ms (16 Gflops/s) |

Thus, a performance of 16 Gflops/s can be achieved for this particular position-dependent diffusion coefficient. This is a significant increase in performance as compared to a matrix-based implementation. In addition, the same concerns regarding storage requirements, as raised above, still apply equally to this problem. No significant difference between the naive and optimized implementation can be observed; this is due to the fact that this problem is now to a large extend compute bound.

Finally, let us note that the results obtained in Tables 1 and 2 are (almost) identical for the $n = 512^3$ case. Thus, for the sake of brevity, we choose to omit those results.

## 3 Evaluating the non-linearity on a GPU

For an exponential integrator, usually the most time consuming part is evaluating the exponential and $\varphi_1$ function. Fortunately, if the field of values of $A$ can be estimated a priori, we can employ polynomial interpolation to reduce that problem to matrix-vector multiplication; a viable possibility is interpolation at Leja points (see [8]). Then, our problem reduces to the evaluation of a series of matrix-vector products of the form given in (3) and discussed in the previous section, and the evaluation of the non-linearity for a number of intermediate approximations. In this section we will be concerned with the efficient evaluation of the non-linearity on a GPU.

Since the non-linearity is highly problem dependent, let us – for the sake of concreteness – take a simple model problem, namely the reaction-diffusion equation modeling combustion in three dimensions (see [10, p. 439])

$$u_t = \Delta v + g(u) \tag{4}$$

with non-linearity

$$g(u) = \frac{1}{4}(2-u)\mathrm{e}^{20\left(1-\frac{1}{u}\right)}$$

and appropriate boundary conditions as well as an initial condition.

In addition to the discretization of the Laplacian which can be conducted by stencil methods (as described in section 2) the parallelization of the non-linearity can be conducted pointwise on the GPU. That is (in a linear indexing scheme) we have to compute

$$\frac{1}{4}(2-u_i)\mathrm{e}^{20\left(1-\frac{1}{u_i}\right)}, \qquad 0 \le i < N. \tag{5}$$

This computation requires only two memory operations per grid point (one read and one store); however, we have to perform a single division and a single exponentiation. Since those operations are expensive, the problem is expected to be compute bound. The results of our numerical experiments are shown in Table 3.

As expected the GPU has a significant advantage over our CPU based system in this case. Fast math routines can be employed if precision is not critical and the evaluation of the non-linearity is the bottleneck in the computation.

Let us note that the non-linearity of certain semi-linear PDEs resembles more the performance characteristics of the stencil methods discussed in section 2. For example, Burger's equation, where $g(\boldsymbol{u}) = (\boldsymbol{u}\cdot\nabla)\boldsymbol{u}$, falls into this category. Such non-linearities can be efficiently implemented by the methods discussed in section 2.

If we combine sections 2 and 3 we have all ingredients necessary to conduct an efficient implementation of exponential integrators on a single GPU. The specific performance characteristics depends on the form of the linear as well as the non-linear part of the differential equation under consideration. In the next section we will turn our attention to the parallelization of exponential integrators to multiple GPUs.

TABLE 3
Timing of a single computation of the non-linearity given in (5). Results for both full precision computations as well as the fast math routines implemented in the GPU are listed. As a reference a comparison to a dual socket Intel Xenon E5620 setup is provided.

**Double precision**

| Device | Method | $n = 256^3$ | $n = 512^3$ |
|---|---|---|---|
| 2x Xenon E5620 | OpenMP | 480 ms | 4 s |
| C1060 | Full precision | 14.6 ms | 120 ms |
| | Fast math | 6.9 ms | 55 ms |
| C2075 | Full precision | 4.2 ms | 33 ms |
| | Fast math | 2.4 ms | 20 ms |

**Single precision**

| Device | Method | $n = 256^3$ | $n = 512^3$ |
|---|---|---|---|
| 2x Xenon E5620 | OpenMP | 515 ms | 4 s |
| C1060 | Full precision | 15.4 ms | 120 ms |
| | Fast math | 7.6 ms | 61 ms |
| C2075 | Full precision | 2.6 ms | 34 ms |
| | Fast math | 1.6 ms | 19 ms |

## 4 Multiple GPU implementation of exponential integrators

If we consider the problem introduced in (4) to be solved with an exponential integrator, we have at least two possibilities to distribute the workload to multiple GPUs. First, one could compute the different matrix functions on different GPUs. However, since even for higher order schemes we only have to evaluate a small number of distinct matrix functions, this approach is not very flexible and depends on the method under consideration. However, if we are able to implement a parallelization of the matrix-vector product and the non-linearity onto multiple GPUs, a much more flexible approach would result.

Such an undertaking however is limited by the fact that in the worst case we have to transfer

$$(m-1)n \tag{6}$$

floating point numbers over the relatively slow PCIe bus ($m$ is the number of GPUs whereas $n$ is, as before, the problem size). However, in the case of differential operators only a halo region has to be updated after every iteration and thus the actual memory transfer is a tiny fraction of the value given by (6). Such a procedure was suggested in [4] for use on a cluster, where parallelization is mainly limited by the interconnection between different nodes. For performance reasons on a GPU it is advantageous to first flatten the halo regions in memory and copy it via a single call to `cudaMemcpy` to the device. Then the vector is updated by using that information in a fully parallelized way on the GPU. As can be seen from the results given in Table 4, the problem in (4) shows good scaling behavior (at least) up to 4 GPUs.

TABLE 4
Performance comparison for the combustion model discussed in section 3 for a single time step using $40$ matrix-vector products (a tolerance of tol $= 10^{-4}$ was prescribed for a time step of size $10^{-4}$). A finite difference discretization with $n = 256^3$ has been used.

**Double precision**

| Device | Method | Number units | Time |
|---|---|---|---|
| 2x Xenon E5620 | CSR/OpenMP | 2 | 9.5 s |
| C1060 | Stencil hom. Dirichlet | 1 | 1.5 s |
| | | 4 | 320 ms |
| C2075 | Stencil hom. Dirichlet | 1 | 1.2 s |

**Single precision**

| Device | Method | Number units | Time |
|---|---|---|---|
| 2x Xenon E5620 | CSR/OpenMP | 2 | 5.6 s |
| C1060 | Stencil hom. Dirichlet | 1 | 1.2 s |
| | | 4 | 540 ms |
| C2075 | Stencil hom. Dirichlet | 1 | 210 ms |

Let us now discuss a different example. In certain discrete quantum systems, for example, the solution of (see, e.g., [6])

$$\partial_t \psi = H(t)\psi$$

is to be determined, where $\psi$ is a vector with complex entries in a high dimensional vector space and $H(t)$ a Hermitian matrix. Such equations are efficiently solved by using Magnus integrators. In this paper we will use the example of a two spin system in a spin bath. In this case $H(t)$ is independent of time and thus we can, in principle, take arbitrarily large time steps. The matrix $H$ is generated beforehand and stored in the generic CSR

format; for 21 spins this yields a vector with $n = 2^{21}$ complex entries and a matrix with approximately $83.9 \cdot 10^6$ non-zero complex entries (the storage requirement is about 2 GB in the double precision and 1 GB in the single precision case). This gives a sparsity of $2 \cdot 10^{-5}$. Note, however, that such quantum systems couple every degree of freedom with every other degree of freedom. Thus, we are in the worst case and have to transfer $(m-1)n$ floating point numbers over the PCIe bus after each iteration. The results of our numerical experiments are shown in Table 5.

TABLE 5
Performance comparison for a system with 21 spins. Integration is performed up to $t = 10$ with a tolerance of tol $= 10^{-5}$.

**Double precision**

| Device | Method | Number units | Time |
|---|---|---|---|
| 2x Xenon E5620 | CSR/OpenMP | 2 | 46 s |
| C1060 | CSR | 1 | 23 s |
| | | 2 | 15 s |
| | | 4 | 15 s |
| C2075 | CSR | 1 | 7.7 s |

**Single precision**

| Device | Method | Number units | Time |
|---|---|---|---|
| 2x Xenon E5620 | CSR/OpenMP | 2 | 44 s |
| C1060 | CSR | 1 | 15 s |
| | | 2 | 10 s |
| | | 4 | 7.5 s |
| C2075 | CSR | 1 | 4 s |

Clearly the scaling behavior in this case is limited by the overhead of copying between the different GPUs. For two GPUs a speedup of about 1.5 can be observed. For any additional GPU no performance gain can be observed. In total a speedup of 3 for double precision and 6 for single precision as compared to a dual socket Xenon configuration is achieved on four C1060 graphic processing units. This is only about 50% better than the speedup of 2 (double precision) and 3 (single precision) achieved with a single C1060 GPU. Thus, in this instance the speedups that are achievable in both single and multi GPU configurations are a consequence of an unstructured matrix that makes coalesced memory access as well as parallelizability between different GPUs difficult. The dramatically better performance of the C2075 as shown in Table 5 is thus expected.

## 5 Conclusion

We have shown that exponential integrators can be efficiently implemented on graphic processing units. For many problems, especially those resulting from the spatial discretization of partial differential equations, this is true for both single and multi GPU setups.

In addition, we have considered stencil implementations that go beyond periodic boundary conditions and constant diffusion coefficients. Such problems can not be handled by implementations based on the fast Fourier transform, for example. Moreover, section 2 shows that for non-homogeneous boundary condition the code handling the interior as well as the boundary of the domain has to be separated if optimal performance is to be achieved. However, for homogeneous or piecewise constant boundary condition an implementation directly into the CUDA kernel does not result in any significant performance decrease.

## References


[1] M.M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs. *IBM Research Report*, RC24704, 2009.

[2] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical report, NVIDIA NVR-2008-004, 2008.

[3] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 18, New York, USA, 2009. ACM Press.

[4] M. Caliari, M. Vianello, and L. Bergamaschi. Interpolating discrete advection-diffusion propagators at Leja sequences. *J. Comput. Appl. Math.*, 172(1):79–99, November 2004.

[5] K. Datta, M. Murphy, V. Volkov, S. Williams, and J. Carter. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. *Journal of Parallel and Distributed Computing*, 69(9):762–777, 2009.

[6] H. De Raedt and K. Michielsen. Computational methods for simulating quantum computers. *arXiv preprint quant-ph/0406210*, 2008.

[7] A. Dziekonski, A. Lamecki, and M. Mrozowski. A memory efficient and fast sparse matrix vector product on a GPU. *Progress in Electromagnetics Research*, 116:49–63, 2011.

[8] M. Hochbruck and A. Ostermann. Exponential integrators. *Acta Numer.*, 19:209–286, 2010.

[9] J. Holewinski, L.N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on GPU architectures. *Proceedings of the 26th ACM international conference on supercomputing*, pages 311–320, 2012.

[10] W. Hundsdorfer and J.G. Verwer. *Numerical solution of time-dependent advection-diffusion-reaction equations*. Springer-Verlag, Berlin, Heidelberg, New York, 2nd edition, 2007.

[11] P. Micikevicius. 3D Finite Difference Computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84, Washington, DC, USA, 2009.

[12] M.V. Vazquez, A.M. Berezhkovskii, and L. Dagdug. Diffusion in linear porous media with periodic entropy barriers: A tube formed by contacting spheres. *J. Chem. Phys*, 129(4), 2008.