University of Innsbruck
Department of Mathematics
Numerical Analysis Group

Master Thesis

Exponential integrators on graphic processing units

Author:
Lukas Einkemmer

Advisor:
Dr. Alexander Ostermann

Submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of Diplom-Ingenieur.

July 11, 2011

Acknowledgments

I want to take the opportunity to thank all the people that helped me during the preparation of this master this.

First of all, my supervisor Dr. Alexander Ostermann who drew my interest to the topic of exponential integrators. Especially, I would like to mention his contribution to understanding and translating the proof of Leja regarding the maximal convergence of Leja points.

I would also like to thank Stefan Rainer for proofreading this master thesis and offering suggestions for improvement.

The hardware that allowed me to run quite extensive numerical simulations on multiple graphic processing units has been procured by the working group Numerical Analysis. Without this support it would have been difficult, at best, to test the algorithms given in this master thesis under "real world" conditions. In this regard, I would like to thank Wolfgang Kucher, who handled the setup and administration of before mentioned system.

Last but not least I would also like to thank Martina Prugger for finding countless typos and other mistakes while proofreading this master thesis.

This work was supported by the

Science Fund of the Tiroler Landesregierung (TWF)

under grant GZ: UNI-0404/880

Contents

1 Introduction

From the standpoint of a computer engineer there are (at least) two ways to improve the execution time of an algorithm. First, one might build sequential processing units with increased speed (this is most common in CPUs, although those have also incorporated parallel processing paradigms), while the second alternative is to build a massive number of processing units into a single integrated circuit (this is most common in GPUs). Although this is hardly a new concept (supercomputers were employing thousands of processors for some time) to integrate them on a single chip with a unified global memory is.

From a purely theoretical standpoint, as is taken in complexity theory, a multitude of processing units is not in any way superior to a single processor. However, in practical applications the constants do matter and thus it is a viable option to parallelize an algorithm to different processing units. Nevertheless, from a theoretical standpoint it is interesting to investigate to what extend we can accelerate an algorithm by distributing it to many processing units.

The next theorem (see e.g. [2]) gives us the amount of speed increase, called the **speedup**, we can expect by the parallelization of an algorithm.

Theorem 1. ***(Amdahl's law).*** *Suppose P is the fraction of an algorithm that is parallelizable and N is the number of processing units available, the maximal speedup S is given by*

$$S = \frac{1}{1 - P + \frac{P}{N}}.$$

Proof. Suppose a program takes an amount of time T to complete. Then the (perfectly) parallelized program takes time

$$(1 - P)T + \frac{PT}{N},$$

and therefore

$$S = \frac{T}{(1 - P)T + \frac{PT}{N}} = \frac{1}{1 - P + \frac{P}{N}}.$$

□

It should be duly noted (as is shown in Figure 1) that a parallelization with $P = 1 - \frac{1}{N}$ is not sufficient if a speedup of N is to be achieved by N processing units. This is only possible if $P = 1$ (i.e. the algorithm is perfectly parallelizable). As Figure 1 illustrates to come even close (for $N = 100$) we need an algorithm with $P = 0.999$. Or if we reverse the argument, an infinite number of processing units is needed to speed up an algorithm with $P = 1 - \frac{1}{N}$ by a factor of N.

For some algorithms finding a parallelization with P close to 1 is very natural. For example, Monte Carlo integration requires almost no modification to get P very close to 1. The same is true for most schemes where the evaluation of a function at different (precomputed) points is the most time expensive part of the algorithm. Many algorithms, however, require careful analysis to find a parallelization such that P is large.

We are mainly interested in the scenario where a stiff ordinary differential equation of first order with prescribed initial conditions is given. Often this ordinary differential equation is the result of the space discretization of an evolutionary partial differential equation. The goal is to integrate the equation from $t = 0$ up to a fixed time $t = T$. To that end we use interpolatory exponential integrators (see e.g. [14]), i.e.

1. Compute a number of interpolation nodes (this is a preprocessing step as is explained further in Remark 37).
2. Compute the coefficients of the interpolant (this can also be considered a preprocessing step).

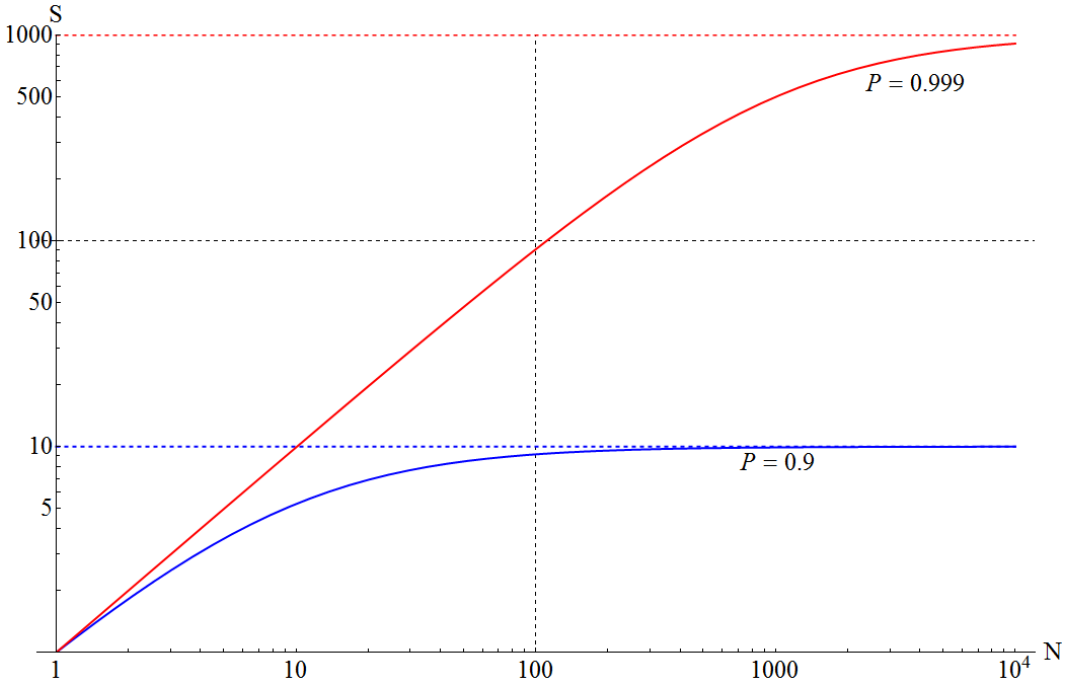

Figure 1: Illustration of Amdahl's law.

3. Use the polynomial interpolant to make an approximate time step.

We will use Newton interpolation and therefore have to multiply a sparse-matrix with a dense-vector as well as add two dense vectors. Matrix multiplication as well as vector addition is, in theory, perfectly parallelizable. This suggests the validity of our approach. However, in practical applications the dimensions of the matrix (and vector) must be large enough so as to make thread switching time a negligible factor. However, as we will show in this master thesis in many instances such matrix-vector multiplications can be parallelized quite well.

2 Theory

In the following section, we will now introduce the theory necessary to prove two central results concerning the interpolation at Leja points on a compact subset of the complex plane. The first result is given in Remark 37 and states that only the computation of Leja points on sets with capacity one is numerically stable. It will also be shown how general sets can be reduced to a set of capacity one for the purpose of such a computation. Second, we will state in corollary 39 that Leja points achieve, in a sense to be made precise, the best possible convergence rate possible for polynomials.

To that end, before we turn to Leja interpolation in section 2.3 we introduce some elements from potential theory (section 2.1) and revisit interpolation theory in section 2.2.

Finally, we turn our attention to extent the results from the complex plane to certain matrices. This is necessary since ultimately our goal is to employ interpolation at Leja points to compute matrix functions necessary in the implementation of exponential integrators. Two types of exponential integrators suitable for the time integration of (discrete) Schrödinger equations and nonlinear evolution equations of certain form are given in sections 2.4 and 2.5 respectively.

2.1 Potential theory

In this section we will introduce concepts from potential theory that we are going to employ heavily in the subsequent sections. The treatment to a large extent follows [5, chap. 1].

Remark 2. In what follows we identify the complex plane $\mathbb{C}$ with $\mathbb{R}^2$ and use the variables (x, y) as a vector in $\mathbb{R}^2$. It is stipulated that the complex variable $z = x+iy$ represents the same point as the vector (x, y) does.

Definition 3. (Positive unit Borel measure).

$$\mathcal{M}(K) = \{\sigma \,\text{Borel measure on } K \,:\, \sigma(K) = 1,\, \sigma(A) \geq 0 \text{ for all } A \in \mathcal{B}(K)\}\,,$$

where $\mathcal{B}(K)$ is the σ-algebra of Borel sets in K.

Definition 4. (Logarithmic potential). Suppose $\sigma \in \mathcal{M}(K)$ and $K \subset\subset \mathbb{R}^2$.[1] Then

$$U^\sigma(z) := \int_K \log \frac{1}{|z-w|}\, \mathrm{d}\sigma(w)$$

is called the l**ogarithmic potential** of σ.

Definition 5. (Capacity/Logarithmic capacity). Suppose $\sigma \in \mathcal{M}(K)$ and $K \subset\subset \mathbb{R}^2$. Then the energy of σ is defined by

$$I(\sigma) := \int U^\sigma(z)\, \mathrm{d}\sigma(z)$$

and we call

$$\gamma(K) := \inf_{\sigma \in \mathcal{M}(K)} I(\sigma),$$

where $\gamma(K) \in \overline{\mathbb{R}}$, the **capacity** of K. Furthermore

$$c(K) := e^{-\gamma(K)}$$

is called the **logarithmic capacity** of K.

[1] $A \subset\subset B$ denotes that A is a compact subset of B.

It should be noted that since we allow $\gamma(K)$ to take values in the extended real number line, the infimum in definition 5 is well defined. Furthermore, for $K = \{a\}$ the unique measure satisfying definition 3 is δ_a and thus we have

$$\gamma(K) = \int\int \frac{1}{|z-w|}\,\mathrm{d}\delta_a(z)\,\mathrm{d}\delta_a(w) = \int \frac{1}{|z-a|}\mathrm{d}\delta_a(z) = \infty. \tag{1}$$

In this case $c(K) = 0$ and $\{I(\sigma)\colon \sigma \in \mathcal{M}(K)\}$ is **not** bounded from above. Furthermore, we can show that if $c(K) > 0$ the infimum is in fact a minimum, as the following theorem states.

Theorem 6. ***(Equilibrium measure).*** *Suppose $K \subset\subset \mathbb{R}^2$ with $c(K) > 0$. Then, there exists a unique measure, denoted by μ_E, such that*

$$\gamma(K) = I(\mu_E).$$

Proof. See e.g. [23, pp. 27-29]. □

Next we show that we can easily scale a set to capacity one by using a linear transformation. As will be shown in theorem 36 this is quite crucial for the application of Leja points being numerically stable.

Theorem 7. *Suppose $r \in \mathbb{C}$ and $a \in \mathbb{R}^2$. Then it holds that*

$$c(rK + a) = |r|c(K).$$

Proof. For $\sigma \in \mathcal{M}(K)$ and $r \neq 0$ we have

$$\begin{aligned} I(\sigma) &= \int U^{\sigma}(z)\,\mathrm{d}\sigma(z) \\ &= \int\int \log\frac{1}{|z-w|}\,\mathrm{d}\sigma(z)\,\mathrm{d}\sigma(w) \\ &= \int\int \log\frac{1}{|z-w|} + \log|r|\,\mathrm{d}\tilde{\sigma}(z)\,\mathrm{d}\tilde{\sigma}(w) \\ &= I(\tilde{\sigma}) + \log|r|, \end{aligned}$$

where $\tilde{\sigma} \in \mathcal{M}(rK + a)$. Therefore, $\gamma(K) = \gamma(rK + a) + \log|r|$ and our result follows from definition 5. The case $r = 0$ follows from equation 1. □

The following characterization for the logarithmic potential is immensely useful in the calculations of the capacities that are done in this section.

Theorem 8. *Suppose $K \subset\subset \mathbb{R}^2$ and $c(K) > 0$. Then for all $\sigma \in \mathcal{M}(K)$ it holds that*

$$\inf_{z\in K} U^{\sigma}(z) \leq \gamma(K) \leq \sup_{z\in K} U^{\sigma}(z).$$

Proof. See e.g. [5, p. 11]. □

We will now calculate the capacity of a circle and the capacity of an interval in the following two examples. The calculation closely follows [5, pp. 12-13].

Example 9. (Capacity of a circle). We consider $\overline{B_1(0)} \subset\subset \mathbb{R}^2$ and the measure $\mathrm{d}\sigma = \frac{\mathrm{d}s}{2\pi}$, where $\mathrm{d}s$ is the usual measure on the boundary of the unit circle. Then

$$\begin{aligned} U^{\sigma}(z) &= \int_{\partial B_1(0)} \log\frac{1}{|z-w|}\,\frac{\mathrm{d}w}{2\pi} \\ &= \begin{cases} \log\frac{1}{|z|} & |z| > 1 \\ 0 & |z| \leq 1 \end{cases}, \end{aligned}$$

since for $|z| > 1$ the function $w \mapsto \log \frac{1}{|z-w|}$ is harmonic for $|w| \leq 1$. In addition, for $|z| < 1$ the function

$$w \mapsto \log \frac{1}{|z-w|} - \log \frac{1}{|w|}$$

is harmonic for $|w| \geq 1$. From the mean value property[2] our result follows.
For $|z| = 1$ we have (take w.l.o.g. $z = 1$)

$$\begin{aligned} U^{\sigma}(z) &= \frac{1}{2\pi} \int_0^{2\pi} \log \frac{1}{|1 - e^{i\phi}|} \, \mathrm{d}\phi \\ &= \frac{1}{2\pi} \int_0^{2\pi} \log \frac{1}{\sqrt{2 - 2\cos\phi}} \, \mathrm{d}\phi \\ &= 0. \end{aligned}$$

The integral is nontrivial but can be Risch integrated by means of a dilogarithm and elementary functions.
By theorem 8 we follow that $\gamma(B_1(0)) = 0$ and thus $c(B_1(0)) = 1$. In addition, by theorem 7 we have

$$c(B_r(a)) = r.$$

The capacity of a circle is therefore equal to its radius.

Example 10. (Capacity of an interval). We consider $K = [-1, 1] \subset\subset \mathbb{C}$ and denote the Joukowski map by

$$\psi(\zeta) := \frac{1}{2} \left(\zeta + \zeta^{-1} \right).$$

The Joukowski map maps $\mathbb{C} \backslash B_1(0)$ conformally to $\mathbb{C} \backslash [-1, 1]$. The measure

$$\mathrm{d}\tilde{\sigma}(w) := \begin{cases} \frac{1}{\pi} \frac{\mathrm{d}w}{\sqrt{1-x^2}} & w \in [-1, 1] \\ 0 & \text{otherwise} \end{cases}$$

yields the logarithmic potential

$$U^{\tilde{\sigma}}(z) = \frac{1}{\pi} \int_{-1}^{1} \log \frac{1}{|z-w|} \frac{\mathrm{d}w}{\sqrt{1-w^2}}.$$

By substituting $w = \psi(e^{i\phi}) = \cos\phi$ and therefore $\mathrm{d}w = -\sin\phi \, \mathrm{d}\phi$ we have (for $z = \psi(\zeta)$)

$$\begin{aligned} U^{\tilde{\sigma}}(z) &= \frac{1}{2\pi} \int_0^{2\pi} \log \frac{1}{|z - \psi(e^{i\phi})|} \, \mathrm{d}\phi \\ &= \frac{1}{2\pi} \int_0^{2\pi} \log \frac{1}{|\psi(\zeta) - \psi(e^{i\phi})|} \, \mathrm{d}\phi \\ &= \frac{1}{2\pi} \int_0^{2\pi} \log \left| \frac{\zeta - e^{i\phi}}{\psi(\zeta) - \psi(e^{i\phi})} \right| \mathrm{d}\phi + \frac{1}{2\pi} \int_0^{2\pi} \log \frac{1}{|\zeta - e^{i\phi}|} \, \mathrm{d}\phi \\ &= -\log \left| \psi^{-1}(z) \right| - \log \frac{1}{2}, \end{aligned}$$

since

$$\frac{1}{2\pi} \int_0^{2\pi} \log \frac{1}{|\zeta - e^{i\phi}|} \, \mathrm{d}\phi = \log \frac{1}{|\zeta|} = \log \frac{1}{|\psi^{-1}(z)|}$$

[2] Note that the mean value property also holds in $\mathbb{R}^2 \cup \{\infty\}$.

and by the harmonicity of

$$h(w) := \log \left| \frac{\zeta - w}{\psi(\zeta) - \psi(w)} \right|$$

in $\overline{\mathbb{C}} \backslash B_1(0)$ we have

$$\frac{1}{2\pi} \int_0^{2\pi} \log \left| \frac{\zeta - e^{i\phi}}{\psi(\zeta) - \psi(e^{i\phi})} \right| \mathrm{d}\phi = h(\infty) = \log 2.$$

Thus for $z \in [-1, 1]$ (since $U^{\tilde{\sigma}}$ is lower semi-continuous)

$$U^{\tilde{\sigma}}(z) \leq -\log \frac{1}{2}$$

and

$$\liminf_{z \to [-1,1], z \in \mathbb{C} \backslash [-1,1]} U^{\tilde{\sigma}}(z) = -\log \frac{1}{2}.$$

Therefore, theorem 8 yields

$$c([-1, 1]) = \frac{1}{2},$$

or more generally (by theorem 7)

$$c([a, b]) = \frac{b - a}{4}.$$

In the following we will on several occasions need the maximum principle (or to be more specific the maximum and minimum principle simultaneously applied to a harmonic function). For completeness we will state the maximum principle that is used in the following (the formulation is similar to [23, p. 39]).

Definition 11. (Quasi everywhere). We say that a property holds **q.e.** (quasi everywhere) if it holds except on a set of capacity zero.

Theorem 12. ***(Maximum principle).*** *Suppose $g : \Omega \to \mathbb{R}$ is subharmonic and bounded from above, where $\Omega \subset \overline{\mathbb{C}}$ is a domain (i.e. open and connected). In addition, assume that*

$$\limsup_{z \to \zeta,\, z \in \Omega} g(z) \leq 0,$$

for q.e. $\zeta \in \partial\Omega$. Then

$$\forall z \in \Omega \colon g(z) \leq 0.$$

Proof. Follows from the result in [23, p. 39]. □

Corollary 13. *Suppose $g : \Omega \to \mathbb{R}$ is subharmonic and bounded, where $\Omega \subset \overline{\mathbb{C}}$ is a domain. In addition, assume that*

$$\lim_{z \to \zeta,\, z \in \Omega} g(z) = 0,$$

for q.e. $\zeta \in \partial\Omega$. Then

$$\forall z \in \Omega \colon g(z) = 0.$$

Proof. Follows from theorem 12. □

Next let us define the Green's function of $\mathbb{R}^2 \backslash K$ with pole at infinity.

Definition 14. (Green's function with pole at infinity). Suppose $K \subset\subset \mathbb{R}^2$. Then we call $G(z)$ a **Green's function with pole at infinity** if

1. $G(z)$ is non-negative and harmonic in $\mathbb{R}^2 \backslash K$.

2. $\lim_{z\to\infty}(G(z)-\log|z|)=\gamma(K)$.
3. $\lim_{z\to\zeta}G(z)=0$ for $\zeta\in\partial K$.

One might ask at this point why we require that $\lim_{z\to\infty}(G(z)-\log|z|)$ is to be equal to $\gamma(K)$. However, as is shown in the next theorem, to require that $G(z)-\log|z|$ in any neighborhood of infinity to be bounded is enough to prove that it converges to $\gamma(K)$. This theorem is important, since in section 2.2 we characterize maximal convergence by employing the before mentioned limit. Therefore, it is interesting to know that the Green's function, even under substantially weaker conditions, is unique for a given set K (a similar treatment is given in [23, pp. 108-109]).

Theorem 15. *Suppose $K\subset\subset\mathbb{R}^2$ and $\tilde{G}(z)$ is a function such that*

1. *$\tilde{G}(z)$ is harmonic in $\mathbb{R}^2\backslash K$.*
2. *$\tilde{G}(z)-\log|z|$ is bounded in any neighborhood of ∞.*
3. *$\lim_{z\to\zeta}\tilde{G}(z)=0$ for $\zeta\in\partial K$.*

Suppose there exists a Green's function with pole at infinity $G(z)$. Then we can conclude that $\tilde{G}(z)$ is a Green's function with pole at infinity.

Proof. We consider $H(z):=\tilde{G}(z)-G(z)$. The function H is harmonic in $\mathbb{R}^2\backslash K$ such that

$$\lim_{z\to\zeta}H(z)=0,$$

except for $\zeta=\infty$. However, this is a set of capacity zero and since

$$H(z)=G(z)-\log|z|-\tilde{G}(z)+\log|z|,$$

we can conclude that H is bounded on $\overline{\mathbb{R}^2}\backslash K$. Therefore, by the maximum principle we conclude that $H(z)=0$ and thus that $\tilde{G}(z)=G(z)$, as desired. □

It is often convenient to work with either a Green's function (i.e. in the setting of potential theory) or a conformal map (in the setting of complex analysis). The following theorem provides a link between the existence of a Green's function and that of a conformal map. Most results that follow can be formulated either in the language of potential theory or in the language of complex analysis.

Theorem 16. *Suppose $K\subset\subset\mathbb{R}^2$. Then there exists a Green's function iff there exists a conformal map $\phi:\mathbb{C}\backslash K\to\mathbb{C}\backslash B_1(0)$ such that*

$$\lim_{z\to\infty}\left|\frac{\phi(z)}{z}\right|=\frac{1}{c(K)}.$$

Proof. Suppose $G(z)$ is a Green's function with pole at infinity. Then

$$z\mapsto G(z)-\log|z|$$

is harmonic in $\overline{\mathbb{C}}\backslash K$ (since at infinity we have a removable singularity). Let us denote the harmonic conjugate of the before mentioned function by $H(z)$. Then

$$z\mapsto e^{G(z)+iH(z)}e^{-\log|z|}$$

is holomorphic in $\overline{\mathbb{C}}\backslash K$ and by multiplying with z (which is holomorphic except at infinity) we get

$$\phi(z):=e^{G(z)+iH(z)}e^{-\log|z|}z.$$

The function ϕ is clearly holomorphic in $\mathbb{C}\backslash K$.

In addition, since for $z \in \overline{\mathbb{C}}\backslash K$

$$|\phi(z)| = e^{G(z)} \geq 1,$$

with equality if and only if $|z| = 1$, ϕ is well defined. It is also true that ϕ is conformal and surjective (see e.g. [24]).

Furthermore,

$$\lim_{z\to\infty} \left|\frac{\phi(z)}{z}\right| = \lim_{z\to\infty} \left|e^{G(z)-\log|z|+iH(z)}\right| = \lim_{z\to\infty} \left|\frac{e^{G(z)}}{z}\right| = \lim_{z\to\infty} e^{G(z)-\log|z|} = e^{\gamma(K)} = \frac{1}{c(K)}.$$

In the other direction we assume the existence of a conformal map ϕ and have to show that

$$G(z) = \log|\phi(z)|$$

is a Green's function with pole at infinity. Thus, let us verify the conditions given in definition 14.

1. Clear.
2. We have
$$\lim_{z\to\infty} (G(z) - \log|z|) = \lim_{z\to\infty} \log\left|\frac{\phi(z)}{z}\right| = \log\frac{1}{c(K)} = \gamma(K).$$
3. We have
$$\lim_{z\to\zeta} G(z) = 0.$$

□

Clearly, it remains to show that for the sets considered in examples 9 and 10 the conditions of theorem 16 hold, i.e. that their exists a Green's function. The next two examples will give an explicit formula for a Green's function on those sets.

Example 17. For $K = B_r(0)$ we get the candidate

$$G(z) := \log|z| + \gamma(K) = \log|z| - \log r$$

from the calculation done in example 9. It is easy to verify that $G(z)$ as defined above is indeed a Green's function.

1. Clear.
2. We have
$$\lim_{z\to\infty} (G(z) - \log|z|) = -\log r = \gamma(K).$$
3. We have
$$\lim_{z\to re^{i\phi}} G(z) = 0,$$
for all $\phi \in [0, 2\pi)$.

Example 18. For $K = [a, b]$ we get the candidate

$$G(z) := \log\left|\psi^{-1}\left(\frac{2}{b-a}\left[z - \frac{a+b}{2}\right]\right)\right|$$

from the calculations done in example 10. It is easy to verify that $G(z)$ as defined above is indeed a Green's function.

1. Clear.
2. We have
$$\lim_{z\to\infty}\left(G(z)-\log|z|\right)=\lim_{z\to\infty}\log\left|\frac{\psi^{-1}\left(\frac{2}{b-a}\left[z-\frac{a+b}{2}\right]\right)}{z}\right|=-\log\frac{b-a}{4}=\gamma(K).$$
3. We have
$$\lim_{z\to\zeta,\,\zeta\in[a,b]}G(z)=\lim_{z\to\zeta,\,\zeta\in[-1,1]}\log\left|\psi^{-1}(z)\right|=0$$
by the same argument as in example 10.

2.2 Interpolation theory

Let us first reproduce the following results from interpolation theory (see e.g. [25, chap. 2]).

Theorem 19. ***(Newton interpolation form).*** *Suppose $((z_i,y_i))_{i=0}^n\in\mathbb{C}^2$ is such that $(z_i)_{i=0}^n$ are pairwise different. Then there exists a polynomial $p_n\in\Pi_n:=\{f\in\mathbb{C}[z]:\deg f\le n\}$ such that*
$$p_n(z_i)=y_i.$$
This polynomial is unique and can be written in ***Newton form*** *as follows*
$$p_n(z)=\sum_{i=0}^n[y_0,\dots,y_i]\omega_i(z),$$
where
$$\omega_i(z)=\prod_{j=0}^{i-1}(z-z_j)$$
are called the ***Newton basis polynomials*** *and the coefficients are called* ***divided differences*** *and are given by the recursion*
$$\begin{aligned}[y_i]&:=y_i\\ [y_i,\dots,y_j]&:=\frac{[y_{i+1},\dots,y_j]-[y_i,\dots,y_{j-1}]}{z_j-z_i}.\end{aligned}$$

Proof. See [25, chap. 2]. □

The following explicit representation of the divided differences will be needed later in this section.

Theorem 20. *Suppose $((z_i,y_i))_{i=0}^n\in\mathbb{C}^2$ is such that $(z_i)_{i=0}^n$ are pairwise different, then it holds that*
$$[y_0,\dots,y_j]=\sum_{m=0}^j\frac{y_m}{\prod_{k\in\{0,\dots,j\}\setminus\{m\}}(z_m-z_k)}.$$

Proof. Essentially we have to verify the recursion relation in theorem 19 for the representation given. For $[y_i]$ the result is straightforward. Now consider
$$\begin{aligned}\frac{[y_{i+1},\dots,y_j]-[y_i,\dots,y_{j-1}]}{z_j-z_i}&=\frac{1}{z_j-z_i}\left[\sum_{m=i+1}^j\frac{y_m}{\prod_{k\in\{i+1,\dots,j\}\setminus\{m\}}(z_m-z_k)}-\sum_{m=i}^{j-1}\frac{y_m}{\prod_{k\in\{i,\dots,j-1\}\setminus\{m\}}(z_m-z_k)}\right]\\ &=\frac{y_j}{\prod_{k\in\{i,\dots,j\}\setminus\{j\}}(z_j-z_k)}+\frac{y_i}{\prod_{k\in\{i,\dots,j\}\setminus\{i\}}(z_i-z_k)}+\\ &\sum_{m=i+1}^{j-1}\frac{1}{z_j-z_i}\left[\frac{1}{z_m-z_j}-\frac{1}{z_m-z_i}\right]\frac{y_m}{\prod_{k\in\{i+1,\dots,j-1\}\setminus\{m\}}(z_m-z_k)}\\ &=[y_i,\dots,y_j],\end{aligned}$$

since

$$\frac{1}{z_j - z_i}\left[\frac{1}{z_m - z_j} - \frac{1}{z_m - z_i}\right] = \frac{1}{(z_m - z_j)(z_m - z_i)}.$$

□

From this we can derive a single-term recurrence relation that is most useful for numerical implementations.

Theorem 21. *Suppose* $((z_i, y_i))_{i=0}^n \in \mathbb{C}^2$ *is such that* $(z_i)_{i=0}^n$ *are pairwise different, then it holds that*

$$[y_0, \ldots, y_j] = d_j^{(j)},$$

where

$$\begin{aligned} d_0^{(j)} &= y_j, \\ d_{i+1}^{(j)} &= \frac{d_i^{(j)} - [y_0, \ldots, y_i]}{z_j - z_i}. \end{aligned}$$

Proof. We have

$$\begin{aligned} d_j^{(j)} &= \frac{y_j}{\prod_{k\in\{0,\ldots,j-1\}}(z_j - z_k)} - \sum_{i=0}^{j-1} \frac{[y_0, \ldots, y_i]}{\prod_{k\in\{i,\ldots j-1\}}(z_j - z_k)} \\ &= \frac{y_j}{\prod_{k\in\{0,\ldots,j-1\}}(z_j - z_k)} - \sum_{i=0}^{j-1}\sum_{m=0}^{i} \frac{y_m}{\prod_{l\in\{0,\ldots,i\}\setminus\{m\}}(z_m - z_l)\prod_{k\in\{i,\ldots j-1\}}(z_j - z_k)} \\ &= \frac{y_j}{\prod_{k\in\{0,\ldots,j-1\}}(z_j - z_k)} - \sum_{m=0}^{j-1} y_m \sum_{i=m}^{j-1} \frac{1}{\prod_{l\in\{0,\ldots,i\}\setminus\{m\}}(z_m - z_l)\prod_{k\in\{i,\ldots j-1\}}(z_j - z_k)} \\ &= \frac{y_j}{\prod_{k\in\{0,\ldots,j-1\}}(z_j - z_k)} + \sum_{m=0}^{j-1} \frac{y_m}{\prod_{k\in\{0,\ldots j\}\setminus\{m\}}(z_m - z_k)} \\ &= [y_0, \ldots, y_j], \end{aligned}$$

where we employed the following calculation

$$\begin{aligned} &\sum_{i=m}^{j-1} \frac{1}{\prod_{l\in\{0,\ldots,i\}\setminus\{m\}}(z_m - z_l)\prod_{k\in\{i,\ldots j-1\}}(z_j - z_k)} \\ &= \frac{1}{\prod_{k\in\{0,\ldots j\}\setminus\{m\}}(z_m - z_k)} \sum_{i=m}^{j-1} \frac{\prod_{k\in\{i+1,\ldots j\}\setminus\{m\}}(z_m - z_k)}{\prod_{k\in\{i,\ldots j-1\}}(z_j - z_k)} \\ &= \frac{-1}{\prod_{k\in\{0,\ldots j\}\setminus\{m\}}(z_m - z_k)}. \end{aligned}$$

□

Now we consider the interpolation polynomial as an approximation to a continuous function on a compact set (in the complex plane).

Theorem 22. *Suppose* $K \subset\subset \mathbb{C}$ *and that* $f : K \to \mathbb{C}$. *For* $z \in K$ *we get*

$$f(z) - p_n(z) = f[z_0, \ldots, z_n, z]\omega_{n+1}(z),$$

where

$$f[z_0, \ldots, z_n, z] := [f(z_0), \ldots, f(z_n), f(z)].$$

Proof. By theorem 19 we know that

$$f(z) = p_n(z) + f\,[z_0, \ldots, z_n, z]\,\omega_{n+1}(z),$$

from which the result readily follows. □

On the real line (i.e. $z_i \in [a,b]$ for all i and $z \in [a,b]$) we can use the fact that $f[z_0, \ldots, z_n, z] = \frac{f^{(n+1)}(\xi)}{(n+1)!}$ holds for some $\xi \in [a,b]$ (see e.g. [25, p. 50]) to get an error bound for an $n+1$ times differentiable function. However for our purpose we will use the above result in a more subtle way. We will derive a representation of the interpolation error that is due to Hermite (as cited in [11]) and allows us to determine, in some sense, the best possible convergence rate of polynomial interpolation. This approach can be found, for example, in in [12] or [11].

Theorem 23. *Suppose $K \subset\subset \mathbb{C}$ and $f : K \to \mathbb{C}$ is a holomorphic function. For $z \in K$ we get*

$$f(z) - p_n(z) = \frac{1}{2\pi i} \int_\gamma \frac{\omega_{n+1}(z)}{\omega_{n+1}(\xi)} \frac{f(\xi)}{\xi - z}\,\mathrm{d}\xi,$$

if γ is a Jordan curve with winding number 1, such that $z_0, \ldots, z_n$ lie in the interior of γ.

Proof. By the residue theorem

$$\begin{aligned}\frac{1}{2\pi i} \int_\gamma \frac{\omega_{n+1}(z)}{\omega_{n+1}(\xi)} \frac{f(\xi)}{\xi - z}\,\mathrm{d}\xi &= f(z) + \sum_{i=0}^{n} \frac{\omega_{n+1}(z) f(z_i)}{(z_i - z)\prod_{k\in\{0,\ldots,n\}\setminus\{i\}}(z_i - z_k)} \\ &= f[z_0, \ldots, z_n, z]\omega_{n+1}(z) \\ &= f(z) - p_n(z),\end{aligned}$$

where the last equality follows from theorem 20 and the fact that

$$f[z_0, \ldots, z_n, z] = \sum_{i=0}^{n} \frac{f(z_i)}{(z_i - z)\prod_{k\in\{0,\ldots,n\}\setminus\{i\}}(z_i - z_k)} + \frac{f(z)}{\omega_{n+1}(z)}.$$

□

In light of theorem 23 our goal is to minimize the right hand side of

$$|f(z) - p_n(z)| \le C \frac{|\omega_{n+1}(z)| \sup_{\xi\in\gamma} |f(\xi)|}{\inf_{\xi\in\gamma} |\omega_{n+1}(\xi)|},$$

i.e. to find interpolation points such that $\omega_n(z)$ is small in K but grows rapidly outside of K (or at least is large on some curve γ outside of K). To accomplish this we can cast the problem either in the language of complex analysis and the study of conformal maps (for a summary see e.g. [9]) or into the language of potential theory (see e.g. [11] or [12]). We will choose the latter, however, as theorem 16 asserts, in the complex plane the two approaches are equivalent.

Definition 24. (Zero counting measure). Suppose $n \in \mathbb{N}$ and let $(z_i)_{i=0}^n \in \mathbb{C}$. Then we call

$$\tau_n := \frac{1}{n+1} \sum_{i=0}^{n} \delta_{z_i}$$

the **zero counting measure**.

Theorem 25. *Suppose τ_n is the zero counting measure. Then it holds that*

$$U^{\tau_n}(z) = -\frac{1}{n+1} \log |\omega_{n+1}(z)|.$$

Proof. A simple calculation shows that

$$U^{\tau_n}(z) = \int \log \frac{1}{|z-w|} \, \mathrm{d}\tau_n(w) = \frac{1}{n+1} \sum_{i=0}^{n} \log \frac{1}{|z-z_i|} = -\frac{1}{n+1} \log |\omega_{n+1}(z)|.$$

□

Definition 26. (Level sets). Suppose $\sigma \in \mathcal{M}(K)$. Then we call

$$\mathcal{E}_\lambda^\sigma := \{z \in \mathbb{C} \, : \, U^\sigma(z) \geq -\log \lambda\}$$

the **level set** of σ with respect to the value λ.

Theorem 27. *Suppose $K \subset\subset \mathbb{C}$ and $\sigma \in \mathcal{M}(K)$. Then U^σ attains a minimum and we have*

$$\min_{z \in K} U^\sigma(z) = -\log \lambda_0,$$

where

$$\lambda_0 = \inf \{\lambda \, : \, K \subset \mathcal{E}_\lambda^\sigma\}.$$

Proof. Since K is a compact set and U^σ is superharmonic (see [13, p. 128]) it follows that U^σ attains a minimum on K. The second statement follows from the definition of $\mathcal{E}_\lambda^\sigma$. □

We recall the following definition from measure theory.

Definition 28. Suppose $(\sigma_i)_{i=0}^\infty$ is a sequence of measures. We say σ_i **converges weakly to** σ (or $\sigma_i \xrightarrow{w} \sigma$) if

$$\forall f \in \mathcal{C}^0(\mathbb{C}) \, : \, \int f(z) \, \mathrm{d}\sigma_i(z) \to \int f(z) \, \mathrm{d}\sigma(z).$$

Theorem 29. *Suppose $(z_i)_{i=0}^\infty \in K$ is a sequence of interpolation points, where $K \subset\subset \mathbb{C}$. In addition, suppose $\lambda > \lambda_0$, $\tau_n \to \sigma$ weakly and that f is holomorphic in an open neighborhood U of $\mathcal{E}_\lambda^\sigma$ and bounded on U. Furthermore, let us denote the interpolation polynomial of f at the nodes $(z_i)_{i=0}^n$ by p_n. Then,*

$$\limsup_{n \to \infty} \|f - p_n\|_K^{1/n} \leq \frac{\lambda_0}{\lambda}.$$

Proof. Our aim is to bound

$$\limsup_{n \to \infty} \|f - p_n\|_K \leq \limsup_{n \to \infty} C \frac{|\omega_{n+1}(z)| \sup_{\xi \in \gamma} |f(\xi)|}{\inf_{\xi \in \gamma} |\omega_{n+1}(\xi)|} \leq \limsup_{n \to \infty} C \|f\|_U \sup_{\xi \in \gamma} \frac{\|\omega_{n+1}(z)\|_K}{|\omega_{n+1}(\xi)|},$$

where γ is a Jordan curve with winding number 1 that encloses $\mathcal{E}_\lambda^\sigma$.

First, since $\tau_n \xrightarrow{w} \sigma$ we have

$$\lim_{n \to \infty} \frac{1}{n+1} \log |\omega_n(z)| = -U^\sigma(z) \geq \min_{z \in \gamma} -U^\sigma(z) > \log \lambda.$$

Second, for $\eta > 0$

$$\limsup_{n \to \infty} \frac{1}{n+1} \log \|\omega_n\|_K \leq \limsup_{n \to \infty} \sup_{z \in \partial \mathcal{E}_{\lambda_0+\eta}^\sigma} \frac{1}{n+1} \log |\omega_n(z)| = \log(\lambda_0 + \eta),$$

by the maximum principle (U^σ is superharmonic) and the definition of $\mathcal{E}_{\lambda_0+\eta}^\sigma$. Combining these two, we get

$$\limsup_{n \to \infty} \sup_{\xi \in \gamma} \frac{\|\omega_{n+1}(z)\|_K}{|\omega_{n+1}(\xi)|} \leq \left(\frac{\lambda_0 + \eta}{\lambda} \right)^{n+1}$$

and therefore (since $\eta > 0$ is arbitrary)

$$\limsup_{n\to\infty} \|f - p_n\|_K^{1/n} \leq \frac{\lambda_0}{\lambda}.$$

□

Now our goal is to find a measure such that the quotient $\frac{\lambda_0}{\lambda}$ is minimized. In what follows, we will show that the equilibrium measure μ_E (introduced in theorem 6) satisfies this condition.

Theorem 30. *Suppose $z \in \mathbb{C}\backslash K$ and $G(z)$ is a Green's function with pole at infinity. Then*

$$U^{\mu_E}(z) = -G(z) + \gamma(K).$$

Proof. See e.g. [6, pp. 150-151]. □

Definition 31. (Level sets of G). Suppose $K \subset\subset \mathbb{C}$ and $G(z)$ is a Green's function with pole at infinity. Then we define the **level sets of** G by

$$E_\lambda := \{z \in \Omega : G(z) \leq \log \lambda\} \cup K.$$

Theorem 32. *Let $(z_i)_{i=0}^n \in K$ be a sequence of interpolation points, where $K \subset\subset \mathbb{C}$. Let λ_0 be defined as in theorem 27. In addition, suppose $\lambda > \lambda_0$, $\tau_n \xrightarrow{w} \mu_E$ and f is holomorphic in an open neighborhood U of E_λ and bounded in U. Furthermore, let us denote the interpolation polynomial of f at the nodes $(z_i)_{i=0}^n$ by p_n. Then,*

$$\limsup_{n\to\infty} \|f - p_n\|_K^{1/n} \leq \frac{1}{\lambda}$$

Proof. Follows from the proof of theorem 29. □

Theorem 33. *The estimate in theorem 32 is optimal (in the sense that μ_E gives a better estimate than any other limit distribution).*

Proof. We show that for all $\lambda \geq \lambda_0$ it holds that $E_{\lambda/\lambda_0} \subset \mathcal{E}_\lambda^\sigma$. This clearly shows the optimality of the λ in theorem 32. To that end, we know that

$$-U^\sigma(z) \leq \log \lambda_0$$

and thus

$$-U^\sigma(z) - \log \lambda_0 \leq 0.$$

Since $G(z) \geq 0$ (by definition) we have (for all $z \in \Omega$)

$$-U^\sigma(z) - \log \lambda_0 \leq G(z)$$

and therefore $E_{\lambda/\lambda_0} \subset \mathcal{E}_\lambda^\sigma$, as desired. □

Therefore let us make the following definition.

Definition 34. (Maximally convergent). Let $(z_i)_{i=0}^\infty \in K$ be a sequence of interpolation points, where $K \subset\subset \mathbb{C}$. We call $(z_i)_{i=0}^\infty$ **maximally convergent** if for $(z_i)_{i=0}^n$ the estimate in theorem 32 holds for all n.

It is clear from theorem 30 that we can verify maximal convergence by verifying that

$$U^{\tau_n}(z) \to U^{\mu_E}(z) = -G(z) + \gamma(K),$$

i.e. the logarithmic potential of the zero counting measures converges to the logarithmic potential of the equilibrium measure.

2.3 Leja points

In this section we will introduce the so called Leja points that fulfill most of the desirable properties given in the previous section. In addition it is (contrary to, for example, the Chebyshev nodes) possible to dynamically change the number of interpolation points used.

Definition 35. (Leja points). Suppose $K \subset\subset \mathbb{C}$. The sequence $(z_n)_{n=0}^{\infty}$ given by $|z_0| = \max_{z\in K} |z|$ and

$$\prod_{k=0}^{n-1} |z_n - z_k| = \max_{z\in K} \prod_{k=0}^{n-1} |z - z_k| .$$

is called a sequence of Leja points.

The Leja points are not unique since different choices of z_0 give different sequences of Leja points. Even if z_0 is fixed, their uniqueness is not assured. The interval $K = [-2, 2]$ with $|z_0| = 2$ is a counter-example to this proposition, since we have

$$\begin{aligned}
&\max_{z\in K} |z-2| = 4 \Rightarrow z_1 = -2 \\
&\max_{z\in K} |z-2||z+2| = 4 \Rightarrow z_2 = 0 \\
&\max_{z\in K} |z-2||z+2||z| = \frac{16}{9}\sqrt{3}.
\end{aligned}$$

However, the function to be optimized in the last line is even and therefore we have two solutions, namely

$$z_3 = \frac{2}{3}\sqrt{3} \vee z_3 = -\frac{2}{3}\sqrt{3}.$$

Next, let us turn to the connection of the capacity of a compact set with a sequence of Leja points on that set. The following theorem (see [22]) gives us the asymptotic behavior of the function that is to be maximized in order to compute a sequence of Leja points.

Theorem 36. *Suppose $(z_i)_{i=0}^{\infty}$ is a sequence of Leja points and $K \subset\subset \mathbb{C}$ such that a conformal map as in theorem 16 exists. Then*

$$\prod_{j=0}^{n-1} |z_n - z_j| \geq c(K)^n$$

and

$$\lim_{n\to\infty} \prod_{j=0}^{n-1} |z_n - z_j|^{1/n} = c(K).$$

Proof. The function (where ϕ is constructed in theorem 16)

$$\psi(z) := \prod_{j=0}^{n-1} \frac{z - z_j}{\phi(z)}$$

is holomorphic (in $\overline{\mathbb{C}}\backslash K$) and since

$$\lim_{z\to\infty} |\psi(z)| = \lim_{z\to\infty} \prod_{j=0}^{n-1} \frac{|z - z_j|}{|\phi(z)|} = c(K)^n$$

by the maximum principle in $\overline{\mathbb{C}}\backslash K$ we have

$$\prod_{j=0}^{n-1} |z_n - z_j| = \max_{z\in\partial K} \prod_{j=0}^{n-1} |z - z_j| = \max_{z\in\partial K} |\psi| \geq c(K)^n.$$

For the second result we refer to [17]. □

Remark 37. From the above result it is clear that if $c(K)$ is not equal to one, the product

$$\prod_{j=0}^{n-1} |z_n - z_j|$$

either grows or decays exponentially (in the number of Leja points n). That is the reason why we scale our function $f(z)$ such that it is evaluated in the interval $[-2, 2]$ or some other set of capacity one. In addition, the Leja points on that set can then be precomputed once and for all (and therefore do not enter into runtime considerations).

Now, let us verify that the Leja points are maximally convergent. The proof given is inspired by the french paper [17]. Since its writing, it has been cited many times (for example in [22] and [9]), however, to the best of our knowledge no proof written in English exists. In the following theorem, we will correct this.

Theorem 38. *Suppose $(z_n)_{n=0}^{\infty}$ is a sequence of Leja points. Then it holds that*

$$U^{\tau_n}(z) \to -G(z) + \gamma(K).$$

Proof. We proof the result in four steps. First, let us define

$$V\left(\zeta^{(n)}\right) := \prod_{0 \leq j < k \leq n} |\zeta_j^{(n)} - \zeta_k^{(n)}|,$$

where $\zeta^{(n)} = \left(\zeta_0^{(n)}, \ldots, \zeta_n^{(n)}\right) \in K^{n+1}$ is arbitrary. We choose $\eta^{(n)} \in K^{n+1}$ such that

$$V\left(\eta^{(n)}\right) = \max_{\zeta^{(n)} \in K^{n+1}} V\left(\zeta^{(n)}\right). \tag{2}$$

Obviously we can order the points in $\eta^{(n)}$ such that for

$$\Delta_j^{(n)} := \prod_{\substack{k=0 \\ k \neq j}}^{n} \left|\eta_j^{(n)} - \eta_k^{(n)}\right|, \qquad j \in \{0, \ldots, n\}$$

it holds that

$$\Delta_0^{(n)} \leq \Delta_j^{(n)},$$

for all $j \in \{1, \ldots, n\}$.

Second, consider the Lagrange basis polynomials given by

$$L_j^{(n)}(z) := \prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{z - \eta_k^{(n)}}{\eta_j^{(n)} - \eta_k^{(n)}}, \qquad j \in \{0, \ldots, n\}.$$

From equation 2 we get

$$V\left(\eta_0^{(n)}, \ldots, \eta_{j-1}^{(n)}, z, \eta_{j+1}^{(n)}, \ldots, \eta_n^{(n)}\right) \leq V\left(\eta^{(n)}\right)$$

and thus

$$\left|L_j^{(n)}(z)\right| = \prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{\left|z - \eta_k^{(n)}\right|}{\left|\eta_j^{(n)} - \eta_k^{(n)}\right|} \leq 1.$$

Therefore, we can apply the result given in [16] and [10] (as cited in [17]) to get

$$\lim_{n\to\infty}\left(\sum_{j=0}^{n}|L_j^{(n)}|\right)^{1/n} = \lim_{n\to\infty}\left|L_0^{(n)}(z)\right|^{1/n} = e^{G(Z)},$$

where $G(z)$ is a Green's function with pole at infinity.

Third, we write

$$\omega_{n+1}(z) = \sum_{i=0}^{n}\omega_{n+1}\left(\eta_j^{(n)}\right)L_j^{(n)}(z).$$

This is true since $\omega_{n+1}(z)$ is a polynomial of degree n. From this, we follow from definition 35 that

$$|\omega_{n+1}(z)| \leq \left(\sup_{z\in\left\{\eta_0^{(n)},\ldots,\eta_n^{(n)}\right\}}\omega_{n+1}\left(\eta_j^{(n)}\right)\right)\left(\sum_{j=0}^{n}\left|L_j^{(n)}(z)\right|\right) \leq \left(\prod_{k=1}^{n}|z_{k+1}-z_k|\right)\left(\sum_{j=0}^{n}\left|L_j^{(n)}(z)\right|\right),$$

where $(z_i)_{i=0}^{\infty}$ is a sequence of Leja points. Therefore, from theorem 36 we follow that

$$\limsup_{n\to\infty}|\omega_{n+1}(z)|^{1/n} \leq c(K)e^{G(z)}. \tag{3}$$

Fourth, we consider

$$R_n(z) := \frac{\omega_{n+1}(z)}{L_0^{(n)}(z)}\frac{1}{\Delta_0^{(n)}e^{i\theta_n}},$$

for $n = 1, 2, \ldots,$ where

$$\theta_n = \arg\prod_{k=1}^{n}\left(\eta_0^{(n)} - \eta_k^{(n)}\right).$$

Clearly $R_n(z)$ as defined above is holomorphic in $\overline{\mathbb{C}}\backslash K$. In addition, since

$$R_n(z) = \prod_{k=1}^{n}\frac{z-z_k}{z-\eta_k^{(n)}},$$

we conclude that

$$\lim_{z\to\infty}\left(R_n(z)\right)^{1/n} = 1.$$

Since by equation 3, we have

$$|R_n(z)| \leq 1$$

and thus the maximum principle implies that

$$\lim_{n\to\infty}|R_n(z)|^{1/n} = 1.$$

Therefore,

$$\lim_{n\to\infty}|\omega_{n+1}(z)|^{1/n} = c(k)e^{G(z)}$$

and

$$U^{\tau_n} \to -G(z) + \gamma(K),$$

as desired.

□

Corollary 39. *A sequence of Leja points is maximally convergent.*

Proof. Follows from theorem 38 by employing theorem 30. □

Next let us extend the convergence results to functions of matrices. Note that we denote the spectrum of a Matrix A by $\sigma(A)$.

Theorem 40. *Suppose $K \subset\subset \mathbb{C}$ and A is a diagonalizable matrix such that $\sigma(A) \subset K$. In addition, suppose that $(z_i)_{i=0}^{\infty}$ is a sequence of Leja points. Then*

$$\|f(A)\boldsymbol{v} - p_n(A)\boldsymbol{v}\| \leq (\mathrm{cond}X)\,\|\boldsymbol{v}\|\|f - p_m\|_K,$$

where $A = XDX^{-1}$ and thus

$$\limsup_{n\to\infty} \|f(A)\boldsymbol{v} - p_n(A)\boldsymbol{v}\| \leq C\frac{1}{\lambda},$$

where C is some constant.

Proof. We have

$$\begin{aligned}\|f(A)\boldsymbol{v} - p_n(A)\boldsymbol{v}\| &= \|X\left(f(A) - p_n(A)\right)X^{-1}\boldsymbol{v}\| \\ &\quad \mathrm{cond}(X)\|f - p_m\|\|\boldsymbol{v}\|,\end{aligned}$$

which is the first result. The second result follows from theorem 32. □

We are now equipped with all the tools necessary to consider the computation of matrix functions by means of Leja interpolation. Thus, in the following sections we will introduce two types of exponential integrators that rely on the efficient implementation of matrix functions.

2.4 Magnus integrators

We will not repeat the motivation for Magnus integrators here (see e.g. [14, pp. 245-246]). However, let us consider the (discrete) Schrödinger equation[3]

$$\psi'(t) = -A(t)\psi(t), \qquad \psi(0) = \psi_0, \tag{4}$$

where $A(t)$ is at every point in time a skew-Hermitian matrix and, as usual in quantum mechanics, $\|\psi_0\|_2 = 1$. The following method is heavily employed in section 3.

Definition 41. (Exponential midpoint rule).

$$\psi_{n+1} = e^{hA(t_n+h/2)}\psi_n. \tag{5}$$

If A is time-independent, the exponential midpoint rule is exact. However, if this is not the case it is often advantageous to use higher order methods such that larger step sizes h can be employed. The following definition gives an example of a fourth-order method.

Definition 42. (Gaussian fourth-order Magnus integrator).

$$\psi_{n+1} = e^{B(t)}\psi_n, \tag{6}$$

where

$$B(t) := \frac{h}{2}\left(A_n(c_1h) + A_n(c_2h)\right) + \frac{\sqrt{3}}{12}h^2\left[A(t_n + c_2h), A(t_n + c_1h)\right]$$

and $c_{1,2} = \frac{1}{2} \mp \frac{\sqrt{3}}{6}$ are the two-point Gaussian quadrature nodes.

[3] In the physics literature usually the equation $i\psi'(t) = H(t)\psi(t)$ is considered and it is demanded that the Hamiltonian H is Hermitian.

It should be noted that usually we can precompute the matrix $B(t)$. Then, for numerical purposes, there is no difference in the implementation of the two matrix exponentials. Our goal is to approximate the matrix exponential given in equations 5 and 6 by a polynomial in Newton form. To that end we use the theory developed in sections 2.2 and 2.3.

It is beyond the scope of this master-thesis to give a complete treatment on Magnus integrators. To that end the interested reader is referred to [14, chap. 3] and the papers cited in this review article.

2.5 Exponential Runge-Kutta integrators

As a second class, we consider partial differential equations that, if discretized in space, can be written as

$$u'(t) = Au(t) + g(u(t)),$$

where $A \in \mathbb{R}^{n\times n}$ and $g\colon \mathbb{R}^n \to \mathbb{R}^n$ is a non-linear function. The most simple exponential integrator in this case is the so called exponential Euler method (see e.g [14, p. 223]).

Definition 43. (Exponential Euler method).

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

Thus, we need to compute two matrix functions, as well as a single evaluation of the function g. It should be noted that while the Magnus integrators introduced in section 2.4 are exact if A is time independent, the method introduced here is not. This is due to the fact that we evaluate g on a single point only. A more sophisticated method of order 2 is given in the following definition.

Definition 44. (Second order, two stage exponential Runge-Kutta method).

$$u_{n+1} = e^{-hA}u_n + h\left[(\varphi_1 - \varphi_2)(-hA)g(u_n) + \varphi_2(-hA)g(u_n + \varphi_1(-hA)g(u_n))\right].$$

In this case we need four matrix functions and two evaluations of g. Higher order methods have also been constructed (see e.g. [14, pp. 226-229]). Error bounds for some exponential integrators, under suitable conditions on g and A, are given in [14, chap. 2], for example.

3 Application

3.1 Quantum computation

Quantum computation is concerned with the exploitation of quantum mechanical phenomena to build a computer system that is, in the framework of complexity theory, more powerful than a classical computer. That this is possible is strongly suggested by the fact that, for example, Grover's algorithm can search an element in an unsorted list with time complexity $\mathcal{O}\left(\sqrt{N}\right)$. A good introduction to this approach to computation can be found in [20, chap. 1].

From this point of view, simulating a quantum computer is a futile attempt, at best. However, since only moderate size (on the order of 8 qubits) quantum computers have been constructed in the laboratory, the hope is that such simulations, even though significantly slower than real quantum computers, can give insight that ultimately will lead to the construction of quantum computers with a large number of qubits.

In this section we will only introduce the basic mathematical machinery that is necessary to understand the examples that follow. It is not our intention to give a complete treatment of quantum computation and physical realizations of such systems (for such a treatment we refer the interested reader to [20] or [21]).

A quantum mechanical system is described by the Schrödinger equation[4]

$$i\frac{\partial\psi}{\partial t} = H(t)\psi,$$

where $H(t)$ is the Hamiltonian and, depending on the problem, can be a (linear) operator or a matrix. The wave vector ψ either depends on continuous real variables (e.g. position) or on discrete variables (e.g. the spin of an elementary particle). A combination of a continuous system with a discrete system is also possible (this is the case, for example, when continuous degrees of freedom, such as position, are coupled to discrete degrees of freedom, such as spin). To combine different quantum systems, we have to employ the tensor product in the following manner $\psi = \psi^{(1)} \otimes \psi^{(2)}$. Note that in the discrete case the dimension of ψ is given by

$$\dim\psi = \left(\dim\psi^{(1)}\right)\left(\dim\psi^{(2)}\right).$$

Incidentally this simple formula leads to the difficulty of quantum mechanics, the many-body problem, which loosely states that the dimension of a system increases exponentially with the number of degrees of freedom in the system.

The physical interpretation of ψ is given by an Hermitian operator (or matrix) O that is called an observable. The eigenvalues of O are possible measurement outcomes (and are ensured to be real by the Hermiticity of the operator), whereas the probability that a system is found in the state corresponding to the eigenvalue λ is given by

$$P(\lambda, O) = \frac{\langle\psi, P_\lambda\psi\rangle_2}{\|\psi\|_2^2},$$

where P_λ is the projector onto the eigenspace of λ.

An important quantum mechanical system, especially for quantum computation, is that of a spin $\frac{1}{2}$ particle. In this case we measure the spin along the x,y or z-axis and get either 1 or -1 (called spin up and spin down, respectively). The observables for those measurements are given by the (scaled) Pauli matrices

$$S_x := \frac{1}{2}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, S_y := \frac{1}{2}\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, S_z = \frac{1}{2}\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

[4] We employ natural units, i.e. $\hbar = 1$ and $c = 1$.

Those matrices can easily be verified to have eigenvalues 1 and -1. As usual in quantum mechanics the eigenvectors correspond to the state where spin up or spin down is measured with probability 1 respectively.

3.1.1 Two spin system in a spin bath (First example)

Let us denote the spin observable along the α-axis acting on the i-th spin as

$$S_{\alpha,i} := \left(\bigotimes_{j=1}^{i-1} I\right) \otimes S_\alpha \otimes \left(\bigotimes_{j=i+1}^{L} I\right),$$

where L is the total number of spins in the system under consideration.

In this notation a system of two interacting spins (this is a special case of the Ising model) is described by the following Hamiltonian

$$H = J_0 \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{\alpha\in\{x,y,z\}} S_{\alpha,i}S_{\alpha,j}.$$

Including a spin bath (that is, spins that do not interact with each other but with our two spin system) we get

$$H = J_0 \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{\alpha\in\{x,y,z\}} S_{\alpha,i}S_{\alpha,j} + \sum_{i=3}^{L}\sum_{\alpha\in\{x,y,z\}} J_i S_{\alpha,i}\left(S_{\alpha,1} + S_{\alpha,2}\right).$$

The wave vector $\psi \in \mathbb{C}^{2^L}$ is initialized such that the first spin is up[5] (i.e. the eigenvector corresponding to the eigenvalue 1 of S_z), and the second spin is down. The remaining spins are normalized and initialized at random. Physically we investigate the time evolution of our two spin system if it is subjected to a number of random spins. This can, for example, be considered as a model for experimental error.

The parameters are chosen such that $J_0 = 8$ and J_i for $i \geq 3$ are the realizations of uncorrelated uniformly distributed random numbers in the interval $[0, 0.4]$.

Since H does not depend on time, in this case we can write the exact solution as

$$\psi(t) = e^{-itH}\psi_0.$$

However, since H is in general a large matrix, standard methods to compute the matrix exponential are not applicable. Therefore, we have to implement, for example, a method based on Leja interpolation[6]. Figure 2 gives the result of the expected value for the first spin subject to a bath of 13 spins (integrated up to time $t = 100$).

[5] If not further specified we measure spin along the z-axis.

[6] It should be noted that this is precisely the special case of the exponential mid-point rule introduced in section 2.4 if H is time-independent.

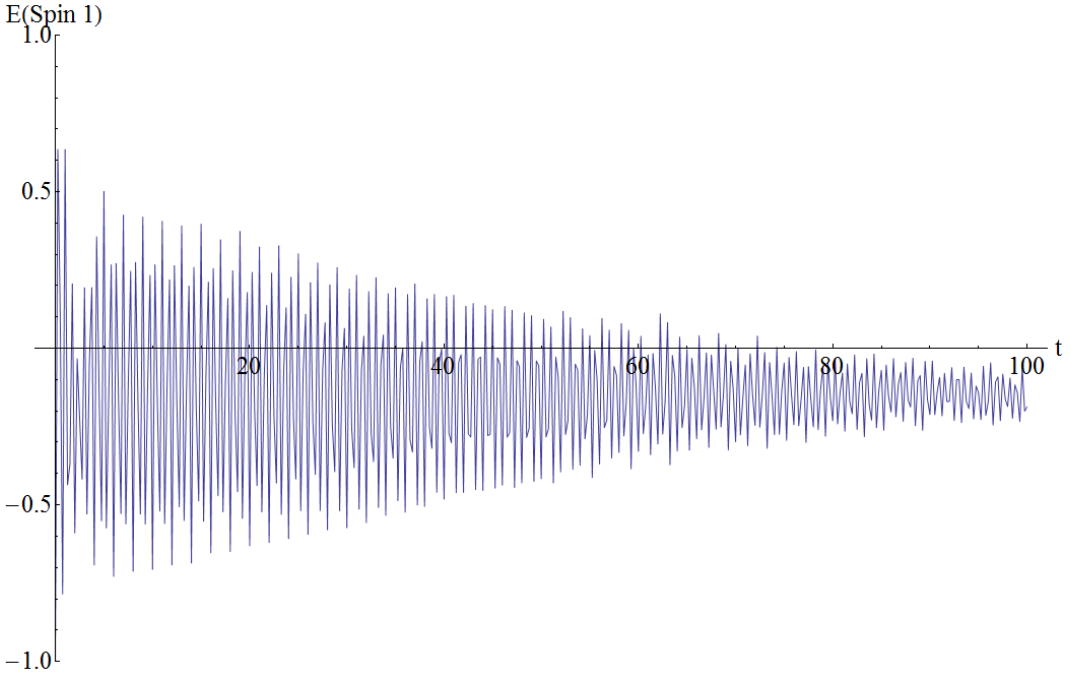

Figure 2: Two spin system in a spin bath ($L = 15$).

3.1.2 CNOT operation in a spin bath (Second example)

Although interesting on its own, for example in the study of decoherence, the example introduced in section 3.1.1 leaves much to be desired from a quantum computation standpoint. In this example we will therefore explain how to perform the "famous" controlled-not (CNOT) operation on two qubits in a spin bath.

First, we note that the CNOT operation is represented by the matrix

$$U_{\mathrm{CN}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

acting on a two spin wave vector (i.e. $\psi \in \mathbb{C}^4$). The matrix representation, however, is not sufficient to implement the gate on a quantum computer. To that end we must devise a physical Hamiltonian that implements the CNOT gate if we let it evolve for a specific amount of time. There are a number of physical realizations. However, in many cases, most notable the NMR quantum computer, a sinusoidal field is applied resulting in a Hamiltonian of a two spin system given by

$$\begin{aligned} H(t) = &-JS_{z,1}S_{z,2} - h_{z,1}S_{z,1} - h_{z,2}S_{z,2} \\ &- \sum_{i=1}^{2} k_{x,i}(t)\left(S_{x,i}\sin\omega t + S_{y,i}\cos\omega t\right) - \sum_{i=1}^{2} k_{y,i}(t)\left(S_{x,i}\cos\omega t - S_{y,i}\sin\omega t\right). \end{aligned} \tag{7}$$

Obviously, the Hamiltonian above does not include the spin bath. We will add this feature later on. For now, we are interested in implementing the CNOT gate by using the the frequency ω, the field strengths $k_{x,1}$, $k_{x,2}\,k_{y,1}$, $k_{y,2}$ as well as the time span t that frequency is applied as parameters.

To that end we decompose U_{CN} as follows

$$U_{\mathrm{CN}} = (I \otimes H)\, Z\, (I \otimes H)\,,$$

where

$$Z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

is called the controlled-Z gate and

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

is called the Hadamard gate. Thus, we need to perform two one-qubit operations[7] as well as a single two qubit operation (the controlled-Z gate). Our task is now to implement the Hadamard gate as well as the controlled-Z gate. To show that this is possible requires some careful analysis that is done in the following two sections.

Implementing the Hadamard gate The Hadamard gate can further be decomposed into single-spin rotations that are defined as

$$R_x(\varphi) := e^{-i\varphi S_x}$$
$$R_y(\varphi) := e^{-i\varphi S_y}.$$

A simple calculation then shows that[8]

$$H = -iR_x(\pi) R_y\left(\frac{\pi}{2}\right) = R_x(\pi) R_y\left(\frac{\pi}{2}\right).$$

Thus, it is sufficient to implement single-spin rotation to implement the Hadamard gate.

The Hamiltonian in equation 7 is not yet suited for analysis. Let us first neglect the spin-spin interaction (since the term is negligible if $t \ll \frac{1}{J}$)[9]. The Hamiltonian given in equation 7 has two terms corresponding to an electric field. From the following analysis it follows that we can either set $k_{x,1}, k_{x,2}$ or $k_{y,1}, k_{y,2}$ to zero (the difference being the axis of rotation). Therefore, let us, for the moment, drop the second term.

Now we change to a rotating frame, i.e. we introduce a new wave vector $\Phi(t)$ such that

$$\psi(t) = e^{i\omega t(S_{z,1}+S_{z,2})}\Phi(t).$$

This simplifies our analysis, since the Hamiltonian (of the time evolution of Φ) can be written as

$$H(t) = -\left[(h_{z,1}-\omega)S_{z,1} + (h_{z,2}-\omega)S_{z,2} + k_{x,1}S_{y,1} + k_{x,2}S_{y,2}\right]$$

and is therefore time-independent. To that end let us first choose the frequency such that

$$\omega = h_{z,1},$$

which in an NMR quantum computer can be interpreted as tuning ω to the resonance frequency of the first spin. Then the time evolution is given by

$$\psi(t) = e^{ith_{z,1}(S_{z,1}+S_{z,2})} e^{itk_{x,1}S_{y,1}} e^{it\boldsymbol{v}\cdot\boldsymbol{S}},$$

[7] A one-qubit operation is an operation on two (or more) qubits such that all but one qubit are left in their original states.

[8] In the following calculation, wave vectors different only by a global phase factor are identified. This follows from the axioms of quantum mechanics.

[9] At this point a mathematician might object that this has to be precisely justified. However, since our goal is to give a plausible argument that quantum computation can be done, rather than to base the discipline on a firm mathematical basis, the reader may permit us the use of the occasional hand-waving argument.

where

$$\boldsymbol{v} := \begin{bmatrix} 0 \\ k_{x,2} \\ h_{z,2} - h_{z,1} \end{bmatrix}$$

and

$$\boldsymbol{S} := \begin{bmatrix} S_{x,2} \\ S_{y,2} \\ S_{z,2} \end{bmatrix}.$$

It is clear that to achieve the desired operation, all but the second term must be equal to the identity. Furthermore, it must hold that

$$tk_{x,1} = -\varphi. \tag{8}$$

The first term can be computed easily and yields

$$e^{ith_{z,1}(S_{z,1}+S_{z,2})} = \begin{bmatrix} e^{ith_{z,1}} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{-ith_{z,1}} \end{bmatrix}.$$

Therefore, as a second condition we demand that

$$th_{z,1} = 2\pi k, \qquad k \in \mathbb{Z}. \tag{9}$$

To analyze the third term the following lemma proves useful (it is stated as an exercise in [20, p. 75]).

Lemma 45. ***(Exponential of the scaled Pauli matrices).*** *Suppose $\boldsymbol{v} \in \mathbb{C}^3$ with $|\boldsymbol{v}|_2 = 1$ and*

$$\boldsymbol{S} := \begin{bmatrix} S_x \\ S_y \\ S_z \end{bmatrix}.$$

Then for all $\theta \in \mathbb{R}$

$$e^{i\theta \boldsymbol{v}\cdot\boldsymbol{S}} = \cos\left(\frac{\theta}{2}\right) I + i \sin\left(\frac{\theta}{2}\right) \boldsymbol{v}\cdot\boldsymbol{S}.$$

Proof. We have

$$v_1 S_x + v_2 S_y + v_3 S_z = \frac{1}{2} \begin{bmatrix} v_3 & v_1 - iv_2 \\ v_1 + iv_2 & -v_3 \end{bmatrix}$$

and using that $\sqrt{v_1^2 + v_2^2 + v_3^2} = 1$ we get

$$e^{i\theta(v_1 S_x + v_2 S_y + v_3 S_z)} = \cos\left(\frac{\theta}{2}\right) I + i \sin\left(\frac{\theta}{2}\right) \boldsymbol{v}\cdot\boldsymbol{S}.$$

□

Using this lemma, we can write (ignoring spin 1 which is not affected by the calculation)

$$e^{it\boldsymbol{v}\cdot\boldsymbol{S}} = \cos\left(\frac{t|\boldsymbol{v}|}{2}\right) I + \frac{i}{|\boldsymbol{v}|} \sin\left(\frac{t|\boldsymbol{v}|}{2}\right) \boldsymbol{v}\cdot\boldsymbol{S},$$

and therefore to eliminate the third term we have to choose

$$t|\boldsymbol{v}| = t\sqrt{(h_{z,1} - h_{z,2})^2 + k_{z,2}^2} = 4\pi n, \qquad n \in \mathbb{Z}. \tag{10}$$

It is possible to give another set of equation that, under some assumptions, are equivalent to the three conditions derived above. This is the content of the next theorem.

Theorem 46. *The conditions 8, 9 and 10 are equivalent to the conditions*

$$h_{z,2} = \gamma h_{z,1}, \quad k_{z,2} = \gamma k_{z,1},$$

where $0 < \gamma < 1$ *together with the equation*

$$(1-\gamma)^2 k^2 + \frac{\gamma^2}{4}\left(\frac{\varphi}{2\pi}\right)^2 = n^2. \tag{11}$$

Proof. Starting with

$$t\sqrt{h_{z,1}^2(1-\gamma)^2 + \gamma^2\frac{\varphi^2}{t^2}} = 4\pi n$$

and by squaring both sides we obtain

$$t^2\left(h_{z,1}^2(1-\gamma)^2 + \gamma^2\frac{\varphi^2}{t^2}\right) = 16\pi^2 n^2.$$

By using that $h_{z,1} = \frac{2\pi k}{t}$ we have

$$(1-\gamma)^2 k^2 + \frac{\gamma^2}{4}\left(\frac{\varphi}{2\pi}\right)^2 = (2n)^2.$$

Since 8 is independent of k and 9 holds if k is an integer, the other direction is shown by solving equation 11 for n and substituting it into condition 10. □

Unfortunately, it is in general not possible to find a solution to equation 11 given the parameters above. However, assuming that $\gamma \in \mathbb{Q}$ we can get an approximation that proves sufficient for our application.

Theorem 47. *Suppose*

$$h_{z,2} = \gamma h_{z,1}, \quad k_{z,2} = \gamma k_{z,1},$$

such that

$$\gamma = \frac{N}{M},$$

where $N, M \in \mathbb{N}$. *Then for* $K \in \mathbb{N}$ *and* $k = KMN^2$ *there exists an* $n \in \mathbb{N}$ *such that*

$$t\sqrt{(h_{z,1}-h_{z,2})^2 + k_{z,2}^2} = 4\pi n + \mathcal{O}\left(\frac{1}{KM^2(M-N)}\right).$$

Proof. By the previous theorem we consider

$$\sqrt{(1-\gamma)^2 k^2 + \frac{\gamma^2}{4}\left(\frac{\varphi}{2\pi}\right)^2} = 4\pi n + \mathcal{O}\left(\frac{1}{KM^2(M-N)}\right).$$

Therefore, let us calculate

$$\begin{aligned}
\sqrt{(1-\gamma)^2 k^2 + \frac{\gamma^2}{4}\left(\frac{\varphi}{2\pi}\right)^2} &= \sqrt{\left(1-\frac{N}{M}\right)^2 K^2 M^2 N^4 + \frac{N^2}{4M^2}\left(\frac{\varphi}{2\pi}\right)} \\
&= \sqrt{K^2 N^4 (M-N)^2 + \frac{N^2\varphi}{4M^2 2\pi}} \\
&= KN^2(M-N) + \mathcal{O}\left(\frac{N^2}{M^2}\frac{1}{KN^2(M-N)}\right) \\
&= n + \mathcal{O}\left(\frac{1}{KM^2(M-N)}\right),
\end{aligned}$$

which gives us the desired result for $n = KN^2(M-N)$. □

In light of the previous theorem the relevant parameters t and $k_{x,1}$ are given by

$$th_{z,1} = -4\pi kMN^2, \qquad \frac{k_{x,1}}{h_{z,1}} = \frac{1}{2kMN^2}\frac{\varphi}{2\pi} = \frac{\varphi}{th_{z,1}}. \tag{12}$$

If the parameters t and $k_{x,1}$ as well as $k_{x,2} = \gamma k_{x,1}$ are determined as in equation 12, an (approximate) rotation $R_{y,1}(\varphi)$ is realized.

By reversing the role of spin 1 and spin 2 we get similar formulas

$$t_2 h_{z,1} = 4\pi kM^3, \qquad \frac{h_{x,2}}{h_{z,1}} = \frac{1}{2kM^3}\frac{\varphi_2}{2\pi}$$

to implement a rotation $R_{y,2}(\varphi)$.

The same formulas (with $h_{y,1}$ and $h_{y,2}$ instead of $h_{x,1}$ and $h_{y,2}$) implement the rotations $R_{x,1}(\varphi)$ and $R_{x,2}(\varphi)$. This is not shown, however, it is clear that the derivation is quite similar to the calculations considered above.

Implementing the Z gate In the previous section we showed that the Hadamard gate is decomposable into single-qubit rotations. That is not surprising since it can be shown that this is true of every single-qubit rotation (this result is not necessary for our purposes, the interested reader is refereed to [20, p. 175]). However, what is nothing short of remarkable is that we can implement the controlled-Z gate by doing nothing for a precisely specified time as well as performing single-qubit rotations only.

If no external RF field is applied, the Hamiltonian reads as

$$H = -\left[JS_{z,1}S_{z,2} + h_{z,1}S_{z,1} + h_{z,2}S_{z,2}\right].$$

Our argument rests on the assumption that if we consider the Hamiltonian

$$H_I := -J\left[S_{z,1}S_{z,2} - \frac{1}{2}S_{z,1} - \frac{1}{2}S_{z,2}\right]$$

and let it evolve for a time $t = \frac{\pi}{J}$. The result is given by (where we once again drop global phase factors)

$$e^{-i\frac{\pi}{J}H_I} = e^{-\frac{i\pi}{4}}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} = Z.$$

Now we can rewrite the Hamiltonian as

$$H = -\left[\left(h_{z,1} + \frac{J}{2}\right)S_{z,1} + \frac{\pi}{J}\left(h_{z,2} + \frac{J}{2}\right)S_{z,2} - H_I\right]$$

which in terms of the solution gives us

$$e^{-i\frac{\pi}{J}H} = e^{i\frac{\pi}{J}\left(h_{z,1}+\frac{J}{2}\right)S_{z,1}}e^{i\frac{\pi}{J}\left(h_{z,2}+\frac{J}{2}\right)S_{z,2}}e^{-i\frac{\pi}{J}H_I},$$

or by rearranging

$$Z = e^{-i\frac{\pi}{J}\left(h_{z,1}+\frac{J}{2}\right)S_{z,1}}e^{-i\frac{\pi}{J}\left(h_{z,2}+\frac{J}{2}\right)S_{z,2}}e^{-i\frac{\pi}{J}H}.$$

The rotation around the z-axis can be decomposed into rotations around the x- and y-axis as follows[10]

$$e^{-i\frac{1}{J}\left(h_z+\frac{J}{2}\right)S_z} = R_y\left(\frac{\pi}{2}\right)^* R_x\left(\frac{\pi}{J}\left(h_z + \frac{J}{2}\right)\right)R_y\left(\frac{\pi}{2}\right).$$

[10] Note, that we drop the spin index here, since this is a single spin operation.

Implementing the CNOT gate Using the results from the previous sections we can finally decompose the CNOT gate into a time evolution of the free Hamiltonian as well as rotations around the x- and y-axis

$$\begin{aligned} U_{CN} &= \left[R_{x,2}\left(\pi\right) R_{y,2}\left(\frac{\pi}{2}\right)\right]\left[R_{y,2}\left(\frac{\pi}{2}\right)^{*} R_{x,2}\left(\frac{\pi}{J}\left(h_{z,2}+\frac{J}{2}\right)\right) R_{y,2}\left(\frac{\pi}{2}\right)\right] \\ &\quad \left[R_{y,1}\left(\frac{\pi}{2}\right)^{*} R_{x,1}\left(\frac{\pi}{J}\left(h_{z,1}+\frac{J}{2}\right)\right) R_{y,1}\left(\frac{\pi}{2}\right)\right]\left[e^{-i\frac{\pi}{J}H}\right]\left[R_{x,2}\left(\pi\right) R_{y,2}\left(\frac{\pi}{2}\right)\right] \\ &= R_{x,2}(\pi) R_{x,2}\left(\frac{\pi}{J}h_{z,2}+\frac{3}{2}\pi\right) R_{y,2}\left(\frac{\pi}{2}\right) R_{y,1}\left(\frac{3}{2}\pi\right) \\ &\quad R_{x,1}\left(\frac{\pi}{J}\left(h_{z,1}+\frac{J}{2}\right)\right) R_{y,1}\left(\frac{\pi}{2}\right) e^{-i\frac{\pi}{J}H} R_{x,2}\left(\pi\right) R_{y,2}\left(\frac{\pi}{2}\right). \end{aligned}$$

In total, we therefore have 8 single qubit-rotations and an evolution of the free Hamiltonian for a time $\frac{\pi}{J} \approx 7.30603 \cdot 10^{6}$ (for $J = -0.431 \cdot 10^{-6}$). It should be noted, that this is a quite large time-frame compared to the single qubit rotation, for example. In Table 1 the parameter values for all necessary operations are listed.

Operation	t	ω	$k_{x,1}$	$k_{x,2}$	$k_{y,1}$	$k_{y,2}$	φ
$R_{y,2}\left(\frac{\pi}{2}\right)$	$256\pi k$	0.25	$4 \cdot k_{x,2}$	$-0.001953125 \cdot \frac{1}{k}$	0	0	$\frac{\pi}{2}$
$R_{x,2}\left(\pi\right)$	$256\pi k$	0.25	0	0	$4 \cdot k_{y,2}$	$-0.00390625 \cdot \frac{1}{k}$	π
$e^{-i\frac{\pi}{J}H}$	$-\frac{\pi}{J}$	-	0	0	0	0	-
$R_{y,1}\left(\frac{\pi}{2}\right)$	$16\pi k$	1	$-0.03125 \cdot \frac{1}{k}$	$\frac{1}{4} \cdot k_{x,1}$	0	0	$\frac{\pi}{2}$
$R_{x,1}\left(\frac{\pi}{J}\left(1+\frac{J}{2}\right)\right)$	$16\pi k$	1	0	0	$-0.00654069 \cdot \frac{1}{k}$	$\frac{1}{4} \cdot k_{y,1}$	0.328771
$R_{y,1}\left(\frac{3}{2}\pi\right)$	$16\pi k$	1	$-0.0117188 \cdot \frac{1}{k}$	$\frac{1}{4} \cdot k_{x,1}$	0	0	$\frac{3}{2}\pi$
$R_{y,2}\left(\frac{\pi}{2}\right)$	$256\pi k$	0.25	$4 \cdot k_{x,2}$	$-0.001953125 \cdot \frac{1}{k}$	0	0	$\frac{\pi}{2}$
$R_{x,2}\left(\frac{\pi}{J}\frac{1}{4}+\frac{3}{2}\pi\right)$	$256\pi k$	0.25	0	0	$4 \cdot k_{y,2}$	$-0.00059048 \cdot \frac{1}{k}$	0.474892
$R_{x,2}\left(\pi\right)$	$256\pi k$	0.25	0	0	$4 \cdot k_{y,2}$	$-0.00390625 \cdot \frac{1}{k}$	π

Table 1: Values for the parameters t, ω, $k_{x,1}$, $k_{x,2}$, $k_{y,1}$, $k_{y,2}$ needed to implement U_{CN} for $\gamma = \frac{1}{4}$ and $h_{z,1} = 1$ as well as $J = -0.431 \cdot 10^{-6}$.

3.2 PDEs with the Laplacian as linear part (Third example)

In this section we consider initial value problems of the form

$$\begin{cases} u_t = \Delta u - g(u) \\ u|_{\{t=0\}} = u_0 \\ u|_{\partial\Omega} = 0, \end{cases}$$

on the domain $\Omega = [0,1]^3$. Burger's equation with homogeneous Dirichlet boundary conditions, for example, fits into this framework. In this case we have

$$g(u) = \frac{1}{\mathrm{Re}} (u \cdot \nabla) u,$$

where Re is the (dimensionless) Reynolds number.

Therefore, in light of the integrators introduced in section 2.5, we have to discretize the Laplacian in three space dimensions.

3.2.1 Discretization of the Laplacian

Our goal is to derive the matrix A that represents a discretization of the Laplacian Δ in three space dimensions. Let us consider the domain $\Omega = [0,1]^3$, where N is the number of points per space dimension.

The position in Ω is given by three indices $i_x, i_y, i_z \in \{0, \ldots, N-1)$ in the following manner

$$(x, y, z) = \left(\frac{i_x + 1}{N+1}, \frac{i_y + 1}{N+1}, \frac{i_z + 1}{N+1} \right)$$

We flatten this three dimensional vector by using the single index

$$i := i_x + i_y N + i_z N^2.$$

By inverting the above operation we get

$$i_x = i \bmod N, \tag{13}$$

$$i_y = \lfloor \frac{i}{N} \rfloor \bmod N, \tag{14}$$

$$i_z = \lfloor \frac{i}{N^2} \rfloor \bmod N. \tag{15}$$

We use the standard approach to discretize the Laplacian (with homogeneous boundary conditions) in three space dimensions using the formula for a three point stencil in each dimension, i.e.

$$\begin{aligned} (Au)(i_x, i_y, i_z) = & -6u(i_x, i_y, i_z) + \\ & + u(i_x + 1, i_y, i_z) + u(i_x - 1, i_y, i_z) \\ & + u(i_x, i_y + 1, i_z) + u(i_x, i_y - 1, i_z) \\ & + u(i_x, i_y, i_z + 1) + u(i_x, i_y, i_z - 1), \end{aligned}$$

where it is understood that $u(i_x, i_y, i_z) = 0$, if any index does not lie in the range of 0 and $N-1$.

Using equations 13, 14 and 15 we can give an explicit formula for the entries of matrix A

$$A_{ij} = (N+1)^2 \left[-6\delta_{ji} + \delta_{j,i-N^2} + \delta_{j,i+N^2} + \delta_{j,i-N}\delta_{i_y 0} + \delta_{j,i+N}\delta_{i_y,N-1} + \delta_{j,i-1}\delta_{i_x 0} + \delta_{j,i+1}\delta_{i_x,N-1} \right].$$

This formula is helpful for implementing the CSR representation of A as well as to implement a GPU kernel for matrix-vector multiplication. The implementation for the GPU kernel is given in algorithm 11 on page 45.

4 Implementation

4.1 The CUDA Programming model

4.1.1 Grid, Blocks, Warps, and Threads

The CUDA programming model is an abstraction of a massively parallel architecture (found in NVIDIA GPUs) on top of the C/C++ programming language[11]. To query hardware information, we use the program deviceQuery which is part of the CUDA SDK (a sample output for a Tesla C1060 is given in Table 2 for the CUDA SDK 4.0 RC2).

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 the other cores, which reside on the same streaming multiprocessor. Therefore, the hardware, in case of the C1060, consists in total of $30 \cdot 8 = 240$ cores that are distributed across 30 multiprocessors of 8 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). This implies that 4 clock-cycles are needed to process one instruction[12]. This situation is quite different from compute capability 2.0 devices (e.g. the C2050) where each SM consists of 32 cores (matching each thread in a warp to a single core). It is also evident from this discussion that even if the threads in the same warp take exactly the same execution path, they 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 notably 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. An example of the latter usage is given, for example, in algorithm 9 on page 41.

Global memory is a RAM (random access memory) that is shared by all SM on the entire GPU. The size of the global memory can be found from Table 2 to be 4 GB, whereas the shared memory is a mere 16 Kb (but obviously this is 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 every block is further subdivided into threads. Much to the confusion of the novice CUDA programmer either the number of blocks in the single grid is called the grid size, whereas the number of threads in a block is called the block size, or the number of blocks in the single grid is called the block dimension, whereas the number of threads in a block is called the threads per block. No matter which naming convention is employed, it should be noted that there is always a single grid, i.e. only block size and thread size (per block) need to be specified. This design is illustrated in Figure 3.

It is guaranteed that all threads in a block are executed on the same SM. Thus, within a block we can use shared resources (like shared memory). It is however not necessary that all threads within a block are executed simultaneously or that all blocks in a grid are executed simultaneously (usually the block size will be much larger than the number of SM on the GPU available). Thus, it is necessary for the programmer to write code that remains correct if executed serially as well as in parallel.

Furthermore, the number of threads per block is limited to 512 (as can be seen from Table 2). The number of threads per block can be given as an integer, as a variable of type dim2, or as a variable of dim3[13]. However, the maximum number of threads is not allowed to exceed 512, i.e.

$$\text{dim3}(32, 16, 8)$$

[11] It should be noted, this is the most common but not the only option. CUDA Fortran as well as third party bindings to many programming languages (including python and matlab) exist.

[12] Obviously it is assumed that the instruction under consideration takes exactly one clock-cycle.

[13] The data types dim2 and dim3 are two and three dimensional arrays of integers, respectively.

```
Device 0: "Tesla C1060"
  CUDA Driver Version / Runtime Version          4.0 / 4.0
  CUDA Capability Major/Minor version number:    1.3
  Total amount of global memory:                 4096 MBytes (4294770688 bytes)
  (30) Multiprocessors x ( 8) CUDA Cores/MP:     240 CUDA Cores   GPU Clock Speed: 1.30 GHz
  Memory Clock rate:                             800.00 Mhz
  Memory Bus Width:                              512-bit
  Max Texture Dimension Size (x,y,z)             1D=(8192), 2D=(65536,32768), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(8192) x 512, 2D=(8192,8192) x 512
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             256 bytes
  Concurrent copy and execution:                 Yes with 1 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Concurrent kernel execution:                   No
  Alignment requirement for Surfaces:            Yes
  Device has ECC support enabled:                No
  Device is using TCC driver mode:               No
  Device supports Unified Addressing (UVA):      No
  Device PCI Bus ID / PCI location ID:           2 / 0
  Compute Mode:      < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

Table 2: Device query for a single Tesla C1060.

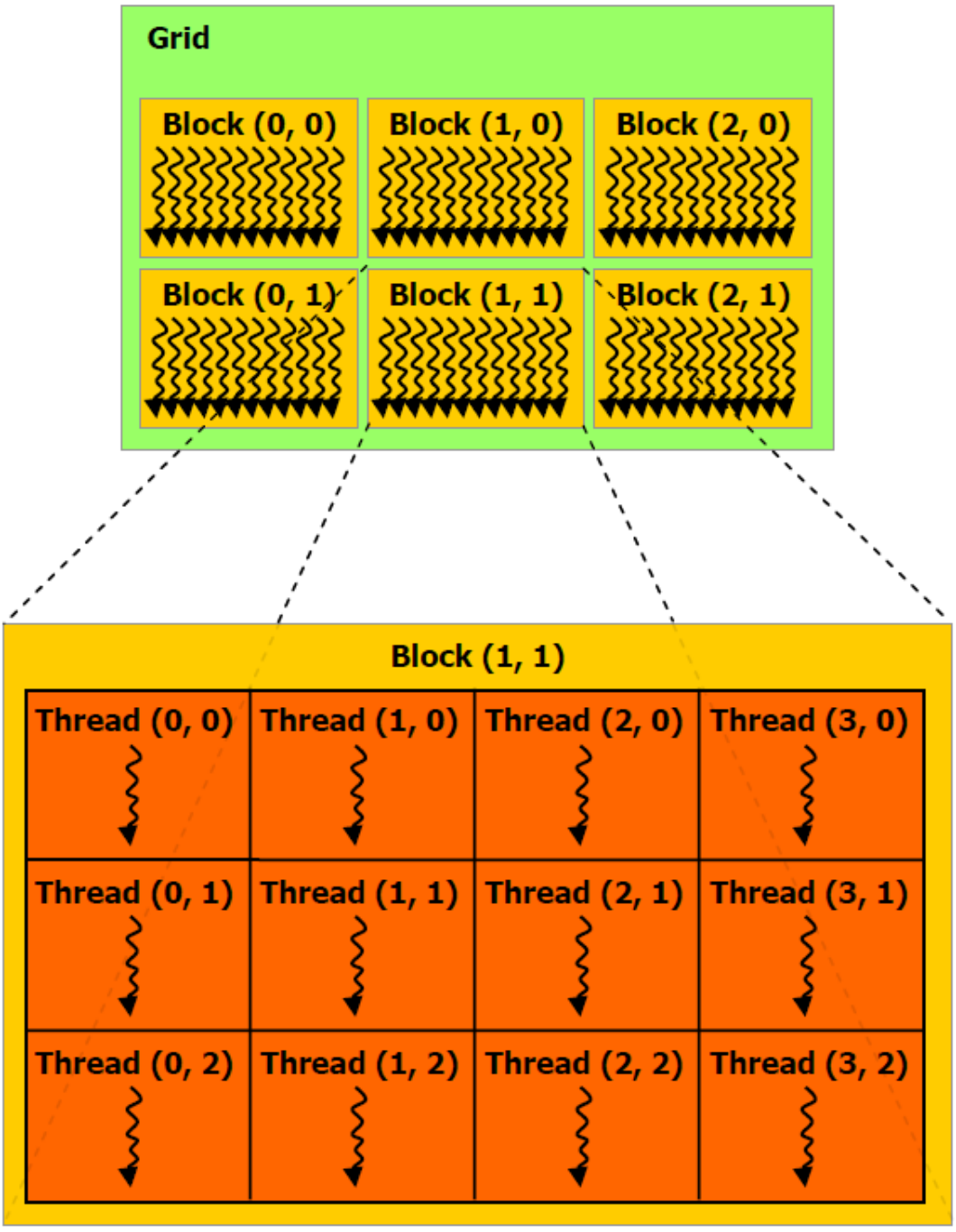

Figure 3: CUDA programming model (illustration taken from [3, p. 21]).

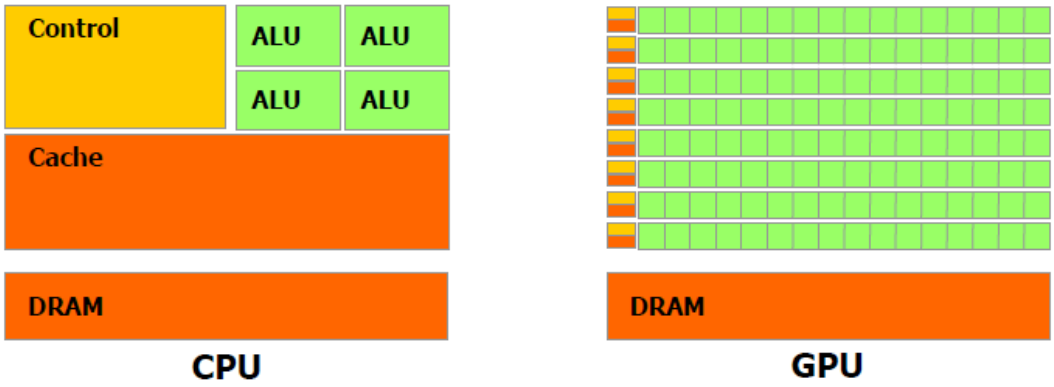

Figure 4: GPU vs. CPU architecture (illustration taken from [3, p. 15]).

is not a valid configuration, while

$$\text{dim3}(16, 8, 4)$$

is. The dimensionality of those variables is merely syntactic sugar, as on the GPU no such two or three dimensional arrays of threads exist. However, it is often convenient to use two or three dimensional variables to map certain problems (i.e. matrix-matrix multiplication) more directly to the underlying mathematics.

The number of blocks per grid is not limited in such a way. In this case it is only required that the z-component is 1 whereas the x and y-components can range from 0 to 65535 (this is once again listed in Table 2). Thus, we can have a total of $4294836225 \approx 4.3 \cdot 10^9$ blocks in a grid. This should prove more than sufficient for any application.

We will, in what follows, use the naming convention of CUDA (see e.g. [3]) that gives the number of threads per block by

$$\text{dim3}(blockDim.x, blockDim.y, blockDim.z)$$

and the number of blocks per grid by

$$\text{dim3}(gridDim.x, gridDim.y, gridDim.z).$$

It should be noted that, as a rule of thumb, performance is increased if the memory access is sequential (often called coalesced in this context). This paradigm is also stated in [2]. We will give some examples as well as investigate this behavior more closely in section 4.3. There we will also find that such a code gives, in most cases, a performance advantage compared to a naive implementation (that does not take sequential memory access into account).

Let us last, but not least, discuss the differences in architecture between a typical GPU and a typical CPU. As illustrated in Figure 4 the highly parallel nature of GPUs is achieved mainly by focusing on floating point performance and by sacrificing the quite complicated hierarchy of caches found on most traditional CPUs. Thus, more area on the die is available for raw data processing.

4.1.2 Kernels

A kernel is a function (in the C/C++ sense) that is launched on the GPU. Such a kernel is executed a total of

$$gridDim.x \cdot gridDim.y \cdot gridDim.z \cdot blockDim.x \cdot blockDim.y \cdot blockDim.z$$

times. That is, the same function is executed a larger number of times (potentially in parallel, but this is by no means guaranteed as discussed in section 4.1.1). However, the programmer can use the index (of a thread within a block) as well as the index of a block (in a grid) to control the behavior of the program

Algorithm 1 Simple kernel launch in CUDA.

```
1  __global__ void k_do(int N, double* x)
2  {
3          /* every thread has a unique id */
4          int id = blockDim.x*blockIdx.x + threadIdx.x;
5
6          /* do stuff depending on id */
7  }
8
9  int main()
10 {
11         int num_blocks;
12         int num_threads;
13
14         /* setup
15         */
16
17         k_do<<<num_blocks, num_threads>>>(N, x);
18
19         /* cleanup
20         */
21
22         return 0;
23 }
```

depending on the specific thread instance. A kernel would use such indices, for example, as an index into an array of data to process. How to declare a kernel and configure the kernel call[14] is illustrated in algorithm 1.

The syntax on line 17 of algorithm 1 is an addition to the C/C++ programming language that is provided by the CUDA compiler (nvcc). It is exclusively used to configure (i.e. to specify the grid and block size) kernel launches on the GPU. As illustrated in the code, we can use the variables blockDim, blockIdx, and threadIdx to assign a unique id to every thread. This id can then be used, e.g., to process a single row of a matrix.

It should be noted if call by value is used (e.g. the first parameter in k_do) the value is automatically copied from host memory to device memory[15]. However, if we pass a pointer (e.g. the second parameter in k_do) we have to manually allocate memory on the device and copy the necessary data to the device. This is important since host memory and device memory are two distinct random access memories that communicate over the PCI-E bus only. We will see in sections 5.2.1 and 4.4.3 that this can be a bottleneck, especially if large amounts of data have to be transferred between different GPUs.

For the purpose of memory allocation and memory transfer, the CUDA environment provides the functions cudaMalloc, cudaMemcpy, and cudaFree (among others). The use of this functions is illustrated in algorithm 2, where we have modified the program in such a way that every thread squares a single element of a one-dimensional array that is copied from the host to a device. The result of this process is then copied back to the host for further processing (note the use of cudaMemcpyHostToDevice as well as cudaMemcpyDeviceToHost to indicate the direction of data transfer).

In algorithm 2 our first, more or less, complete CUDA example is given. We have omitted the logic of reading the data into host memory as well as doing something useful once the modified data is copied back to the host. Nevertheless, all essential steps to CUDA programming, i.e.

[14] To configure the kernel is a synonym for launching the kernel with a specified number of blocks (per grid) and threads (per block).

[15] In CUDA we refer to the CPU as the host, whereas the GPU is referred to as the device.

Algorithm 2 Memory allocation/copy on/to the device in CUDA.

```
__global__ void k_do(int N, double* x)
{
        /* every thread has a unique id */
        int id = blockDim.x*blockIdx.x + threadIdx.x;

        x[id] = x[id]*x[id];
}

#define N 256*256

int main()
{
        double x[N];

        /* populate x
        */

        float *d_x;
        /* allocate memory on the device */
        cudaMalloc(&d_x, sizeof(double)*N);
        /* copy memory to the device */
        cudaMemcpy(d_x, x, sizeof(double)*N, cudaMemcpyHostToDevice);

        k_do<<<N/256, 256>>>(N, x);

        /* copy memory back to the host */
        cudaMemcpy(x, d_x, sizeof(double)*N, cudaMemcpyDeviceToHost);
        /* free memory on the device */
        cudaFree(d_x);

        /* process x
        */

        return 0;
}
```

1. Allocate memory on the device.
2. Copy data to the device.
3. Launch one (or multiple) kernels to process the data.
4. Copy the data back to the host.

have been implemented in this simple example.

On several occasions it is necessary to copy data on the device to a different memory location. In this case, cudaMemcpy with cudaMemcpyDeviceToDevice is employed (in this case we obviously don't have to use the PCI-E bus; i.e. such copy operations are quite fast). It should, however, be noted that to copy data between two different GPUs it is necessary to first copy it back from the source GPU to the CPU, before copying it to the destination GPU. Thus, we have to transfer twice the memory over the (slow) PCI-E bus.

Since this is potentially a performance bottleneck, for devices of compute capability 2.x and CUDA 4.0, a more direct method is introduced via a technology called unified addressing (UVA). It can also be

Algorithm 3 The fp and fp_real data types.

```
#if !COMPLEX_ARITH
  #if USE_DOUBLE
    #define fp           double
    #define fp_real double
  #else
   #define fp            float
        #define fp_real float
  #endif
#else
  #if USE_DOUBLE
        #define fp                doublecomplex
        #define fp_real double
        #define Complex(a,b) \
        doublecomplex::make_cudacomplex(tofloat(a),tofloat(b))
  #else
        #define fp                singlecomplex
    #define fp_real      float
        #define Complex(a,b) \
        singlecomplex::make_cudacomplex(tofloat(a),tofloat(b))
  #endif
#endif
```

determined from the output given in Table 2, whether this mode is available. In this case, we can copy data directly between different GPUs (in the same host system). Since UVA is not available for devices of compute capability 1.3 (such as the C1060) we will not elaborate on this feature in what follows. It should however be duly noted, that the extension of the library described in section 4.2 to this model is straight forward (since only the memory copy operations need to be modified).

4.2 The cexp library

4.2.1 Introduction

The purpose of the cexp library is to provide a library that implements exponential integrators based on interpolation at Leja points. Specifically, the goal is to provide a fast method for computing matrix functions (with the Real Leja point method) by utilizing the CUDA framework. To that end we will mainly consider the hardware listed in section 5.1.

The library is designed to be modular (this is achieved by inclusion of some object oriented paradigms) and facilitates that different implementations of matrix-vector multiplication (be it on the CPU or the GPU) can be used and compared easily. The library uses certain features of the C++ programming language (e.g. inheritance) only if it is clear that performance is not negatively affected by doing so.

The following sections describes the main classes that are employed in an exponential integrator using this library. We will employ a bottom-up approach, i.e. the more basic classes are described first. For example, we first discuss the class responsible for matrix-vector multiplication (MatrixMultiplicator). This treatment is then followed by the class that implements matrix functions (MatrixFunction).

In our development we use the define flags given in algorithm 3 to map fp and fp_real to the corresponding data types (fp is either complex or real, while fp_real is always real).

The types doublecomplex and singlecomplex are classes that have been written by Christian Buchner and are available at `http://forums.nvidia.com/index.php?showtopic=73978`. They can be used on the CPU as well as on the GPU and provide many operators that have been overloaded to conform to the expected behavior of complex numbers.

Algorithm 4 CSRMatrix and Vector class.

```
struct CSRMatrix {
        int n;  // dimension
        int nnz;
        int on_gpu;
        fp* val;
        int* col_ind;
        int* row_ptr;

        void load(int n, int nnz, char* file_prefix);
        CSRMatrix copy_to_gpu();
        void gershgorin(fp_real* a, fp_real* b);
        void destroy();
};

struct Vector {
        int n;
        fp *val;
        int on_gpu;

        Vector();
        Vector(int n, fp* val, int on_gpu);
        void load(int n, char* file);
        Vector copy_to_gpu();
        void destroy();
};
```

4.2.2 Class: CSRMatrix and Vector

These two classes define a square sparse matrix (in CSR format) and a dense vector respectively. They provide methods to load matrices/vectors from disk as well as include member variables that give the dimension (n), number of nonzero elements (nnz), as well as a flag, specifying if the data pointers reside on the host or the device. The CSRMatrix class also provides a gershgorin method that computes an upper and lower bound for the absolute values of the eigenvalues of a matrix. The copy_to_gpu methods leave the calling matrix/vector unchanged and return an identical matrix/vector that resides on the device (i.e. memory is allocated on the device and the data from the existing matrix in host memory is transferred to the allocated device memory). The definitions are given in algorithm 4.

4.2.3 Class: MatrixMultiplicator

The MatrixMultiplicator is a virtual class which is an interface for an implementation of a sparse matrix-vector multiplication, where the matrix is given in the CSR format[16]. The definition is given in algorithm 5.

The init method takes a CSRMatrix that resides on the host and is responsible for (if applicable) copying that matrix to a device (or multiple devices). The compute function implements the matrix-vector multiplication in the m-th step of the Leja interpolation of

$$e^{\alpha A+\beta I}y,$$

where the vector vec has to be updated accordingly. The return code 0 of the compute method continues the computation, while 1 is returned if the desired tolerance condition has been met. Thus, an imple-

[16] It should be noted that, beside matrix-vector multiplication, this class is responsible for some other tasks in the Leja algorithm (i.e. copying from host to device memory).

Algorithm 5 MatrixMultiplicator virtual class definition.

```
class MatrixMultiplicator
{
public:
        virtual void init(CSRMatrix mat) = 0;
        virtual int compute(int m, Vector vec, fp_real alpha,
                            fp_real beta, fp divdiff, fp_real tol) = 0;
        virtual void destroy() = 0;
};
```

Class name	Architecture	Implementation method
MMCPU	CPU	Parallelized to multiple CPUs (OpenMP)
MMSingleGPU	GPU	Parallelized to a single GPU (Cusparse)
MMMultipleGPU	Multiple GPU	Parallelized to multiple GPUs (Cusparse)
MMLaplace3GPU	GPU	Parallelized to a single GPU (fast method of section 4.4.4)
MMSingleGPUHandwritten	GPU	Parallelized to a single GPU (method given in section 4.3)

Table 3: Implementations of the MatrixMultiplicator virtual class provided with cexp.

mentation of the MatrixMultiplicator class is also responsible for deciding if this tolerance condition has been achieved.

The current state of the library implements a number of matrix multiplicators that are given in Table 3. If starting a new implementation, it is most instructive to study the implementations already provided (especially the CPU as well as the single GPU implementation).

It should also be noted that for the "fast" method the matrix passed as a parameter to init is only used to compute an estimate of the eigenvalues (via the Gershgorin disks). In the actual computation no matrix has to be stored and therefore no CSR representation in (any) memory is required.

The multiple GPU implementation includes two additional methods (list_gpus, set_gpus) that must be called before calling the init method. This methods are used to specify the number of GPUs to parallelize to, as well as which devices are used.

4.2.4 Class: MatrixFunction

The class MatrixFunction is a rather simple class that uses a MatrixMultiplicator to compute a matrix function. A partial definition is given in algorithm 6.

The init method expects a function pointer to the matrix function that is to be computed (the divided

Algorithm 6 The MatrixFunction class.

```
struct MatrixFunction
{
public:
        void init(fp_real h, fp_real tol, CSRMatrix mat,
                  int on_gpu, fp (*fun)(fp_real, fp_real, fp_real, fp_real),
                  MatrixMultiplicator* mat_mul);
        void compute(Vector vec);
        void destroy();
}
```

Algorithm 7 Implementation of two spins in a spin bath (2 GPUs, $L = 18$, $h = 1$, and double precision).

```
#define COMPLEX_ARITH 1
#define USE_DOUBLE    1
#define MAX_NOF_LEJA_POINTS 500

#include "matrixfunc/matrixfunc.cu"
#include "matrixmultiplicators/mmmultiplegpu.cu"

int main()
{

        CSRMatrix mat;
        mat.load(262144, 8912896, "data");

        Vector vec;
        vec.load(262144, "data_initial.bin");

        MMMultipleGPU mat_mul_mgpu;
        mat_mul_mgpu.list_gpus();
        int gpu_l[2] = {0, 2};
        mat_mul_mgpu.set_gpus(2,gpu_l);

        MatrixFunction mat_exp;
        mat_exp.init(0.1, 1e-5, mat, 1, exp_fun, &mat_mul_gpu);

        tic();
        mat_exp.compute(vec);
        toc("Runtime");

        mat_exp.destroy();
        mat.destroy();
        vec.destroy();

        return 0;
}
```

differences are then computed automatically), as well as an instance of a MatrixMultiplicator that is **not** initialized (i.e. the init function has not yet been called).

The class automatically initializes the MatrixMultiplicator given by mat_mul and computes the necessary spectral data for appropriate scaling of the matrix. Since this may take some time it is good practice to call this function only once, even if a number of calls is made to the compute function.

4.2.5 Example

Let us, as an example, give the code used to implement the problem given in 3.1.1 using the multiple GPU implementation on two devices (employing devices number 0 and 2). The code is rather self-explanatory and is given in algorithm 7. It is assumed that the binary files data_row_ptr.bin, data_col_ind.bin, data_val.bin, and data_initial.bin are constructed by a Mathematica program and are located in the path of the program.

The tic and toc functions are also provided by the cexp library and are somewhat inspired by the equivalently named MATLAB functions that are used to measure runtime (they employ the high performance counters clock_gettime and QueryPerformanceCounter on Linux and Windows, respectively).

Algorithm 8 Naive k_csrmv.

```
void __global__ k_csrmv(int N, fp_real alpha, fp* val, int* row_ptr,
                        int* col_ind, fp* d_x, fp_real beta, fp* d_y)
{
        int istart       = blockIdx.x*blockDim.x + threadIdx.x;
        int stride       = blockDim.x * gridDim.x;
        int idx, idx_end;
        fp y;

        for(int i=istart;i<N;i+=stride)
        {
                idx      = row_ptr[i];
                idx_end  = row_ptr[i+1];
                y        = beta*d_y[i];
                while(idx < idx_end)
                {
                        y += alpha*val[idx]*d_x[col_ind[idx]];
                        idx++;
                }
                d_y[i] = y;
        }
}
```

4.3 Writing a fast matrix-vector multiplication

The purpose of this section is to write a GPU kernel that implements a matrix-vector multiplication of (possibly non-rectangular) sparse matrices in the CSR format. One might argue that this is unnecessary, since the Cusparse library already implements such an operation (the use of the Cusparse library is further explained in section 4.4.3).

However, to write such a program is instructive for two reasons. First, it gives an example of a non-trivial (at least if properly optimized) kernel and second it shows, quite surprisingly, that it is indeed possible to beat the implementation of the Cusparse library (at least for the two examples considered here). However, since the Cusparse library is not open source, we can not directly compare the method used in its implementation and it is therefore difficult to determine why exactly such a performance gain is accomplished.

4.3.1 The naive approach

The first implementation we introduce here, is a straightforward parallelization of the matrix-vector multiplication. In this case, we assign (a maximum of) one thread to every row of the matrix. The implementation of such a kernel is given in algorithm 8.

This implementation is not as good as the Cusparse library as can bee seen from the performance comparison conducted in Table 4.

From a theoretical point of view we can analyze the memory access pattern of this implementation. Let us use, for example, the matrix

$$A = \begin{bmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{bmatrix}, \tag{16}$$

that is represented in the CSR format as follows (if unfamiliar with this format, a simple example is given in [1, pp. 9-11])

$$\begin{aligned} \text{val} &= \begin{bmatrix} -2 & 1 & 1 & -2 & 1 & 1 & -2 & 1 & 1 & -2 & 1 & 1 & -2 \end{bmatrix}, \\ \text{row\_ptr} &= \begin{bmatrix} 0 & 2 & 5 & 8 & 11 & 13 \end{bmatrix}, \\ \text{col\_ind} &= \begin{bmatrix} 0 & 1 & 0 & 1 & 2 & 1 & 2 & 3 & 2 & 3 & 4 & 3 & 4 \end{bmatrix}. \end{aligned}$$

Then (suppose that stride = 1, i.e. we have exactly one thread per row) we have the following memory access pattern (we list all memory access into the val array for a given iteration of the while loop, numbered by thread id)

$$\begin{aligned} \text{Iteration: } 0 &\begin{bmatrix} 0 & & 1 & & 2 & & 3 & & 4 & \end{bmatrix}, \\ \text{Iteration: } 1 &\begin{bmatrix} & 0 & & 1 & & 2 & & 3 & & 4 \end{bmatrix}, \\ \text{Iteration: } 2 &\begin{bmatrix} & & & & 1 & & 2 & & 3 & \end{bmatrix}. \end{aligned}$$

As mentioned in section 4.1.1 this is not optimal, since for best performance, threads should access memory sequentially. Thus, let us investigate if it is possible to write a code that enables a more sequential memory access pattern.

4.3.2 Improving upon the performance of Cusparse

To improve the memory access pattern of the code discussed in the previous section, we use the idea given in [8]. In this implementation it is suggested to use a full warp for every row of our matrix. The code is reproduced in algorithm 9.

Let us once again consider the matrix given in equation 16. We launch $32 \cdot 5 = 160$ threads (e.g. 5 blocks per grid and one warp per block) and get the following memory access pattern (listed is the thread index inside a warp, i.e. the lane variable)

$$\begin{aligned} \text{Warp: } 0 &\begin{bmatrix} 0 & 1 & & & & & & & & & & & \end{bmatrix}, \\ \text{Warp: } 1 &\begin{bmatrix} & & 0 & 1 & 2 & & & & & & & & \end{bmatrix}, \\ \text{Warp: } 2 &\begin{bmatrix} & & & & & 0 & 1 & 2 & & & & & \end{bmatrix}, \\ \text{Warp: } 3 &\begin{bmatrix} & & & & & & & & 0 & 1 & 2 & & \end{bmatrix}, \\ \text{Warp: } 4 &\begin{bmatrix} & & & & & & & & & & & 0 & 1 \end{bmatrix}. \end{aligned}$$

Clearly, the memory access pattern into val is now more sequential. However, the performance of this approach, as numerical experiments suggest, is from slightly worse (for the example given in section 3.1.1) to significantly worse (for the example given in section 3.2). This, is also apparent from the Tables 4 and 5.

However, this is far from surprising since the number of non-zero elements in a row are not larger than 32. In this regime we would expect the code to be significantly more competitive. However, the code given in algorithm 9 can be easily optimized to our problems. This is done by setting SUB = 4 and modifying the code as is indicated in the comments. Following this modification, our code is from slightly (for the example given in section 3.2) to significantly faster (for the example given in section 3.1.1) than the implementation of Cusparse. A more detailed performance comparison can be found in Tables 4 and 5. It is quite interesting to see that even the naive method gives some speedup compared to the Cusparse implementation in the case of the problem of the two spins in a spin bath example (see section 3.1.1).

Algorithm 9 Improved kernel with a more sequential memory access pattern.

```
#define SUB 32

void __global__ k_csrmv_w(int N, fp_real alpha, fp* val, int* row_ptr,
                          int* col_ind, fp* d_x, fp_real beta, fp* d_y)
{
        __shared__ fp vals [512];
    int thread_id = blockDim.x * blockIdx.x + threadIdx.x ;
    int warp_id = thread_id / SUB;
    int lane = thread_id & (SUB - 1);
    int row = warp_id;

    if ( row < N )
    {
        int row_start = row_ptr[row];
        int row_end   = row_ptr[row + 1];

        /* compute partial sum */
        vals [ threadIdx.x ] = 0;
        for ( int jj = row_start + lane ; jj < row_end ; jj += SUB)
            vals[ threadIdx.x ] += alpha*val[ jj ] * d_x[ col_ind[ jj ]];

        /* compute total sum */
        /* remove for SUB=4 */
        if ( lane < 16) vals[ threadIdx.x ] += vals[ threadIdx.x + 16];
        /* remove for SUB=4 */
        if ( lane < 8)  vals[ threadIdx.x ] += vals[ threadIdx.x + 8];
        /* remove for SUB=4 */
        if ( lane < 4)  vals[ threadIdx.x ] += vals[ threadIdx.x + 4];
            if ( lane < 2)  vals[ threadIdx.x ] += vals[ threadIdx.x + 2];
            if ( lane < 1)  vals[ threadIdx.x ] += vals[ threadIdx.x + 1];

        /* store in global memory */
        if ( lane == 0)
            d_y[ row ] = beta*d_y[row] + vals [ threadIdx.x ];
    }
}
```

Implementation	Execution time	Speedup
Cusparse	∼10.5 ms	
Naive	∼18.5 ms	∼0.6
Sequential memory access (SUB = 32)	∼31.0 ms	∼0.3
Sequential memory access (SUB = 4)	∼8.1 ms	∼1.3

Table 4: Performance comparison for different implementations of the CSR matrix-vector multiplication (discretized Laplace operator, $N = 128$).

Implementation	Execution time	Speedup
Cusparse	∼13.5 ms	
Naive	∼11.6 ms	∼1.2
Sequential memory access (SUB = 32)	∼14.5 ms	∼0.9
Sequential memory access (SUB = 4)	∼7.3 ms	∼1.8

Table 5: Performance comparison for different implementations of the CSR matrix-vector multiplication (two spins in a spin bath, $L = 18$).

4.4 Numerical implementation

4.4.1 Real Leja point method

To compute a number of Leja points, we have to maximize the function

$$f(z) := \prod_{j=0}^{n-1} |z - z_j|$$

on the set $\partial K \subset\subset \mathbb{C}$. Naively, we would use the derivative test to find a maximum. However, due to the large number of terms, if n is large, this method is inefficient as well as introduces round-off errors. Therefore, we take an alternate approach that extracts the Leja points from a uniformly distributed discrete grid. This, so called discrete Leja points, are given, e.g., in [9]. In what follows we shall keep in mind that the Leja points lie on the boundary of K exclusively.

Definition 48. (Discrete Leja points). Suppose $(y_i)_{i=0}^{M}$ is a grid of uniformly distributed points on ∂K, where $K \subset\subset \mathbb{C}$. Then $(z_i)_{i=0}^{N}$ is called a **list of discrete Leja points** if $z_0 \in \{z \in \mathbb{C} : \max_{z\in\partial K} |z|\}$ and

$$\prod_{k=0}^{n-1} |z_n - z_k| = \max_{i\in\{0,\dots,M\}} \prod_{k=0}^{n-1} |z_i - z_k|.$$

It is obvious that for $N \ll M$ a list of discrete Leja points approximates a sequence of Leja points. We will use this method in all our implementations. However, for the sake of completeness it should be noted that other methods have been devised. For example, the Fast Leja points that are given in [7].

It should again be emphasized that all computations involving Leja points in this section are done in accord with theorem 36 on the compact set $K = [-2, 2]$. For the purpose of computation of the discrete Leja points a program is provided, that writes a given number of Leja points in a plain binary format (the suffix denotes the format, i.e. real_leja_d.bin holds double precision Leja points whereas real_leja_s.bin holds single precision Leja points).

As mentioned in sections 2.4 and 2.5 we try to approximate the discrete Schrödinger equation or the discretization of a partial differential equation by a number of matrix functions (usually some form of matrix exponential and the φ functions). These matrix functions are then evaluated by using an approximation with Newton polynomials. It should be duly noted that since, for example, the matrix H considered in section 2.4 is Hermitian it's eigenvalues lie on the real line. Therefore, the use of the Real Leja point method is justified as can be easily seen from theorem 40.

For the sake of simplicity we limit ourselves to the case where a single matrix function has to be computed (as is the case for example for the problems discussed in section 3.1.1). The algorithm presented here is based on [9] and its pseudo-code is given in algorithm 10.

The real numbers γ and c are such that all eigenvalues of A lie inside $[c - 2\gamma, c + 2\gamma]$ and are computed by Gershgorin's theorem. Clearly then the eigenvalues of

$$\frac{1}{\gamma}A - \frac{c}{\gamma}I$$

Algorithm 10 Generic algorithm in pseudo-code.

```
1  x = read_leja_points()
2  d = comp_div_diff()
3  A = read_matrix()
4  p = read_initial_value()
5  h = T/N
6
7  for i=0:N
8      t = i*t
9      p = d[0]*p
10     for m=1:MAX_NUMOF_LEJA_POINTS
11         alpha = 1.0/gamma
12         beta  = -c/gamma - x[m-1]
13         y = alpha*A(t+h) - beta*y
14         p = p + d[m]*y
15
16         if abs(d[m])*norm_2(y) < tol
17             break
```

lie inside the interval $[-2, 2]$.

The computation of the divided differences is done in the program itself (to accommodate different matrix functions). To that end, the recursion given in theorem 21 on page 10 is most easily implemented and is therefore used in all our programs.

Since ultimately our goal is to use the CUDA sparse matrix library (see [1]) in the GPU implementation, we limit ourselves to matrices described in the so called CSR format (Compressed sparse row format). It should also be noted, that in all programs zero-based indexing is used. That is, the first element of an array is indexed by 0, as is consistent with the standard convention in C/C++.

As given in algorithm 10 the algorithm can be implemented on almost any computer system. However, for our purpose it is crucial to identify the part of the algorithm which, when parallelized, would give us a significant speed increase. To that end let us analyze the different types of expressions present in the algorithm and its approximate runtime. Let us suppose that N is the size of a vector and M the number of nonzero elements in A. As usual we will count one addition and one multiplication as a single operation. This analysis is presented in Table 6.

Expression	Lines	Runtime
Vector addition	9, 12	N
Matrix multiplication	11	$M + N$
Norm	14	N

Table 6: Analysis of algorithm 10.

Since usually $M > 5N$, it is clear that we have to focus on parallelization of the matrix multiplications.

On the CPU we use OpenMP (see e.g. [4]) to parallelize the matrix-vector multiplication (this implementation is also used in section 5 to benchmark the GPU implementations against the CPU). To emphasize this, we also use a parallel implementation on the CPU such that the program is distributed to the different cores. This parallelization is however, in scale, significantly different from the parallelization on the GPU.

It should be noted, that the actual C/C++ program is more abstract than suggested in this section. It employs some object oriented paradigms to facilitate the implementation of exponential integrators that are more complex than those described in this section. In addition it enables to use different matrix multiplication algorithms for a given exponential integrator with a minimum of effort. However, the challenge in parallelization of the problem remains essentially the same.

4.4.2 Tree summation

Tree summation has been proposed as a method to avoid rounding errors if a large number of floating point numbers have to be summed up (see e.g. [18], or [15] for a simple implementation). Such an approach has the advantage that it is per construction very well parallelizable. In our situation, the summation of a large number of items arises in the computation of the Euclidean norm of a vector. This is among the operations to be performed in algorithm 10 on line 16.

The principle idea of a summation tree is to first compute the partial sums

$$S_{1;k} := \sum_{i=kn+1}^{kn+n} \overline{\psi_i}\psi_i, \qquad 0 \leq k \leq \frac{N}{n} - 1,$$

where n is small compared to $N = n^m$. Then, summing up these partials sums we get a new set of partial sums. Continuing until only one partial sum is left, we have

$$S_{j;k} := \sum_{i=kn+1}^{kn+n} S_{j-1;i}, \qquad 0 \leq k \leq \frac{N}{n^j} - 1,\ 1 \leq j \leq m,$$

where

$$\sum_{i=1}^{N} \overline{\psi_i}\psi_i = S_{m;0}.$$

The above method has been modified such that it generalizes to arbitrary N (not only N that can be written as n^m for some n). This, for some N, has the effect that the grows of the tree is stopped well before only a single element is left. In this case the remaining elements are summed up in series. Numerical experiments, however, indicate that this does not impede performance.

4.4.3 Single GPU matrix-vector multiplication

Our goal in this section is the parallelization of the matrix-vector multiplication (line 11 in algorithm 10) to a single GPU. The implementation on a single GPU is straight forward using the function (this is in fact a BLAS Level 2 function and is included in the Cusparse library)

```
cusparseStatus_t cusparse{S,D,C,Z}csrmv(cusparseHandle_t handle,
        cusparseOperation_t transA, int m, int n, float alpha,
        const cusparseMatDescr_t *descrA, const float *csrValA,
        const int *csrRowPtrA, const int *csrColIndA, const float *x,
        float beta, float *y )
```

as can be found in [1, p. 34]. The above mentioned function computes

$$\alpha Ax + \beta y$$

and stores the result in y. The Cusparse library already takes care of parallelizing the matrix-vector multiplication to a single GPU. The matrix is given in the CSR format and it is expected that all pointers are already stored in memory on the device[17].

However, in certain cases it is advantageous to exploit that a given matrix has a very regular structure (this is the case in the example introduced in section 3.2). It is then possible to exploit the structure of this matrix such that no information pertaining to the matrix is stored in memory (all the necessary data to perform matrix-vector multiplication are stored directly in the program code). Such an implementation for the example introduced in section 3.2 can be found in algorithm 11.

[17] We usually refer to the CPU as the host, whereas the GPU is referred to as the device.

Algorithm 11 Device kernel for the matrix-vector multiplication of section 3.2 ($N = 2^n$ is assumed).

```
__global__ void k_csrmv_laplace3(fp_real alpha, fp_real beta,int N,
                                 fp* x, fp* y)
{
        int i   = blockDim.x*blockIdx.x + threadIdx.x;
        int i1  = i-N*N;
        int i2  = i-N;
        int i3  = i-1;
        int i4  = i+1;
        int i5  = i+N;
        int i6  = i+N*N;
        int n   = N*N*N;

        fp_real f    = alpha*((fp_real)N+1)*((fp_real)N+1);
        if(i1 >= 0)
                ya += x[i1];
        if(i2 >= 0  && ((int)ceilf(((float)i+1.0)/((float)N)) & (N-1)) != 1)
                ya += x[i2];
        if(i3 >= 0 && ((i+1) & (N-1)) != 1)
                ya += x[i3];
        if(i4 < n && ((i+1) & (N-1)) != 0)
                ya += x[i4];
        if(i5 < n && ((int)ceilf(((float)i+1.0)/((float)N)) & (N-1)) != 0)
                ya += x[i5];
        if(i6 < n)
                ya += x[i6];

        y[i] = beta*y[i] + f*ya;
}
```

Algorithm 12 Fast modulo operation for $N = 2^n$.

```
i % N = i & (n-1)
```

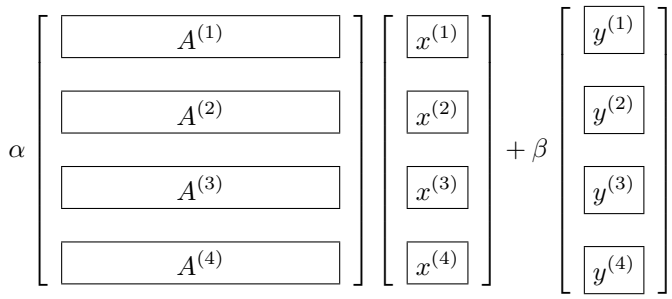

Figure 5: Illustrates of the parallelization of the multi GPU matrix-vector multiplication for 4 GPUs.

One might expect that such an implementation does not give a speed advantage if performed on the CPU (in fact the numerical experiments done in section 5 show that the performance decreases considerably). However, the GPU is vastly superior on raw number crunching ability and therefore we expect a performance increase for the GPU implementation of this method (this is confirmed in section 5.2.2 where the speedup is found to be approximately equal from 2 to 3, as compared to the CSR method).

For the purpose of writing a GPU kernel it should be noted that to compute the modulo operation takes many instructions (see [2, pp. 50-51]). Thus, if N is a multiple of 2, i.e. $N = 2^n$, we can replace the modulo operation by a bit-wise AND operation. In C/C++ code, this equivalence is stated in algorithm 12.

The implementation suggested here, as given in algorithm 11, is also used in the performance comparison in section 5.

4.4.4 Multiple GPU matrix-vector multiplication

In order to use multiple GPUs we have to take care of distributing the workload across multiple GPUs. To that end we split the matrix multiplication in m sub-matrices (the case $m = 4$ is illustrated in Figure 5).

It is a priori not clear whether this is a viable option. Clearly, we can speed up the matrix multiplication but after every step we have to copy back the result to the host and copy the result of all the other GPUs to every other device. Naively, this would require the transfer of

$$(m-1)\,N$$

floating point numbers. This is still a smaller quantity than $N + M$. Nevertheless, we have to take into account that random memory access is faster than copying memory over the PCI-E bus.

However, it is often possible to only transfer the subset of the data that has been changed (i.e., that are affected as a result of the multiplication by one of the sub-matrices illustrated in Figure 5). This method is also suggested in [19]. The numerical experiments done in this paper and on the problem given in section 3.2 suggest the validity of this approach. However, in section 3.1.1 we introduced a problem that highly mixes the different states (this is not surprising for a discrete quantum many-body problem). In this case the numerical experiments done in this master thesis show a slight performance advantage of the naive approach, i.e. making a full copy of all data. In some situation this problem can be mitigated, for example by choosing single precision floating point number. However, in general this poses a quite serious obstacle for parallelization such problems across multiple GPUs.

Finally, it seems instructive to compare the number of code lines written in each of the approaches discussed in this section. The CPU implementation of the matrix-vector multiplication as well as the

single GPU implementation are both done in approximately 100 lines of code. It should be mentioned, however, that this excludes code to copy the matrices and vectors to and from the GPU (approximately 50 lines of code) and the tree summation algorithm discussed in the previous section (150 lines of code). All together the single GPU implementation is implemented in about 300 lines of code. The "fast" version of the single GPU is implemented in about 200 lines (excluding host-device copy and tree summation). The multiple GPU implementation consists of about 400 lines of code. In this context it should be noted, that the implementation of the multiple GPU code with the CUDA SDK 4.0 is shorter as well as significantly more readable compared to the original multiple GPU implementation using pthreads and the CUDA SDK 3.2. This is due to the fact that the multi GPU programming model of the CUDA SDK 4.0 does not require one thread per device and it is therefore more convenient to switch between different GPUs.

In the above considerations, code common to all three implementations is not counted towards a specific implementation. In total about 6000 lines of C++/CUDA code has been written in the course of this master thesis.

5 GPU/CPU performance comparison

In this section we will compare the performance of computing the matrix exponential

$$e^{-itA}\psi_0$$

for the the problem given in section 3.1.1 as well as the matrix exponential

$$e^{-tA}u_0$$

for the problem given in section 3.2. We will limit ourselves to the computation of the two exponential functions, since from a performance standpoint, nothing is gained by employing arbitrary matrix functions[18].

5.1 Hardware

This section summarizes the hardware that is used in the following to make all performance comparisons. We will exclusively use the two hardware configurations listed in Tables 7 and 8 respectively. In both cases Nvidia GPUs and the corresponding CUDA development platform is employed.

Since the multi-GPU programming has been extensively simplified, for some examples, the CUDA 4.0 SDK is necessary. However, as of now, compute capability 1.1 is sufficient to run all the applications (i.e. no Fermi or Tesla specific instructions are employed).

Component	Manufacturer	Model
System	FluiDyna	TWS 4xC1060-2IQ-24
OS	-	OpenSuse 11
CPU	Intel	2x Xeon X5570 (2.93 GHz, 2x 4 cores)
GPU	Nvidia	4x NVIDIA Tesla C0160, 4 GB (240 stream processors)
RAM	Crucial	24 GB GDDR3-1333

Table 7: Hardware specifications (server system).

Component	Manufacturer	Model
OS	-	Windows 7
CPU	Intel	i7 920 (2.66 GHz, 4 cores)
GPU	Gainward/Nvidia	GeForce 9800 GTX+, 512 MB (128 stream processors)
RAM	Crucial	4 GB DDR3-1066

Table 8: Hardware specifications (consumer system).

These two configurations correspond to a consumer and a server system respectively. The consumer system is included for completeness. Since the GeForce 9800 GTX+ only supports single precision floating point arithmetic, its usefulness for high performance computations seems limited. However, since it performed quite well in the Monte Carlo computations done in [15], we believe it is, if nothing else, interesting to see how it performs in the implementation of the exponential integrators considered in this master thesis. A picture of the server system is given in Figure 6.

[18] As can be seen in section 4.4, the runtime of the algorithm does not depend on the actual value of the divided differences.

Figure 6: FluiDyna TWS 4xC1060-2IQ-24 (picture taken from `http://www.fluidyna.de/`).

5.2 Comparison

In this section all compilations are done either with the g++ compiler or the Nvidia nvcc compiler[19] All compilations are done with the -O3 optimization flag and use OpenMP where necessary (with the -fopenmp and /openmp flag respectively).

It should be noted that due to memory limitations not all GPU examples can be run on the consumer system. If this is the case the results are, without any further remark, dropped from the performance tables. All speedups are computed with respect to the CPU parallelization that is run on the i7 920 (4 core) processor.

5.2.1 Spin bath example

First, let us consider the example given in section 4.4. We compute the matrix function

$$e^{-itA}\psi_0,$$

where ψ_0 is randomly determined as in section 4.4. The results are listed in Table 9 (for $L = 18$) and in Table 10 (for $L = 21$) respectively.

The performance gain in this case is not as impressive as for the problem considered in section 5.2.2. The speedup compared to a single GPU varies, depending on the method and the problem size, between 2.5 and 8 (compared to a dual socket workstation gives a speedup of about 1.5 to 5). Especially in the single precision domain the parallelization to two GPUs is close to the theoretical speedup of 2. The parallelization to three or more GPUs does not improve the performance due to the bandwidth problem that is further discussed in section 4.4.4.

It should also be noted that for large problems as well as problems involving double precision floating point numbers the handwritten method introduced in section 4.3 gives a significant speedup compared to Cusparse (the speedup is of the order of 2).

[19] On Windows that implies that the underlying compiler used is cl.exe from the Microsoft Windows SDK 2008.

Algorithm	Model	Precision	Cores/Stream processors	Execution time	Speedup
CPU (OMP)	Intel Core i7	Single	4	∼11.3 s	
CPU (OMP)	2 x Intel Xenon E5620	Single	8	∼8.4 s	∼1.3
CPU (OMP)	Intel Core i7	Double	4	∼10.3 s	
CPU (OMP)	2 x Intel Xenon E5620	Double	8	∼6.2 s	∼1.7
GPU	1x Tesla C1060	Single	240	∼2.2 s	∼5.1
GPU	1x Tesla C1060	Double	240	∼5.4 s	∼1.9
GPU	2x Tesla C1060	Single	2x 240	∼1.2 s	∼9.4
GPU	2x Tesla C1060	Double	2x 240	∼3.0 s	∼3.4
GPU	4x Tesla C1060	Single	4x 240	∼1.2 s	∼9.4
GPU	4x Tesla C1060	Double	4x 240	∼3.0 s	∼3.4
GPU (Handwritten)	1x Tesla C1060	Single	240	∼2.2 s	∼5.1
GPU (Handwritten)	1x Tesla C1060	Double	240	∼3.1 s	∼3.3
GPU	1x GeForce 9800 GTX+	Single	128	∼3.7 s	∼3.0

Table 9: Performance comparison of a two spin system in a spin bath ($L = 18$, $t = 20$, tol $= 10^{-5}$), compiled with option -O3.

Algorithm	Model	Precision	Cores/Stream processors	Execution time	Speedup
CPU (OMP)	Intel Core i7	Single	4	∼61 s	
CPU (OMP)	2 x Intel Xenon E5620	Single	8	∼44 s	∼1.4
CPU (OMP)	Intel Core i7	Double	4	∼63 s	
CPU (OMP)	2 x Intel Xenon E5620	Double	8	∼33 s	∼1.9
GPU	1x Tesla C1060	Single	240	∼15 s	∼4.1
GPU	1x Tesla C1060	Double	240	∼23 s	∼2.7
GPU	2x Tesla C1060	Single	2x 240	∼10 s	∼6.1
GPU	2x Tesla C1060	Double	2x 240	∼15 s	∼4.2
GPU	4x Tesla C1060	Single	2x 240	∼10 s	∼6.1
GPU	4x Tesla C1060	Double	2x 240	∼15 s	∼4.2
GPU (Handwritten)	1x Tesla C1060	Single	240	∼8 s	∼7.6
GPU (Handwritten)	1x Tesla C1060	Double	240	∼13 s	∼4.8

Table 10: Performance comparison of a two spin system in a spin bath ($L = 21$, $t = 10$, tol $= 10^{-5}$), compiled with option -O3.

5.2.2 Laplace example

In this case we employ the discretized Laplacian from the example introduced in section 3.2. Our goal is to compute the matrix function

$$e^{-hA}u_0,$$

where u_0 is the discretization of

$$(x, y, z) \mapsto \sin(2\pi x).$$

The matrix, as a discretization of the Laplacian, has a regular structure that is quite easily analyzable. Since GPUs are far superior to the CPU in raw number-crunching ability, in addition to the method based on the CSR representation of the matrix A we have implemented the matrix-vector multiplication of A with a given vector as a kernel on the GPU (see section 4.4.3 for details). Since this method is faster on the GPU we denote it by "Fast" in all tables that follow. It should be noted that this is despite the fact that the method is, in fact, slower on the CPU (and thus is not listed for computations done on the CPU).

Tables 11 and 12 list the execution time and speedup for the case of 128 and 256 discretization points in any dimension respectively. This corresponds to a vector size of roughly 10^6 and $16 \cdot 10^6$ elements.

From these tables we can conclude that for the problems considered in this section, quite an significant speedup can be achieved. A single C1060 GPU is roughly equivalent to a system with 3 processors (with 4 cores each). The problem scales very well to multiple GPUs where a speedup of about 4 (near the theoretical limit) compared to a single GPU is realized (This would correspond to a cluster of 16 processors).

However, what is even more impressive is the raw number crunching performance of the C1060. Using this "Fast" algorithm a single C1060 rivals the performance of between 5 and 8 processors. At current no multi GPU implementation for this approach has been constructed. However, it is to be expected that such an implementation would scale equally well (totaling to a speedup of about 20-35).

6 Conclusion

In this master-thesis, we have shown that parallelizing sparse matrix-vector multiplications to GPU(s) in the hope to increase the performance of exponential integrators, is certainly a viable option (at least for the Real Leja point method). As is shown in section 5, we get quite significant performance improvements, as compared to a CPU implementation. Especially, most promising are the structured real matrices that are the result of discretizing differential operators. In this case speedups of 8 have been observed (see Table 12). This example, as well as the discussion in section 4.3 imply that we can achieve even better performance, if we use some optimizations that takes the problem at hand into account (e.g. writing algorithms that exploit the knowledge of properties like the nonzero elements per row or the special structure of some matrices).

It is also our hope that the introduction to the theory of Leja points given in section 2 can act as a starting point for more theoretical considerations in the field of interpolatory exponential integrators. We have therefore tried to give a coherent treatment of results, that are usually distributed across many papers as well as books, that culminates in the theorem 36 (which is essential for numerical implementations) as well as corollary 39 (which gives us a theoretical justification for choosing Leja points as interpolation nodes).

Algorithm	Model	Precision	Cores/Stream processors	Execution time	Speedup
CPU (OMP)	Intel Core i7	Single	4	∼10.2 s	
CPU (OMP)	2 x Intel Xenon E5620	Single	8	∼ 7.3 s	∼1.4
CPU (OMP)	Intel Core i7	Double	4	∼16.7 s	
CPU (OMP)	2 x Intel Xenon E5620	Double	8	∼13.3 s	∼1.3
GPU	1x Tesla C1060	Single	240	∼3.5 s	∼2.9
GPU	1x Tesla C1060	Double	240	∼5.4 s	∼3.1
GPU	2x Tesla C1060	Single	2x 240	∼1.2 s	∼8.5
GPU	2x Tesla C1060	Double	2x 240	∼3.1 s	∼5.4
GPU	4x Tesla C1060	Single	4x 240	∼1.4 s	∼7.3
GPU	4x Tesla C1060	Double	4x 240	∼2.1 s	∼8.0
GPU (Fast)	1x Tesla C1060	Single	240	∼2.4 s	∼4.3
GPU (Fast)	1x Tesla C1060	Double	240	∼2.4 s	∼7.0
GPU (Handwritten)	1x Tesla C1060	Single	240	∼3.3 s	∼3.1
GPU (Handwritten)	1x Tesla C1060	Double	240	∼4.3 s	∼3.9
GPU	1x GeForce 9800 GTX+	Single	128	∼3.5 s	∼2.9

Table 11: Performance comparison of the discretized Laplacian ($N = 128$, $h = 0.1$, tol $= 10^{-5}$), compiled with option -O3.

Algorithm	Model	Precision	Cores/Stream processors	Execution time	Speedup
CPU (OMP)	Intel Core i7	Single	4	∼58 s	
CPU (OMP)	2 x Intel Xenon E5620	Single	8	∼47 s	∼1.2
CPU (OMP)	Intel Core i7	Double	4	∼98 s	
CPU (OMP)	2 x Intel Xenon E5620	Double	8	∼76 s	∼1.3
GPU	1x Tesla C1060	Single	240	∼27 s	∼2.1
GPU	1x Tesla C1060	Double	240	∼31 s	∼3.2
GPU	4x Tesla C1060	Single	4x 240	∼6 s	∼7.8
GPU	4x Tesla C1060	Double	4x 240	∼8 s	∼12.3
GPU (Fast)	1x Tesla C1060	Single	240	∼12 s	∼4.8
GPU (Fast)	1x Tesla C1060	Double	240	∼12 s	∼8.2

Table 12: Performance comparison of the discretized Laplacian ($N = 256$, $h = 0.01$, tol $= 10^{-5}$), compiled with option -O3.

References

[1] CUDA CUSPARSE Library. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUSPARSE_Library.pdf (Last retrieved July 11, 2011).

[2] NVIDIA CUDA, Best Practice Guide. http://www.nvidia.com/object/cuda_develop.html (Last retrived July 11, 2011).

[3] NVIDIA CUDA C Programming Guide. http://developer.nvidia.com/cuda-downloads (Last retrieved July 11, 2011).

[4] The OpenMP API specification for parallel programming. http://openmp.org/wp/resources/ (Last retrieved July 11, 2011).

[5] Vladimir V. Andrievskii and H.P. Blatt. *Discrepancy of Signed Measures and Polynomial Approximation (Springer Monographs in Mathematics)*. Springer, 1st edition, 2010.

[6] D.H. Armitage and S.J. Gardiner. *Classical potential theory*. Springer-Verlag New York Berlin Heidelberg, 2000.

[7] J. Baglama, D. Calvetti, and L. Reichel. Fast Leja points. *Electronic Transactions on Numerical Analysis*, 7:124–140, 1998.

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

[9] M. Caliari, M. Vianello, and L. Bergmaschi. Interpolating discrete advection-diffusion propagators at Leja sequences. *Journal of Computational and Applied Mathematics*, 172:79–99, 2004.

[10] J. Górski. Sur l'équivalence de deux constructions de la fonction de Green généralisée. *Ann. Soc. Pol. Math.*, 21:70–73, 1948.

[11] R. Grothmann. Interpolation Points and Zeros of Polynomials in Approximation Theory. http://www.ku-eichstaett.de/Fakultaeten/MGF/Mathematik/Grothmann/Veroeffentlichung/HF_sections/content/Habilitation.pdf.

[12] R. Grothmann. Distribution of interpolation points. *Arkiv för Mathematik*, 34(1):103–117, 1996.

[13] L.L. Helms. *Potential Theory*. Springer-Verlag London Limited, 2009.

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

[15] L. Einkemmer. Monte Carlo methods. Bachelor thesis (University of Innsbruck), 2010.

[16] F. Leja. Sur les suites de polynômes, les ensembles fermés et la fonction de Green. *Ann. Soc. Pol. Math.*, 12:57–71, 1934.

[17] F. Leja. Sur certaines suites liées aux ensembles plans et leur application á la représentation conforme. *Ann. Polon. Math.*, 4:8–13, 1957.

[18] P. Linz. Accurate Floating-Point Summation. *Coummunications of the ACM*, (13):361–362, 1970.

[19] A. Martinez, L. Bergamaschi, M. Caliari, and M. Vianello. Efficient massively parallel implementation of the relpm exponential integrator for advection-diffusion models. Technical report, 2006.

[20] M.A. Nielson and I.L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[21] H. De Raedt and K. Michielsen. Computational Methods for Simulating Quantum Computers. *Handbook of Theoretical and Computational Nanotechnology (American Scientific Publishers)*, 2008.

[22] L. Reichel. Newton interpolation at Leja points. *Bit Numerical Mathematics*, 30(2):332–246, 1990.

[23] E.B. Saff and V. Totik. *Logarithmic Potentials with External Fields.* Springer, 1st edition, 2010.

[24] E.D. Solomentsev. Green function – Green function in function theory. `http://eom.springer.de/g/g045090.html`, 2002.

[25] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis.* Springer-Verlag New York Berlin Heidelberg, 1993.