Exponential Integrators on Graphic Processing Units

Lukas Einkemmer, Alexander Ostermann
Department of Mathematics
University of Innsbruck
Innsbruck, Austria
lukas.einkemmer@uibk.ac.at, alexander.ostermann@uibk.ac.at

***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 the 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 further 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.**

***Keywords*—GPGPU, exponential integrators, time integration of differential equations, stencil methods, multi GPU setup**

I. Introduction

The emergence of graphic processing units as a massively parallel computing architecture as well as their inclusion in high performance computing systems have made them an attractive platform for the parallelization of well established computer codes.

Many problems that arise in science and engineering can be modeled as differential equations. In most circumstances the resulting equations are sufficiently complex such that they can not be solved exactly. However, an approximation computed by the means of a given numerical scheme can still serve as a valuable tool for scientists and engineers. The collection of techniques generally referred to as general-purpose computing on graphics processing units (GPGPU) provide the means to speed up such computations significantly (see e.g. [1] or [2]).

If a finite difference approximation in space is employed (such methods are widely used in computational fluid dynamics, for example), stencil methods provide an alternative to storing the matrix in memory (see e.g. [3]). In many instances, this is advantageous both from a memory consumption as well as from a performance standpoint. The resulting system of ordinary differential equations then has to be integrated in time.

Much research has been devoted to the construction of efficient time integration schemes as well as their implementation (see e.g. [4] and [5]). The implementation of Runge–Kutta methods, which are the most widely known time integration schemes, on GPUs for ordinary differential equations can result in a significant speedup (see [1]). However, a class of problems has been identified, so called stiff problems, where standard integration routines (such as the above mentioned Runge–Kutta methods) are inefficient (see e.g. [6]).

Exponential integrators are one class of methods that avoid the difficulties of Runge–Kutta method if applied to stiff problems. For such schemes analytical functions (e.g. the exponential function) of large matrices have to be computed. Exponential integrators and some of their applications are discussed in detail in [6]. In this paper we will consider a polynomial interpolation scheme to compute the matrix functions; this essentially reduces the problem of efficiently implementing exponential integrators to sparse matrix-vector multiplication as well as computing the nonlinearity of a given differential equation. The computation of matrix-vector multiplications, e.g. by using stencil methods, is usually the most time intensive part of any exponential integrator; thus, an efficient implementation of stencil methods is vital.

A. Research problems & Results

In the literature, see section II-B, stencil methods are considered for trivial boundary conditions in the context of a differential operator with constant coefficients (i.e. the Laplacian). Such simplifying assumptions, however, are usually not satisfied in a given application. It is not clear from the literature how much stencil methods can be extended beyond the situation described above while still maintaining an efficient implementation. We propose a method based on the integration of boundary conditions and position-dependent coefficients directly into the CUDA kernel and show that such methods can be applied widely without a significant impact on performance.

In addition, it has been shown in [2] that stencil methods can be efficiently parallelized to at least 4 GPUs. Our objective is to show that such results can be generalized to implementations of exponential integrators for a large class of nonlinearities.

The remainder of this paper is structured as follows. In section II-A a introduction explaining the GPU architecture and the corresponding programming model is given. In addition, we discuss previous work which considers the implementation of stencil methods on GPUs and elaborate on the necessary steps to efficiently implement an exponential integrator (sections II-B and II-C, respectively). In section III we present our results as summarized below.

- Stencil methods that include simple, but non-trivial, boundary conditions, such as those required in many applications, can still be efficiently implemented on GPUs (section III-A). For homogeneous boundary conditions on the C2075 33.5 Gflops/s are observed.
- Position dependent coefficients (such as a position dependent diffusion) can efficiently be implemented on the GPU provided that the coefficients are not extremely expensive to compute (section III-B). For a real world example 16 Gflops/s are observed.
- A wide class of nonlinearities can be computed efficiently on the GPU (section III-C).
- The parallelization of exponential integrators to multiple GPUs can be conducted very efficiently for discretized differential operators (perfect scaling to at least 4 GPUs) and is mainly limited by the throughput of the PCIe express bus for unstructured matrices (section III-D).

Finally, we conclude in section IV.

II. BACKGROUND & MOTIVATION

A. GPU architecture

A graphic processing unit (GPU) is a massively parallel computing architecture. At the time of writing two frameworks to program such systems, namely OpenCL and NVIDIA's CUDA, are in widespread use. In this section we will discuss the hardware architecture as well as the programming model of the GPU architecture using NVIDIA's CUDA (all our implementations are CUDA based). Note, however, that the principles introduced here can, with some change in terminology, just as well be applied to the OpenCL programming model. For a more detailed treatment we refer the reader to [7].

The hardware consists of so called SM (streaming multiprocessors) that are divided into cores. Each core is (as the analogy suggests) an independent execution unit that shares certain resources (for example shared memory) with other cores, which reside on the same streaming multiprocessor. For example, in case of the C2075, the hardware consists in total of 448 cores that are distributed across 14 streaming multiprocessors of 32 cores each.

For scheduling, however, the hardware uses the concept of a warp. A warp is a group of 32 threads that are scheduled to run on the same streaming multiprocessor (but possibly on different cores of that SM). On devices of compute capability 2.0 and higher (e.g. the C2075) each SM consists of 32 cores (matching each thread in a warp to a single core). However, for the C1060, where 240 cores are distributed across 30 SM of 8 core each, even threads in the same warp that take exactly the same execution path are not necessarily scheduled to run in parallel (on the instruction level).

To run a number of threads on a single SM has the advantage that certain resources are shared among those threads; the most notable being the so called shared memory. Shared memory essentially acts as an L1 cache (performance wise) but can be fully controlled by the programmer. Therefore, it is often employed to avoid redundant global memory access as well as to share certain intermediate computations between cores. In addition, devices of compute capability 2.0 and higher are equipped with a non-programmable cache.

The global memory is a RAM (random access memory) that is shared by all SM on the entire GPU. For the C1060 and C2075 GPUs used in this paper the size of the global memory is 4 GB and 6 GB respectively (with memory bandwidth of 102.4 GB/s and 141.7 GB/s respectively), whereas the shared memory is a mere 16 KB for the C1060 and about 50 KB for the C2075. However, this memory is available per SM.

From the programmer some of these details are hidden by the CUDA programming model (most notably the concept of SM, cores, and warps). If a single program is executed on the GPU we refer to this as a grid. The programmer is responsible for subdividing this grid into a number of blocks, whereas each block is further subdivided into threads. A thread in a single block is executed on the same SM and therefore has access to the same shared memory and cache.

GPUs are therefore ideally suited to problems which are compute bound. However, also memory bound problems, such as sparse matrix-vector multiplication, can significantly benefit from GPUs. We will elaborate on this statement in the next section.

B. Stencil methods and matrix-vector products on GPUs

The parallelization of sparse matrix-vector products to GPUs 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. [8] or [9]). This is especially important on GPUs as coalesced memory access is of 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 [10]. 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 GPUs has been studied, for example, in [2] and [3]. Even though such methods do not have to store the matrix in memory they are still memory bound; for example, in [3] 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. In [3] a performance of 36.5 Gflops/s has been demonstrated for a seven-point stencil on a GTX280.

Both papers mentioned above do not consider boundary conditions in any detail. However, in applications of science and engineering where exponential integrators are applied at least simple boundary conditions have to be imposed (see e.g. [6]). In addition, in the literature stated above only the discretization of the Laplacian is considered. However, often position-dependent coefficients have to be employed (to model a position-dependent diffusion as in [11], for example). In this case it is not clear if stencil methods retain their superior performance characteristics (as compared to schemes that store the matrix in memory). We will show in sections III-A and III-B that for many applications both of these difficulties can be overcome and stencil methods on GPUs can be implemented efficiently.

C. Exponential integrators

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 in many applications 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 φ_1 is an entire function. In [6] 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 [12] (for an advection-diffusion equation that is discretized in space by finite differences).

Finally, let us discuss the evaluation of the nonlinearity. In many instances the nonlinearity can be computed pointwise. In this case its evaluation is expected to be easily parallelizable to GPUs. In section III-C this behavior is confirmed by numerical experiments. If the nonlinearity does include differential operators, such as in Burgers' equation, the evaluation is essentially reduced to sparse-matrix vector multiplication, which we will discuss in some detail in this paper (in the context of stencil methods).

III. Results

A. Stencil methods with boundary conditions

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 Δx 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 [3] 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 [3] and [2]. 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 I. 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 [3] gives 2.5 Gflops/s.

In [3] a double precision performance of 36.5 Gflops/s is reported for a GTX280 of compute 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 and 33.5 Gflops/s for 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 I).

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 [13]). The memory requirements 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.

B. Stencil methods with a position-dependent coefficient

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 $\nabla \cdot (D\nabla u)$). Compared to the previous section we expect that the direct implementation of the position-dependent diffusion coefficient in the CUDA kernel, for a sufficiently complicated D, results in an compute bound problem. For the particular choice of $D(x,y,z) = 1/\sqrt{1+x^2+y^2}$, taken from [11], the results are shown in Table II.

TABLE II
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 I and II are (almost) identical for the $n = 512^3$ case. Thus, for the sake of brevity, we choose to omit those results.

C. Evaluating the nonlinearity on a GPU

For an exponential integrator, usually the most time consuming part is evaluating the exponential and φ_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 [6]). 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 nonlinearity for a number of intermediate approximations. In this section we will be

TABLE I
TIMING OF A SINGLE STENCIL BASED MATRIX-VECTOR COMPUTATION FOR A NUMBER OF IMPLEMENTATIONS AND BOUNDARY CONDITIONS. THE CORRESPONDING GFLOPS/S ARE SHOWN IN PARENTHESES. ALL COMPUTATIONS ARE PERFORMED WITH $n = 256^3$.

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

concerned with the efficient evaluation of the nonlinearity on a GPU.

Since the nonlinearity 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 [14, p. 439])

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

with nonlinearity

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

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 III-A) the parallelization of the nonlinearity 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 III.

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 nonlinearity contributes significantly to the runtime of the program. Let us duly note that the speedups observed here can not be extended to the entire exponential integrator as the sparse-matrix vector multiplication is usually the limiting factor.

TABLE III
TIMING OF A SINGLE COMPUTATION OF THE NONLINEARITY 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

The nonlinearity of certain semi-linear PDEs resemble more the performance characteristics of the stencil methods discussed in sections III-A and III-B. For example, Burgers' equation, where $g(\boldsymbol{u}) = (\boldsymbol{u}\cdot\nabla)\boldsymbol{u}$, falls into this category. Such nonlinearities can be efficiently implemented by the methods discussed in sections III-A and III-B.

If we combine sections III-A, III-B and III-C we have all ingredients necessary to conduct an efficient implementation of exponential integrators on a single GPU. The specific performance characteristics depend on the form of the linear as well as the nonlinear 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.

D. 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 nonlinearity 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 [12] 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 IV, the problem in (4) shows good scaling behavior (at least) up to 4 GPUs.

TABLE IV
PERFORMANCE COMPARISON FOR THE COMBUSTION MODEL DISCUSSED IN SECTION III-C 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., [15])

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

is to be determined, where ψ 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 V. The implementation used is based on the code given in [16]. However, we have found that for the problem under consideration using a full warp for every row of our matrix results in a performance reduction. Therefore, we use only four threads per row which results, for the specific problem under consideration, in a performance increase of approximately 50%, as compared against the CUSPARSE library (see [17]). Apart from this consideration the code has been adapted to compute the problem stated in (3), which includes an additional term as compared to the sparse matrix-vector multiplication considered in [16]. Clearly the scaling behavior

TABLE V
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

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. It should be noted that a GPU centric data format (as discussed in section II-B) could be employed instead of the CSR format. However, also in this case the overhead of copying between different GPUs would persist.

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 V is thus expected.

IV. CONCLUSION & OUTLOOK

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 based 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 III-A shows that for non-homogeneous boundary conditions 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.

The results presented in this paper show that exponential integrators, for many realistic settings, can be efficiently implemented on GPUs with significant speedups compared to more traditional implementations. Therefore, GPU computing provides a viable way to increase the efficiency of simulations in which exponential integrators are employed. The implementation of exponential integrators on the current generation of GPUs would conceivably result in a further performance increase of our memory bound stencil implementation, as compared to the C2075, as the Kepler architecture offers a memory throughput of up to 250 GB/s. Furthermore, such an implementation is expected to be relatively straightforward as the cache implemented on the newer generations of GPUs works quite well in the case of stencil methods.

REFERENCES

[1] L. Murray, "GPU acceleration of Runge-Kutta integrators," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 1, pp. 94–101, 2012.

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

[3] 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*, vol. 69, no. 9, pp. 762–777, 2009.

[4] E. Hairer, S.P. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*, 2nd ed. Springer-Verlag, Berlin, 1993.

[5] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, 2nd ed. Springer-Verlag, Berlin, 1996.

[6] M. Hochbruck and A. Ostermann, "Exponential integrators," *Acta Numer.*, vol. 19, pp. 209–286, 2010. [Online]. Available: http://www.journals.cambridge.org/abstract_S0962492910000048

[7] "CUDA C Programming Guide," http://docs.nvidia.com/cuda/pdf/CUDA_C_Programmin (Last retrieved September 19, 2013).

[8] 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*. New York, USA: ACM Press, 2009, p. 18. [Online]. Available: http://dl.acm.org/citation.cfm?doid=1654059.1654078

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

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

[11] M. Vazquez, A. Berezhkovskii, and L. Dagdug, "Diffusion in linear porous media with periodic entropy barriers: A tube formed by contacting spheres," *J. Chem. Phys*, vol. 129, no. 4, p. 046101, 2008.

[12] M. Caliari, M. Vianello, and L. Bergamaschi, "Interpolating discrete advectionâĂŞ-diffusion propagators at Leja sequences," *J. Comput. Appl. Math.*, vol. 172, no. 1, pp. 79–99, Nov. 2004. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0377042704000998

[13] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA NVR-2008-004, Tech. Rep., 2008.

[14] W. Hundsdorfer and J. Verwer, *Numerical Solution of Time-Dependent Advection-Diffusion-Reaction Equations*, 2nd ed. Springer-Verlag, Berlin, Heidelberg, New York, 2007.

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

[16] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, Dec. 2008.

[17] "CUDA CUSPARSE Library," http://docs.nvidia.com/cuda/pdf/CUDA_CUSPARSE_Us (Last retrieved September 19, 2013).