PACO 2015
July 6-7

Effects of dynamic frequency scaling of Nvidia GPUs during the computation of the generalized matrix sign function

Martin Köhler

Max-Planck-Institute Magdeburg

Outline

Introduction

Generalized Matrix Sign Function

Matrix Sign Function

Let $A \in \mathbb{R}^{n \times n}$ be a matrix with no eigenvalues on the imaginary axis with the *Jordan canonical form*

$$Y \begin{pmatrix} J_1 & 0 \\ 0 & J_2 \end{pmatrix} Y^{-1} = A,$$

where $\Lambda(J_1) \subset \mathbb{C}_-$ and $\Lambda(J_2) \subset \mathbb{C}_+$. Then $\operatorname{sign}(A)$ is given by

$$\operatorname{sign}(A) := Y \begin{pmatrix} -I_1 & 0 \\ 0 & I_2 \end{pmatrix} Y^{-1},$$

where $\dim(I_i) = \dim(J_i)$, $i = 1, 2$.

Introduction

Generalized Matrix Sign Function

Matrix Sign Function

Let $A \in \mathbb{R}^{n \times n}$ be a matrix with no eigenvalues on the imaginary axis with the *Jordan canonical form*

$$Y \begin{pmatrix} J_1 & 0 \\ 0 & J_2 \end{pmatrix} Y^{-1} = A,$$

where $\Lambda(J_1) \subset \mathbb{C}_-$ and $\Lambda(J_2) \subset \mathbb{C}_+$. Then $\operatorname{sign}(A)$ is given by

$$\operatorname{sign}(A) := Y \begin{pmatrix} -I_1 & 0 \\ 0 & I_2 \end{pmatrix} Y^{-1},$$

where $\dim(I_i) = \dim(J_i)$, $i = 1, 2$.

Generalized Matrix Sign Function [Gardiner, Laub '86]

Let (A, B) be a matrix pencil with no eigenvalues on the imaginary axis then $\operatorname{sign}(A, B)$ is given by

$$\operatorname{sign}(A, B) = B \operatorname{sign}\left(B^{-1}A\right) = \operatorname{sign}\left(AB^{-1}\right) B.$$

Introduction

Generalized Matrix Sign Function

Applications

- Solution of Riccati equations: [Gardiner, Laub '86]

$$A^T XE + E^T XA - E^T XGXE + Q = 0,$$

Introduction

Generalized Matrix Sign Function

Applications

- Solution of Riccati equations: [Gardiner, Laub '86]

$$A^T XE + E^T XA - E^T XGXE + Q = 0,$$

- Solution of stable Lyapunov equations: [Benner, Quintana-Ortí '98]

$$A^T XE + E^T XA + Q = 0,$$

Introduction

Generalized Matrix Sign Function

Applications

- Solution of Riccati equations: [Gardiner, Laub '86]

$$A^T XE + E^T XA - E^T XGXE + Q = 0,$$

- Solution of stable Lyapunov equations: [Benner, Quintana-Ortí '98]

$$A^T XE + E^T XA + Q = 0,$$

- Spectral Division, [Sun, Quintana-Ortí '04]

Introduction

Generalized Matrix Sign Function

Applications

- Solution of Riccati equations: [Gardiner, Laub '86]

$$A^T XE + E^T XA - E^T XGXE + Q = 0,$$

- Solution of stable Lyapunov equations: [Benner, Quintana-Ortí '98]

$$A^T XE + E^T XA + Q = 0,$$

- Spectral Division, [Sun, Quintana-Ortí '04]
- Fast approximation of the generalized Schur decomposition: [Benner, K., Saak '13]

$$(A, B) = (Q^T SZ, Q^T TZ).$$

Introduction

Newton-Method for sign (A, B)

From $\operatorname{sign}(A)^2 = I$ follows the Newton scheme:

$$A_0 \leftarrow A, \quad A_{k+1} \leftarrow \frac{1}{2}\left(A_k + A_k^{-1}\right), \quad k = 0, 1, 2, \ldots$$

to compute the sign of a matrix.

Introduction

Newton-Method for sign (A, B)

From $\operatorname{sign}(A)^2 = I$ follows the Newton scheme:

$$A_0 \leftarrow A, \quad A_{k+1} \leftarrow \frac{1}{2}\left(A_k + A_k^{-1}\right), \quad k = 0, 1, 2, \ldots$$

to compute the sign of a matrix.

The Generalized Sign function iteration:

$$A_0 \leftarrow A, \quad A_{k+1} \leftarrow \frac{1}{2}\left(A_k + BA_k^{-1}B\right), \quad k = 0, 1, 2, \ldots$$

Introduction

Newton-Method for sign (A, B)

From $\operatorname{sign}(A)^2 = I$ follows the Newton scheme:

$$A_0 \leftarrow A, \quad A_{k+1} \leftarrow \frac{1}{2}\left(A_k + A_k^{-1}\right), \quad k = 0, 1, 2, \ldots$$

to compute the sign of a matrix.

The Generalized Sign function iteration:

$$A_0 \leftarrow A, \quad A_{k+1} \leftarrow \frac{1}{2c_k}\left(A_k + c_k^2 B A_k^{-1} B\right), \quad k = 0, 1, 2, \ldots$$

where c_k is an additional scaling factor. Typical: $c_k = \left(\frac{|\det(A_k)|}{|\det(B)|}\right)^{\frac{1}{n}}$.

Introduction

Frequency Scaling on K20 GPUs

NVIDIA® K20 accelerators can adjust their GPU and Memory frequency:

- **GPU clock rate (operation mode):** 758MHz, 705MHz, 666MHz, 640MHz, and 614MHz with a memory clock rate of 2600MHz.
- **GPU clock rate (idle mode):** 324MHz with a memory clock rate of 324MHz.
- **GPU clock rate (emergency):** 378MHz or 127MHz.

Introduction

Frequency Scaling on K20 GPUs

NVIDIA® K20 accelerators can adjust their GPU and Memory frequency:

- **GPU clock rate (operation mode):** 758MHz, 705MHz, 666MHz, 640MHz, and 614MHz with a memory clock rate of 2600MHz.
- **GPU clock rate (idle mode):** 324MHz with a memory clock rate of 324MHz.
- **GPU clock rate (emergency):** 378MHz or 127MHz.

Frequency is scaled if:

- Power consumption is too high for a longer time span (>1 minute),
- Device temperature reaches 90 ℃.

Introduction

Frequency Scaling on K20 GPUs

NVIDIA® K20 accelerators can adjust their GPU and Memory frequency:

- **GPU clock rate (operation mode):** 758MHz, 705MHz, 666MHz, 640MHz, and 614MHz with a memory clock rate of 2600MHz.
- **GPU clock rate (idle mode):** 324MHz with a memory clock rate of 324MHz.
- **GPU clock rate (emergency):** 378MHz or 137MHz.

Frequency is scaled if:

- Power consumption is too high for a longer time span (>1 minute),
- Device temperature reaches 90 °C.

Reasons:

- High density server farms,
- Bad cooling system,
- Bad airflow design of the chassis,

combined with a continuous **high load**.

→ Extremely hardware dependent.

Introduction

Frequency Scaling on K20 GPUs

NVIDIA® K20 accelerators can adjust their GPU and Memory frequency:

- **GPU clock rate (operation mode):** 758MHz, 705MHz, 666MHz, 640MHz, and 614MHz with a memory clock rate of 2600MHz.
- **GPU clock rate (idle mode):** 324MHz with a memory clock rate of 324MHz.
- **GPU clock rate (emergency):** 378MHz or 137MHz.

Reasons:

- High density server farms,
- Bad cooling system,
- Bad airflow design of the chassis,

combined with a continuous **high load**.

→ Extremely hardware dependent.

Frequency is scaled if:

- Power consumption is too high for a longer time span (>1 minute),
- Device temperature reaches 90 ℃.

Aim:

Develop a GPU implementation which can handle (power- or) temperature-caused frequency scaling.

Basic Implementation

Straight-Forward Approach using LAPACK

$$A_0 \leftarrow A, \quad A_{k+1} \leftarrow \frac{1}{2c_k}\left(A_k + c_k^2 B A_k^{-1} B\right), \quad k = 0, 1, 2, \ldots$$

Basic Implementation

Straight-Forward Approach using LAPACK

$$A_0 \leftarrow A, \quad A_{k+1} \leftarrow \frac{1}{2c_k}\left(A_k + c_k^2 B A_k^{-1} B\right), \quad k = 0, 1, 2, \ldots$$

Solution of a linear system:

- LU - decomposition: `GETRF`,
- Forward/backward substitution: `GETRS`.

Basic Implementation

Straight-Forward Approach using LAPACK

$$A_0 \leftarrow A, \quad A_{k+1} \leftarrow \frac{1}{2c_k}\left(A_k + c_k^2 BX\right), \quad k = 0, 1, 2, \ldots$$

Solution of a linear system:

- LU - decomposition: `GETRF`,
- Forward/backward substitution: `GETRS`.

Matrix-Matrix product:

- $C := \alpha AB + \beta C$ in BLAS: `GEMM`
- Requires three non-overlapping memory locations of size n^2.

$\rightsquigarrow$ lower bound for the memory requirements on the device.

Basic Implementation

Straight-Forward GPU Approach using MAGMA

The matrix-matrix product requires $3 \cdot n^2$ memory on the device.

Basic Implementation

Straight-Forward GPU Approach using MAGMA

The matrix-matrix product requires $3 \cdot n^2$ memory on the device.

- LU decomposition works on n^2 memory,
- Forward/Backward substitution works on $2n^2$ memory.

$\rightarrow$ **Matrix-Matrix product is the limiting factor!**

Basic Implementation

Straight-Forward GPU Approach using MAGMA

The matrix-matrix product requires $3 \cdot n^2$ memory on the device.

- LU decomposition works on n^2 memory,
- Forward/Backward substitution works on $2n^2$ memory.

$\rightarrow$ **Matrix-Matrix product is the limiting factor!**

Maximum problem size on NVIDIA® Tesla K20m:

double precision	14 481
single precision	20 480

Basic Implementation

Straight-Forward GPU Approach using MAGMA

The matrix-matrix product requires $3 \cdot n^2$ memory on the device.

Generalized Sign Function on $3n^2$ device memory

1: Upload A_0 and B to the device.
2: **for** $k = 1, \ldots$ **do**
3: Copy B to $\tilde{B}$ on the device.
4: Use getrf_gpu from MAGMA to compute $LU = PA_k$,
5: Use getrs_gpu from MAGMA to solve $LUX = P\tilde{B}$,
6: Upload A_k again onto the location of LU,
7: Compute $A_{k+1} := \frac{1}{2c_k}(A_k + c_k^2 BX)$ using cublas?gemm,
8: Download A_{k+1} to the host and check convergence.
9: **end for**

Basic Implementation

Straight-Forward GPU Approach using MAGMA

The matrix-matrix product requires $3 \cdot n^2$ memory on the device.

Generalized Sign Function on $3n^2$ device memory

Upload A_0 and B to the device.
for $k = 1, \dots$ **do**
    Copy B to $\tilde{B}$ on the device.
    Use `getrf_gpu` from MAGMA to compute $LU = PA_k$,
    Use `getrs_gpu` from MAGMA to solve $LUX = P\tilde{B}$,
    Upload A_k again onto the location of LU,
    Compute $A_{k+1} := \frac{1}{2c_k}(A_k + c_k^2 BX)$ using `cublas?gemm`,
    Download A_{k+1} to the host and check convergence
end for

Only n^2 additional memory necessary to remove this transfer.

Basic Implementation

Straight-Forward GPU Approach using MAGMA

The matrix-matrix product requires $3 \cdot n^2$ memory on the device.

Generalized Sign Function on $3n^2$ device memory

Upload A_0 and B to the device.
for $k = 1, \ldots$ **do**
  Copy B to $\tilde{B}$ on the device.
  Use `getrf_gpu` from MAGMA to compute $LU = PA_k$,
  Use `getrs_gpu` from MAGMA to solve $LUX = P\tilde{B}$,
  Upload A_k again onto the location of LU,
  Compute $A_{k+1} := \frac{1}{2c_k}(A_k + c_k^2 BX)$ using `cublas?gemm`,
  Download A_{k+1} to the host and check convergence.
end for

Only n^2 additional memory necessary to remove this transfer.
Maximum problem size shrinks down to:

double precision	12 541
single precision	17 736

Basic Implementation

Observations and Pitfalls

- High memory requirements on the device,

Basic Implementation

Observations and Pitfalls

- High memory requirements on the device,
- Three sweeps over A_k to solve $A_k X = B$,

Basic Implementation

Observations and Pitfalls

- High memory requirements on the device,
- Three sweeps over A_k to solve $A_k X = B$,
- Stopping criterion $||A_{k+1} - A_k|| < \tau$ costs additional n^2 memory,

Basic Implementation

Observations and Pitfalls

- High memory requirements on the device,
- Three sweeps over A_k to solve $A_k X = B$,
- Stopping criterion $||A_{k+1} - A_k|| < \tau$ costs additional n^2 memory,
- Due to the involved large matrix-matrix products the device temperature might increase to a critical value.

Basic Implementation

Observations and Pitfalls

- High memory requirements on the device,
- Three sweeps over A_k to solve $A_k X = B$,
- Stopping criterion $||A_{k+1} - A_k|| < \tau$ costs additional n^2 memory,
- Due to the involved large matrix-matrix products the device temperature might increase to a critical value.

Goal:

Reduce the memory requirements to $2n^2 + \mathcal{O}(n)$ and increase the maximum problem dimension:

double precision	17 736
single precision	25 082

Basic Implementation

Observations and Pitfalls

- High memory requirements on the device,
- Three sweeps over A_k to solve $A_kX = B$,
- Stopping criterion $||A_{k+1} - A_k|| < \tau$ costs additional n^2 memory,
- Due to the involved large matrix-matrix products the device temperature might increase to a critical value.

Idea:

Replace LU-decomposition with forward/backward substitution by Gauss-Jordan-Elimination.

Basic Implementation

Observations and Pitfalls

- High memory requirements on the device,
- Three sweeps over A_k to solve $A_k X = B$,
- Stopping criterion $||A_{k+1} - A_k|| < \tau$ costs additional n^2 memory,
- Due to the involved large matrix-matrix products the device temperature might increase to a critical value.

Cheap convergence check:

Generalize $\operatorname{sign}(A)^2 = I$ to derive a memory efficient stopping criterion.

Basic Implementation

Observations and Pitfalls

- High memory requirements on the device,
- Three sweeps over A_k to solve $A_k X = B$,
- Stopping criterion $||A_{k+1} - A_k|| < \tau$ costs additional n^2 memory,
- Due to the involved large matrix-matrix products the device temperature might increase to a critical value.

Idea:

Continuous monitoring of the device temperature and movement of workload to the host.

Memory-Efficient Implementation

Gauss-Jordan-Elimination

Gauss-Jordan Elimination

The Gauss-Jordan is a rearrangement of the LU decomposition to compute A^{-1} without setting up L and U and only sweeping once over the matrix A.
$\rightarrow$ Cost: $2n^3$ flops.

Memory-Efficient Implementation

Gauss-Jordan-Elimination

Gauss-Jordan Elimination

The Gauss-Jordan is a rearrangement of the LU decomposition to compute A^{-1} without setting up L and U and only sweeping once over the matrix A.
$\rightarrow$ Cost: $2n^3$ flops.

Augmented Gauss-Jordan Elimination

The Augmented Gauss-Jordan Elimination scheme computes

$$X = A^{-1}B$$

without setting up A^{-1}, L or U.
$\rightarrow$ Cost: $3n^3$ flops.

Algorithm 1 Augmented Gauss-Jordan Elimination

Input: $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$, blocking parameter N_B.
Output: B overwritten by $A^{-1}B$.

1: Set $D := \begin{bmatrix} A & B \end{bmatrix} \in \mathbb{R}^{n\times n+m}$.
2: **for** $i = 1, 1+N_B, 1+2N_B, \ldots, n$ **do**
3: Partition D into

$$D := \left[\begin{array}{c|c|c|c} A_{11} & A_{12} & A_{13} & B_1 \\ \hline A_{21} & A_{22} & A_{23} & B_2 \\ \hline A_{31} & A_{32} & A_{33} & B_3 \end{array}\right],$$

where $A_{11} \in \mathbb{R}^{i-1\times i-1}$ and $A_{22} \in \mathbb{R}^{N_B\times N_B}$.
4: Update D by

$$D := \left[\begin{array}{c|c|c|c} A_{11} & A_{12} & A_{13} & B_1 \\ \hline A_{21} & A_{22} & 0 & 0 \\ \hline A_{31} & A_{32} & A_{33} & B_3 \end{array}\right] + \begin{bmatrix} -A_{12}A_{22}^{-1} \\ A_{22}^{-1} \\ -A_{32}A_{22}^{-1} \end{bmatrix} \begin{bmatrix} 0 & 0 & A_{23} & B_2 \end{bmatrix}.$$

5: **end for**

Algorithm 1 Augmented Gauss-Jordan Elimination

Input: $A \in \mathbb{R}^{n \times n}$, $B \in \mathbb{R}^{n \times m}$, blocking parameter N_B.
Output: B overwritten by $A^{-1}B$.

1: Set $D := \begin{bmatrix} A & B \end{bmatrix} \in \mathbb{R}^{n \times n+m}$

2: **for** $i = 1, 1 + N_B, 1 + 2$ [illegible]

3: Partition D into

$$D := \left[\begin{array}{c|c|c|c} A_{11} & A_{12} & A_{13} & B_1 \\ \hline A_{21} & A_{22} & A_{23} & B_2 \\ \hline A_{31} & A_{32} & A_{33} & B_3 \end{array}\right],$$

where $A_{11} \in \mathbb{R}^{i-1 \times i-1}$ and $A_{22} \in \mathbb{R}^{N_B \times N_B}$.

4: Update D by

$$D := \left[\begin{array}{c|c|c|c} A_{11} & A_{12} & A_{13} & B_1 \\ \hline A_{21} & A_{22} & 0 & 0 \\ \hline A_{31} & A_{32} & A_{33} & B_3 \end{array}\right] + \boxed{\begin{bmatrix} -A_{12}A_{22}^{-1} \\ A_{22}^{-1} \\ -A_{32}A_{22}^{-1} \end{bmatrix}} \begin{bmatrix} 0 & 0 & A_{23} & B_2 \end{bmatrix}.$$

5: **end for**

GPU Implementation:

- Asynchronous precomputation on the host,
- Remaining matrix-matrix-products on the device,
- **Requires $2nN_B$ additional memory on the device for intermediate results.**

Memory-Efficient Implementation

Gauss-Jordan-Elimination – Remarks

- Gauss-Jordan Elimination needs 12.5% more flops compared to LU+forward/backward substitution,☹
- Better GPU utilization $\rightarrow$ gains a higher performance. ☺
- Easily distributable across multiple GPUs, ☺
- Additional accumulation of A^{-1} costs only n^3 flops more.
- Larger memory requirements. ☹

Memory-Efficient Implementation

Gauss-Jordan-Elimination – Remarks

- Gauss-Jordan Elimination needs 12.5% more flops compared to LU+forward/backward substitution,☹
- Better GPU utilization $\rightarrow$ gains a higher performance. ☺
- Easily distributable across multiple GPUs, ☺
- Additional accumulation of A^{-1} costs only n^3 flops more.
- Larger memory requirements. ☹

Preliminary results on one GPU

- For small problems $n < 8\,000$, Gauss-Jordan is faster than the LU decomposition.
- Similar performance for larger problems ($n \geq 8\,000$).

Memory-Efficient Implementation

Gauss-Jordan-Elimination – Remarks

- Gauss-Jordan Elimination needs 12.5% more flops compared to LU+forward/backward substitution,☹
- Better GPU utilization → gains a higher performance. ☺
- Easily distributable across multiple GPUs, ☺
- Additional accumulation of A^{-1} costs only n^3 flops more.
- Larger memory requirements. ☹

Preliminary results on one GPU

- For small problems $n < 8\,000$, Gauss-Jordan is faster than the LU decomposition.
- Similar performance for larger problems ($n \geq 8\,000$).

Except of the scalability no advantage compared with LU, yet.

Memory-Efficient Implementation

Gauss-Jordan-Elimination – Remarks

- Gauss-Jordan Elimination needs 12.5% more flops compared to LU+forward/backward substitution,☹
- Better GPU utilization → gains a higher performance. ☺
- Easily distributable across multiple GPUs, ☺
- Additional accumulation of A^{-1} costs only n^3 flops more.
- Larger memory requirements. ☹

Preliminary results on one GPU

- For small problems $n < 8\,000$, Gauss-Jordan is faster than the LU decomposition.
- Similar performance for larger problems ($n \geq 8\,000$).

Except of the scalability no advantage compared with LU, yet.

Memory-Efficient Implementation

Asynchronous Matrix-Matrix Product

After computing $X = A^{-1}B$ we have on the device:

- n^2 memory in use to store X,
- $n^2 + 2nNB$ memory of intermediate data from the Gauss-Jordan-Elimination,
- no unmodified copy of A or B on the device.

Memory-Efficient Implementation

Asynchronous Matrix-Matrix Product

After computing $X = A^{-1}B$ we have on the device:

- n^2 memory in use to store X,
- $n^2 + 2nNB$ memory of intermediate data from the Gauss-Jordan-Elimination,
- no unmodified copy of A or B on the device.

$\rightarrow$ **Compute** $A_{k+1} := \frac{1}{2c_k}(A_k + c_k^2 BX)$ **with only** $2n^2 + 2nN_B$ **memory.**

Memory-Efficient Implementation

Asynchronous Matrix-Matrix Product

After computing $X = A^{-1}B$ we have on the device:

- n^2 memory in use to store X,
- $n^2 + 2nNB$ memory of intermediate data from the Gauss-Jordan-Elimination,
- no unmodified copy of A or B on the device.

$\rightarrow$ **Compute** $A_{k+1} := \frac{1}{2c_k}(A_k + c_k^2 BX)$ **with only** $2n^2 + 2nN_B$ **memory.**

Split $A_{k+1} := \frac{1}{2c_k}(A_k + c_k^2 BX)$ into

$$\begin{bmatrix} A^{(1)} \\ A^{(2)} \\ \vdots \\ A^{(N)} \end{bmatrix} := \frac{1}{2c_k} \left(\begin{bmatrix} A^{(1)} \\ A^{(2)} \\ \vdots \\ A^{(N)} \end{bmatrix} + c_k^2 \begin{bmatrix} B^{(1)} \\ B^{(2)} \\ \vdots \\ B^{(N)} \end{bmatrix} X \right),$$

where $A^{(\ell)}, B^{(\ell)} \in \mathbb{R}^{N_B \times n}$.

Memory-Efficient Implementation

Asynchronous Matrix-Matrix Product

Basic Workflow

- Upload $A^{(\ell)}$ block-by-block to the free n^2 memory location $\rightarrow A_{k+1}$ available for the next Gauss-Jordan Elimination,
- Upload $B^{(\ell)}$ to the two nN_B locations in an alternating way,
- Compute $A^{(\ell)} := \frac{1}{2c_k}(A^{(\ell)} + c_k^2 B^{(\ell)} X)$ from the alternating locations of $B^{(\ell)}$.

Memory-Efficient Implementation

Asynchronous Matrix-Matrix Product

Basic Workflow

- Upload $A^{(\ell)}$ block-by-block to the free n^2 memory location $\rightarrow A_{k+1}$ available for the next Gauss-Jordan Elimination,
- Upload $B^{(\ell)}$ to the two nN_B locations in an alternating way,
- Compute $A^{(\ell)} := \frac{1}{2c_k}(A^{(\ell)} + c_k^2 B^{(\ell)} X)$ from the alternating locations of $B^{(\ell)}$.

Algorithm 2 Asynchronous matrix-matrix product on the GPU

1: Asynchronous upload of $A^{(1)}$ and $B^{(1)}$.
2: **for** $i = 1, 1 + N_B, \ldots, n$ **do**
3:     Asynchronous upload of $A^{(i+1)}$ and $B^{(i+1)}$.
4:     Wait until the upload of $A^{(i)}$ and $B^{(i)}$ is done and compute $A^{(i)} := \frac{1}{2c_k}(A^{(i)} + c_k^2 B^{(i)} X)$.
5:     Asynchronous download of $A^{(i)}$ to the host.
6: **end for**

Memory-Efficient Implementation

Overheating and Throttling Detection

Due to worse thermal design of the chassis and/or previous computations the device might overheat, i.e. temperature on the die reaches 90 ℃.

Memory-Efficient Implementation

Overheating and Throttling Detection

Due to worse thermal design of the chassis and/or previous computations the device might overheat, i.e. temperature on the die reaches 90 ℃.

NVIDIA® Management Library (NVML)

- Part of the CUDA deployment kit,
- Read various performance metrics from the device, including temperature, clock speed, and power consumption,
- Fetch the frequency throttling state and its reasons.

Memory-Efficient Implementation

Overheating and Throttling Detection

Due to worse thermal design of the chassis and/or previous computations the device might overheat, i.e. temperature on the die reaches 90 °C.

NVIDIA® Management Library (NVML)

- Part of the CUDA deployment kit,
- Read various performance metrics from the device, including temperature, clock speed, and power consumption,
- Fetch the frequency throttling state and its reasons.

Monitoring Thread

- Checks the throttling state with a given frequency,
- Sets an indicator flag if power or temperature caused frequency throttling gets active,
- Throttling indicator must be reset manually.

Memory-Efficient Implementation

Overheating and Throttling Detection

Due to worse thermal design of the chassis and/or previous computations the device might overheat, i.e. temperature on the die reaches 90 ℃.

Nvidia Management Library (NVML)

- Part of the CUDA deployment kit,
- Read various performance metrics from the device, including temperature, clock speed, and power consumption,
- Fetch the frequency throttling state and its reasons.

Idea:

- Compute $X = A^{-1}B$ on the device,
- Check if the throttling indicator
  - **is set**: copy X to the host and compute $A_{k+1} := \frac{1}{2c_k}(A_k + c_k^2 BX)$ on the CPU.
  - **is not set**: compute $A_{k+1} := \frac{1}{2c_k}(A_k + c_k^2 BX)$ on the device using Algorithm 2.

Monitoring Thread

- Checks the throttling state with a given frequency,
- Sets an indicator flag if power or temperature caused frequency throttling gets active,
- Throttling indicator must be reset manually.

Memory-Efficient Implementation

Convergence Criteria

The convergence check

$$||A_{k+1} - A_k|| < \tau$$

costs additional n^2 and requires at least $2n^2$ memory transfers.

Memory-Efficient Implementation

Convergence Criteria

The convergence check

$$||A_{k+1} - A_k|| < \tau$$

costs additional n^2 and requires at least $2n^2$ memory transfers.

Necessary Convergence Criteria [Bierman '84]

Suppose $A_k \overset{k\to\infty}{\longrightarrow} \operatorname{sign}(A)$ and $\operatorname{sign}(A)^2 = I$ then we have

$$\det(\operatorname{sign}(A)) = \pm 1$$

.

Memory-Efficient Implementation

Convergence Criteria

The convergence check

$$||A_{k+1} - A_k|| < \tau$$

costs additional n^2 and requires at least $2n^2$ memory transfers.

Necessary Convergence Criteria [Bierman '84]

Suppose $A_k \overset{k\to\infty}{\longrightarrow} \operatorname{sign}(A)$ and $\operatorname{sign}(A)^2 = I$ then we have

$$\det(\operatorname{sign}(A)) = \pm 1$$

and

$$\det(A_k) \overset{k\to\infty}{\longrightarrow} \pm 1$$

.

Memory-Efficient Implementation

Convergence Criteria

The convergence check

$$||A_{k+1} - A_k|| < \tau$$

costs additional n^2 and requires at least $2n^2$ memory transfers.

Necessary Convergence Criteria [Bierman '84]

Suppose $A_k \stackrel{k\to\infty}{\longrightarrow} \operatorname{sign}(A)$ and $\operatorname{sign}(A)^2 = I$ then we have

$$\det(\operatorname{sign}(A)) = \pm 1$$

and

$$\det(A_k) \stackrel{k\to\infty}{\longrightarrow} \pm 1$$

which yields

$$c_k = |\det(A_k)|^{\frac{1}{n}} \stackrel{k\to\infty}{\longrightarrow} 1.$$

Memory-Efficient Implementation

Convergence Criteria

The convergence check

$$||A_{k+1} - A_k|| < \tau$$

costs additional n^2 and requires at least $2n^2$ memory transfers.

Necessary Convergence Criteria [Bierman '84]

Suppose $A_k \xrightarrow{k\to\infty} \operatorname{sign}(A)$ and $\operatorname{sign}(A)^2 = I$ then we have

$$\det(\operatorname{sign}(A)) = \pm 1$$

Scaling factor of the standard sign function iteration.

and

$$\det(A_k) \xrightarrow{k\to\infty} \pm 1$$

which yields

$$c_k = \boxed{|\det(A_k)|^{\frac{1}{n}}} \xrightarrow{k\to\infty} 1.$$

Memory-Efficient Implementation

Convergence Criteria

The convergence check

$$||A_{k+1} - A_k|| < \tau$$

costs additional n^2 and requires at least $2n^2$ memory transfers.

Generalized Necessary Criteria

From $A_k \overset{k\rightarrow\infty}{\longrightarrow} \operatorname{sign}(A, B)$ and $\operatorname{sign}(A, B)^2 = B^2 \operatorname{sign}\left(B^{-1}A\right) = B^2$ we get:

$$A_k^2 \overset{k\rightarrow\infty}{\longrightarrow} B^2,$$

Memory-Efficient Implementation

Convergence Criteria

The convergence check

$$||A_{k+1} - A_k|| < \tau$$

costs additional n^2 and requires at least $2n^2$ memory transfers.

Generalized Necessary Criteria

From $A_k \stackrel{k\to\infty}{\longrightarrow} \operatorname{sign}(A, B)$ and $\operatorname{sign}(A, B)^2 = B^2 \operatorname{sign}\left(B^{-1}A\right) = B^2$ we get:

$$A_k^2 \stackrel{k\to\infty}{\longrightarrow} B^2,$$

$$\det(A_k) \stackrel{k\to\infty}{\longrightarrow} \det(B),$$

$$\frac{\det(A_k)}{\det(B)} \stackrel{k\to\infty}{\longrightarrow} 1,$$

Memory-Efficient Implementation

Convergence Criteria

The convergence check

$$||A_{k+1} - A_k|| < \tau$$

costs additional n^2 and requires at least $2n^2$ memory transfers.

Generalized Necessary Criteria

From $A_k \overset{k\to\infty}{\longrightarrow} \operatorname{sign}(A, B)$ and $\operatorname{sign}(A, B)^2 = B^2 \operatorname{sign}\left(B^{-1}A\right) = B^2$ we get:

$$A_k^2 \overset{k\to\infty}{\longrightarrow} B^2,$$

$$\det(A_k) \overset{k\to\infty}{\longrightarrow} \det(B),$$

$$\frac{\det(A_k)}{\det(B)} \overset{k\to\infty}{\longrightarrow} 1,$$

and especially

$$c_k = \left(\frac{|\det(A_k)|}{|\det(B)|}\right)^{\frac{1}{n}} \overset{k\to\infty}{\longrightarrow} 1.$$

Memory-Efficient Implementation

Convergence Criteria

The convergence check

$$||A_{k+1} - A_k|| < \tau$$

costs additional n^2 and requires at least $2n^2$ memory transfers.

Generalized Necessary Criteria

From $A_k \stackrel{k\to\infty}{\longrightarrow} \operatorname{sign}(A,B)$ and $\operatorname{sign}(A,B)^2 = B^2 \operatorname{sign}\left(B^{-1}A\right) = B^2$ we get:

$$A_k^2 \stackrel{k\to\infty}{\longrightarrow} B^2,$$

$$\det(A_k) \stackrel{k\to\infty}{\longrightarrow} \det(B),$$

Scaling factor of the generalized sign function iteration.

$$\frac{\det(A_k)}{\det(B)} \stackrel{k\to\infty}{\longrightarrow} 1,$$

and especially

$$c_k = \boxed{\left(\frac{|\det(A_k)|}{|\det(B)|}\right)^{\frac{1}{n}}} \stackrel{k\to\infty}{\longrightarrow} 1.$$

Memory-Efficient Implementation

Convergence Criteria

The convergence check

$$||A_{k+1} - A_k|| < \tau$$

costs additional n^2 and requires at least $2n^2$ memory transfers.

Generalized Necessary Criteria

From $A_k \xrightarrow{k\to\infty} B^2 \operatorname{sign}(B^{-1}A) = B^2$ we get:

$$A_k^2 \xrightarrow{k\to\infty} B^2,$$

$$\det(A_k) \xrightarrow{k\to\infty} \det(B),$$

$$\frac{\det(A_k)}{\det(B)} \xrightarrow{k\to\infty} 1,$$

and especially

$$c_k = \left(\frac{|\det(A_k)|}{|\det(B)|}\right)^{\frac{1}{n}} \xrightarrow{k\to\infty} 1.$$

New Convergence Criterion:

Stop the iteration if

$$|c_k - 1| < \delta \quad \text{for} \quad k > k_0.$$

$\rightarrow$ **No additional work or memory necessary.**

Results

Test Setup

Optimal Cooled System

- 19", 1 HU Supermicro chassis
- 2× Intel® Xeon® E5-2640 v3
- 2× NVIDIA® Telsa K20m, passive cooling

Worse Cooled System

- 19", 2 HU Dell R720 chassis
- 2× Intel® Xeon® E5-2690
- 2× NVIDIA® Telsa K20m, passive cooling

Software

- CentOS 7 – 64bit
- Intel® Compiler 15
- Intel® MKL 11
- NVIDIA® CUDA 6.5
- MAGMA 1.6

Software

- Ubuntu 14.04 – 64bit
- Intel® Compiler 14
- Intel® MKL 11
- NVIDIA® CUDA 6.5
- MAGMA 1.6

Results

Test Setup

- $A, B \in \mathbb{R}^{n \times n}$ chosen as random matrices,

Results

Test Setup

- $A, B \in \mathbb{R}^{n \times n}$ chosen as random matrices,
- Sign function iteration is fixed to 25 iterations,

Results

Test Setup

- A, $B \in \mathbb{R}^{n \times n}$ chosen as random matrices,
- Sign function iteration is fixed to 25 iterations,
- Power computation measured with LMG450 @ 20Hz,

Results

Test Setup

- $A, B \in \mathbb{R}^{n \times n}$ chosen as random matrices,
- Sign function iteration is fixed to 25 iterations,
- Power computation measured with LMG450 @ 20Hz,
- GPUs running at 758MHz,

Results

Test Setup

- $A, B \in \mathbb{R}^{n \times n}$ chosen as random matrices,
- Sign function iteration is fixed to 25 iterations,
- Power computation measured with LMG450 @ 20Hz,
- GPUs running at 758MHz,
- Preheat device to 60 ℃ for the temperature tests on the worse system to simulate previous computational work on the device.

Results

Computational Performance – On the optimal System

Problem size	$4n^2$ memory		$3n^2$ memory		$2n^2 + \mathcal{O}(n)$ memory	
	Runtime	Floprate	Runtime	Floprate	Runtime	Floprate
2 000	2.87	341.7	3.10	311.8	**2.26**	446.6
4 000	12.33	620.0	14.63	519.8	**11.34**	711.8
6 000	**33.26**	769.6	38.32	666.1	**33.27**	818.6
8 000	**70.94**	853.1	80.05	754.3	73.49	878.4
10 000	**133.51**	883.8	147.57	798.1	136.74	921.9
12 000	**224.32**	908.1	244.86	830.6	228.31	954.0
14 000	-	-	370.11	873.5	**353.50**	978.4
16 000	-	-	-	-	**521.39**	990.2
17 000	-	-	-	-	**627.96**	985.8[1]

Table: Runtime (in s) and Floprate (GFlops/s) on the optimal system.

[1] Optimal blocksize restricted by the available memory of the device.

Results

Computational Performance – On the optimal System

Problem size	$4n^2$ memory		$3n^2$ memory		$2n^2 + \mathcal{O}(n)$ memory	
	Runtime	Floprate	Runtime	Floprate	Runtime	Floprate
2 000	2.87	341.7	3.10	311.8	**2.26**	446.6
4 000	12.33	620.0	14.63	519.8	**11.34**	711.8
6 000	**33.26**	769.6	38.32	666.1	**33.27**	818.6
8 000	**70.94**	853.1	80.05	754.3	73.49	878.4
10 000	**133.51**	883.8	147.57	798.1	136.74	921.9
12 000	**224.32**	908.1	244.86	830.6	228.31	954.0
14 000	-	-	370.11	873.5	**353.50**	978.4
16 000	-	-	-	-	**521.39**	990.2
17 000	-	-	-	-	**627.96**	985.8[1]

Table: Runtime (in s) and Floprate (GFlops/s) on the optimal system.

Maximum GPU temperature: 55 ℃.

[1] Optimal blocksize restricted by the available memory of the device.

Results

Energy Efficiency

Problem size	$4n^2$ memory		$3n^2$ memory		$2n^2 + \mathcal{O}(n)$ memory	
	avg. Pwr.	Eff.	avg. Pwr.	Eff.	avg. Pwr.	Eff.
2 000	351.83	0.97	353.66	0.88	376.23	**1.19**
4 000	382.83	**1.62**	363.56	1.43	420.40	1.69
6 000	377.68	**2.03**	355.51	1.87	420.56	1.95
8 000	386.85	**2.21**	368.96	2.04	423.80	2.07
10 000	388.21	**2.28**	375.19	2.13	430.74	2.14
12 000	393.56	**2.31**	377.86	2.20	429.37	2.22
14 000	-	-	384.32	2.27	421.91	**2.31**
16 000	-	-	-	-	419.43	**2.36**
17 000	-	-	-	-	419.32	**2.35**

Table: Average power consumption (in W) and computational efficiency (in GFlops $\cdot (s \cdot W)^{-1}$).

Results

Overheating Problem - Worse Hardware

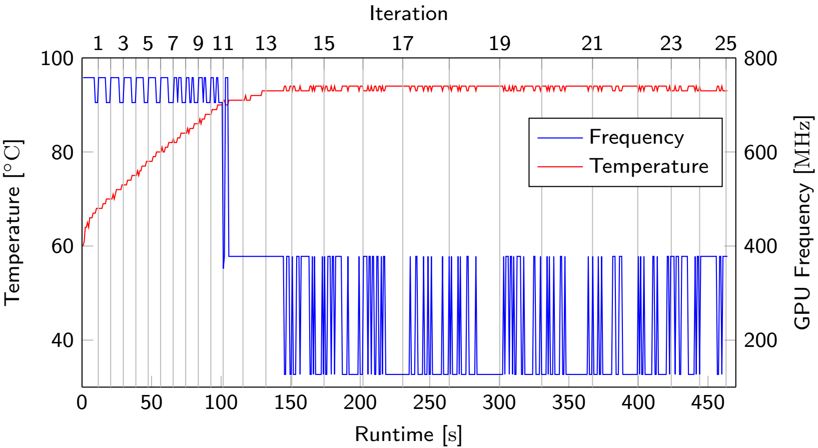
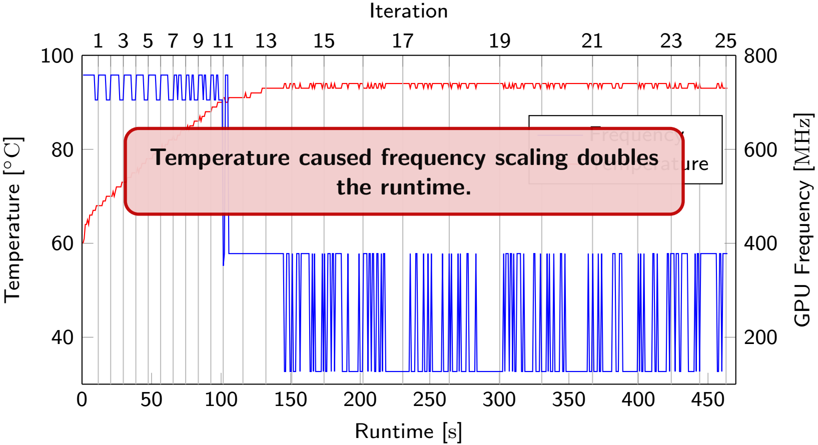


Figure: Device temperature and GPU clock frequency for the $4n^2$ algorithm with $n = 12\,000$.

Results

Overheating Problem - Worse Hardware

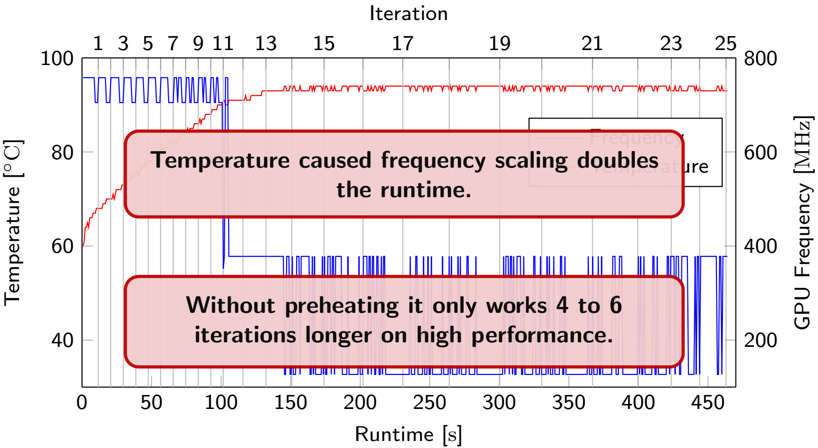


Temperature caused frequency scaling doubles the runtime.

Figure: Device temperature and GPU clock frequency for the $4n^2$ algorithm with $n = 12\,000$.

Results

Overheating Problem - Worse Hardware



Temperature caused frequency scaling doubles the runtime.

Without preheating it only works 4 to 6 iterations longer on high performance.

Figure: Device temperature and GPU clock frequency for the $4n^2$ algorithm with $n = 12\,000$.

Results

Overheating Problem - Worse Hardware

Problem size	$4n^2$ memory		$3n^2$ memory		$2n^2 + \mathcal{O}(n)$ memory	
	Runtime	Floprate	Runtime	Floprate	Runtime	Floprate
$n = 10\,000$	149.02	792	164.28	717	168.27	759
$n = 12\,000$	468.66	433	341.04	565	273.12	808
$n = 14\,000$	-	-	674.61	476	437.34	797
$n = 16\,000$	-	-	-	-	717.66	725
$n = 17\,000$	-	-	-	-	845.40	737

Table: Runtime (in s) and Floprate (GFlops/s) on the worse system.

Results

Overheating Problem - Worse Hardware

Problem size	$4n^2$ memory		$3n^2$ memory		$2n^2 + \mathcal{O}(n)$ memory	
	Runtime	Floprate	Runtime	Floprate	Runtime	Floprate
$n = 10\,000$	149.02	792	164.28	717	168.27	759
$n = 12\,000$	468.66	433	341.04	565	273.12	808
$n = 14\,000$	-	-	674.61	476	437.34	797
$n = 16\,000$	-	-	-	-	717.66	725
$n = 17\,000$	-	-	-	-	845.40	737

Table: Runtime (in s) and Floprate (GFlops/s) on the worse system.

- Both straight forward implementations are affected dramatically by the frequency scaling. $\rightarrow$ Lost nearly 50% of their performance.
- Moving single matrix-matrix products to the host allows the GPU to cool down and recover its full performance.

Conclusions and Outlook

Conclusions

- Bad thermal hardware designs require different algorithms.
- The generalized sign function iteration can be implemented on the GPU with the memory restrictions of the `GEMM` operation.
  $\rightarrow$ allows to increase the maximum problem size by $\sqrt{2}$.
- For small problems ($n \leq 6\,000$) the Gauss-Jordan-Elimination approach is faster than the MAGMA-LU based one.

Conclusions and Outlook

Conclusions

- Bad thermal hardware designs require different algorithms.
- The generalized sign function iteration can be implemented on the GPU with the memory restrictions of the `GEMM` operation.
  $\rightarrow$ allows to increase the maximum problem size by $\sqrt{2}$.
- For small problems ($n \leq 6\,000$) the Gauss-Jordan-Elimination approach is faster than the MAGMA-LU based one.

Outlook

- Combine the asynchronous matrix-matrix products with the MAGMA solvers.
- Improve the energy efficiency of the algorithms.
- Develop a multi-GPU aware generalized sign function iteration on top of the Gauss-Jordan Elimination and the asynchronous matrix-matrix product.
- Extend the GPU implementation to solve generalized Lyapunov and Riccati equations.

Conclusions and Outlook

Conclusions

- Bad thermal hardware designs require different algorithms.
- The generalized sign function iteration can be implemented on the GPU with the memory restrictions of the `GEMM` operation.
  $\rightarrow$ allows to increase the maximum problem size by $\sqrt{2}$.
- For small problems ($n \leq 6\,000$) the Gauss-Jordan-Elimination approach is faster than the MAGMA-LU based one.

Thank you for you attention!

Outlook

- Combine the asynchronous matrix-matrix products with the MAGMA solvers.
- Improve the energy efficiency of the algorithms.
- Develop a multi-GPU aware generalized sign function iteration on top of the Gauss-Jordan Elimination and the asynchronous matrix-matrix product.
- Extend the GPU implementation to solve generalized Lyapunov and Riccati equations.