Applying OOC Techniques in the Reduction to Condensed Form for Very Large Symmetric Eigenproblems on GPUs

Davor Davidović[1] Enrique S. Quintana-Ortí[2]

[1]Centre for Informatics and Computating
Rudjer Bošković Institute

[2]Depto. de Ingeniería y Ciencia de Computadores
Universitat Jaume I

PDP, 2012

Motivation

Why large-scale eigenproblems?

Large-scale eigenproblem arises in different fields:

- molecular dynamics,
- computational quantum chemistry,
- finite element modeling,
- multivariate statistics.

Require a hugh amount of the memory space and computational power

Motivation

Why large-scale eigenproblems?

Large-scale eigenproblem arises in different fields:

- molecular dynamics,
- computational quantum chemistry,
- finite element modeling,
- multivariate statistics.

Require a hugh amount of the memory space and computational power

Motivation

Problems with the Large-scale eigensolvers on the GPU

- GPU implementations exist but can not handle problems that oversize the GPU memory!
- Small GPU memory: increase the number of I/O memory transfers!
- PCI-e bottleneck:
  - High latency
  - Slow bandwidth compared to GPU theoretical peak performance
  - To override the problem → reduce the number of transfers and increase the memory chunks

Solution in applying out-of-core (OOC) techniques

Motivation

Problems with the Large-scale eigensolvers on the GPU

- GPU implementations exist but can not handle problems that oversize the GPU memory!
- Small GPU memory: increase the number of I/O memory transfers!
- PCI-e bottleneck:
  - High latency
  - Slow bandwidth compared to GPU theoretical peak performance
  - To override the problem → reduce the number of transfers and increase the memory chunks

Solution in applying out-of-core (OOC) techniques

Motivation

Problems with the Large-scale eigensolvers on the GPU

- GPU implementations exist but can not handle problems that oversize the GPU memory!
- Small GPU memory: increase the number of I/O memory transfers!
- PCI-e bottleneck:
  - High latency
  - Slow bandwidth compared to GPU theoretical peak performance
  - To override the problem $\rightarrow$ reduce the number of transfers and increase the memory chunks

Solution in applying out-of-core (OOC) techniques

Motivation

Problems with the Large-scale eigensolvers on the GPU

- GPU implementations exist but can not handle problems that oversize the GPU memory!
- Small GPU memory: increase the number of I/O memory transfers!
- PCI-e bottleneck:
  - High latency
  - Slow bandwidth compared to GPU theoretical peak performance
  - To override the problem $\rightarrow$ reduce the number of transfers and increase the memory chunks

Solution in applying out-of-core (OOC) techniques

Motivation

Problems with the Large-scale eigensolvers on the GPU

- GPU implementations exist but can not handle problems that oversize the GPU memory!
- Small GPU memory: increase the number of I/O memory transfers!
- PCI-e bottleneck:
  - High latency
  - Slow bandwidth compared to GPU theoretical peak performance
  - To override the problem $\rightarrow$ reduce the number of transfers and increase the memory chunks

Solution in applying out-of-core (OOC) techniques

Motivation

Problems with the Large-scale eigensolvers on the GPU

- GPU implementations exist but can not handle problems that oversize the GPU memory!
- Small GPU memory: increase the number of I/O memory transfers!
- PCI-e bottleneck:
  - High latency
  - Slow bandwidth compared to GPU theoretical peak performance
  - To override the problem $\rightarrow$ reduce the number of transfers and increase the memory chunks

Solution in applying out-of-core (OOC) techniques

Motivation

Problems with the Large-scale eigensolvers on the GPU

- GPU implementations exist but can not handle problems that oversize the GPU memory!
- Small GPU memory: increase the number of I/O memory transfers!
- PCI-e bottleneck:
  - High latency
  - Slow bandwidth compared to GPU theoretical peak performance
  - To override the problem $\rightarrow$ reduce the number of transfers and increase the memory chunks

Solution in applying out-of-core (OOC) techniques

Outline

Outline

Eigenvalue problem

Problem statement

- The eigenproblem is defined as:

$$AX = \Lambda X\,,$$

where A is symmetric and Λ is diagonal with the sought-after eigenvalues and X contains the associated eigenvectors.

Techniques for solving eigenvalue problems

Standard algorithm for finding eigenvalues

1. Reduce starting matrix to tridiagonal form
2. Apply fast algorithm (i.e. MR^3) to find eigenvalues of the tridiagonal matrix $\rightarrow$ less expensive

One-stage reduction to tridiagonal form

- Reduction of full dense matrix to tridiagonal form using orthogonal transforms

$$Q^T A Q \rightarrow T,$$

  where T is tridiagonal, and Q is accumulation of orthogonal transforms
- Most of execution time spent in level 2 BLAS operations $\rightarrow$ 50% of total flops!

Techniques for solving eigenvalue problems

Standard algorithm for finding eigenvalues

1. Reduce starting matrix to tridiagonal form
2. Apply fast algorithm (i.e. MR^3) to find eigenvalues of the tridiagonal matrix $\rightarrow$ less expensive

One-stage reduction to tridiagonal form

- Reduction of full dense matrix to tridiagonal form using orthogonal transforms

$$Q^T A Q \rightarrow T,$$

  where T is tridiagonal, and Q is accumulation of orthogonal transforms
- Most of execution time spent in level 2 BLAS operations $\rightarrow$ 50% of total flops!

Techniques for solving eigenvalue problems

Two-stage reduction to tridiagonal form

1. First reduce full dense matrix to banded form

$$Q_1^T A Q_1 \rightarrow B_1$$

Note: All performed in level 3 BLAS operations (blocked operations - higher efficiency)

2. Reduce the banded matrix to tridiagonal form

$$Q_2^T B_1 Q_2 \rightarrow T$$

Techniques for solving eigenvalue problems

Two-stage reduction to tridiagonal form

1. First reduce full dense matrix to banded form

$$Q_1^T A Q_1 \rightarrow B_1$$

Note: All performed in level 3 BLAS operations (blocked operations - higher efficiency)

2. Reduce the banded matrix to tridiagonal form

$$Q_2^T B_1 Q_2 \rightarrow T$$

What can we do?

Goals

- Re-implement existing two-stage algorithms to apply OOC techniques to solve large scale eigenvalue problems
  - Disk = main memory (CPU)
  - Main memory = global memory (GPU)
- Optimize memory transfers and maximize amount of computation on GPU
- Make an algorithm that can operate on any problem size (scalable in problem dimension)

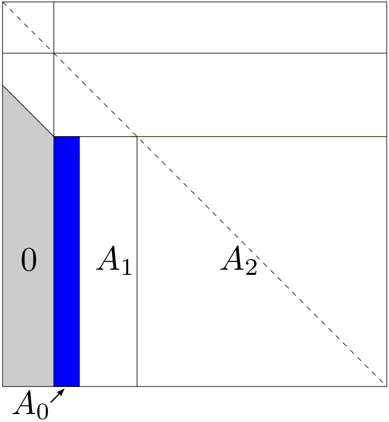
Outline

Successive Band Reduction

SBR toolbox

- Software package for reduction of dense symmetric matrices to banded or tridiagonal form
- Routines for multi-stage reduction to tridiagonal form
    - xSYRDB: Full $\rightarrow$ band form
    - xSBRDB: Band $\rightarrow$ narrower band form
    - xSBRDT: Band $\rightarrow$ tridiagonal form

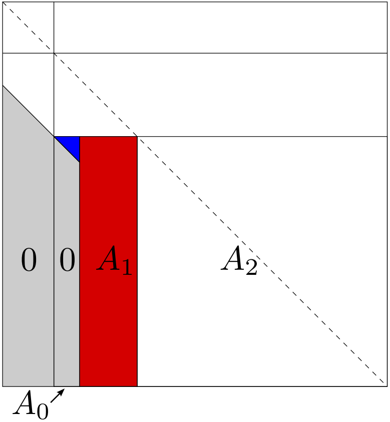
SBR reduction from full to band form



One iteration of the xSYRDB routine

1. Factorize panel $A_0 \rightarrow Q_0 R_0$ and construct W, Y factors s. t. $Q_0 = I + WY^T$
2. Apply orthogonal matrix Q_0 to $A_1 := Q_0^T A_1$
3. Apply Q_0 to $A_2 := Q_0^T A_2 Q_0$

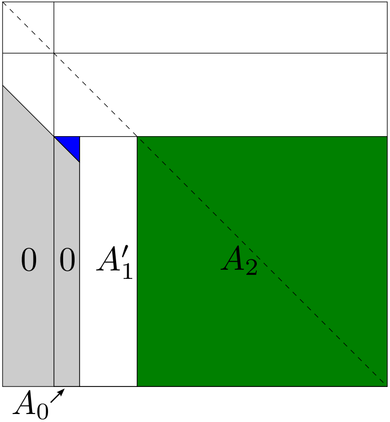
SBR reduction from full to band form



One iteration of the xSYRDB routine

1. Factorize panel $A_0 \to Q_0 R_0$ and construct W, Y factors s. t. $Q_0 = I + WY^T$
2. Apply orthogonal matrix Q_0 to $A_1 := Q_0^T A_1$
3. Apply Q_0 to $A_2 := Q_0^T A_2 Q_0$

SBR reduction from full to band form



One iteration of the xSYRDB routine

1. Factorize panel $A_0 \to Q_0 R_0$ and construct W, Y factors s. t. $Q_0 = I + WY^T$
2. Apply orthogonal matrix Q_0 to $A_1 := Q_0^T A_1$
3. Apply Q_0 to $A_2 := Q_0^T A_2 Q_0$

SBR Toolbox

Flops count

- The total cost of the reduction to band form: $2n^3/3$ flops
- The bulk of the computation is cast in terms of BLAS3 operations (better than one-stage approach)
- The most time consuming step is applying orthogonal matrix Q_0 to A_2

$$A_2 := Q_0^T A_2 Q_0 = A_2 + YW^T A_2 + A_2 WY^T + YW^T A_2 WY^T$$

- Good candidate to be executed on the GPU

SBR Toolbox

Flops count

- The total cost of the reduction to band form: $2n^3/3$ flops
- The bulk of the computation is cast in terms of BLAS3 operations (better than one-stage approach)
- The most time consuming step is applying orthogonal matrix Q_0 to A_2

$$A_2 := Q_0^T A_2 Q_0 = A_2 + YW^T A_2 + A_2 WY^T + YW^T A_2 WY^T$$

- Good candidate to be executed on the GPU

SBR Toolbox

Flops count

- The total cost of the reduction to band form: $2n^3/3$ flops
- The bulk of the computation is cast in terms of BLAS3 operations (better than one-stage approach)
- The most time consuming step is applying orthogonal matrix Q_0 to A_2

$$A_2 := Q_0^T A_2 Q_0 = A_2 + YW^T A_2 + A_2 WY^T + YW^T A_2 WY^T$$

- Good candidate to be executed on the GPU

SBR Toolbox

Flops count

- The total cost of the reduction to band form: $2n^3/3$ flops
- The bulk of the computation is cast in terms of BLAS3 operations (better than one-stage approach)
- The most time consuming step is applying orthogonal matrix Q_0 to A_2

$$A_2 := Q_0^T A_2 Q_0 = A_2 + YW^T A_2 + A_2 WY^T + YW^T A_2 WY^T$$

- Good candidate to be executed on the GPU

Outline

OOC reduction to band form on hybrid GPU platforms

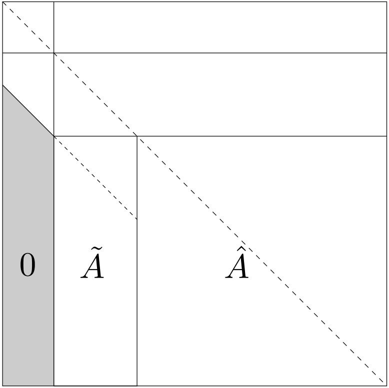
To compensate memory transfer with the computation, blocks (band size) have to be large enough

One step of the OOC reduction to band form

1. Set blocks $\tilde{A} := [A_0 A_1]$ and $\hat{A} := A_2$
2. QR factorization of $\tilde{A} = Q_i R_i$ and construction of W_i and Y_i
3. Two-sided update of $\hat{A} := Q_i^T \hat{A} Q_i$

OOC reduction to band form on hybrid GPU platforms

To compensate memory transfer with the computation, blocks (band size) have to be large enough

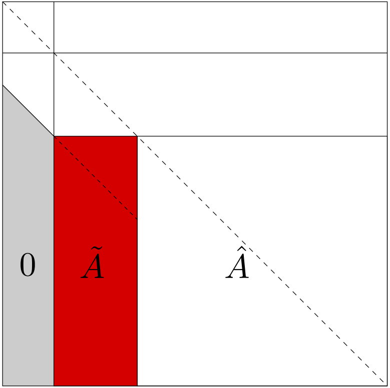


One step of the OOC reduction to band form

1. Set blocks $\tilde{A} := [A_0 A_1]$ and $\hat{A} := A_2$
2. QR factorization of $\tilde{A} = Q_i R_i$ and construction of W_i and Y_i
3. Two-sided update of $\hat{A} := Q_i^T \hat{A} Q_i$

OOC reduction to band form on hybrid GPU platforms

To compensate memory transfer with the computation, blocks (band size) have to be large enough

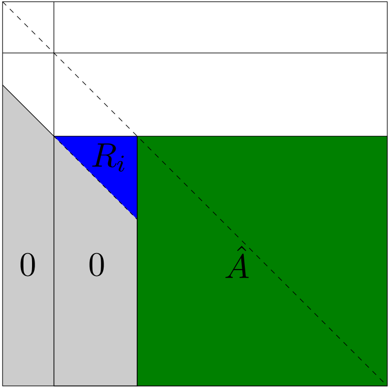


One step of the OOC reduction to band form

1. Set blocks $\tilde{A} := [A_0 A_1]$ and $\hat{A} := A_2$
2. QR factorization of $\tilde{A} = Q_i R_i$ and construction of W_i and Y_i
3. Two-sided update of $\hat{A} := Q_i^T \hat{A} Q_i$

OOC reduction to band form on hybrid GPU platforms

To compensate memory transfer with the computation, blocks (band size) have to be large enough



One step of the OOC reduction to band form

1. Set blocks $\tilde{A} := [A_0 A_1]$ and $\hat{A} := A_2$
2. QR factorization of $\tilde{A} = Q_i R_i$ and construction of W_i and Y_i
3. Two-sided update of $\hat{A} := Q_i^T \hat{A} Q_i$

Hybrid (in-core) QR decomposition

- QR factorization of $\tilde{A}$ rich in small-sized BLAS2 operations
- Bad performance when $\tilde{A}$ is big, even on multi-core systems
- Solution: Implement panel QR factorization
- $\tilde{A}$ is divided into panels $\rightarrow$ do panel factorization on the CPU, and update on the GPU

Hybrid (in-core) QR decomposition

- QR factorization of $\tilde{A}$ rich in small-sized BLAS2 operations
- Bad performance when $\tilde{A}$ is big, even on multi-core systems
- Solution: Implement panel QR factorization
- $\tilde{A}$ is divided into panels $\rightarrow$ do panel factorization on the CPU, and update on the GPU

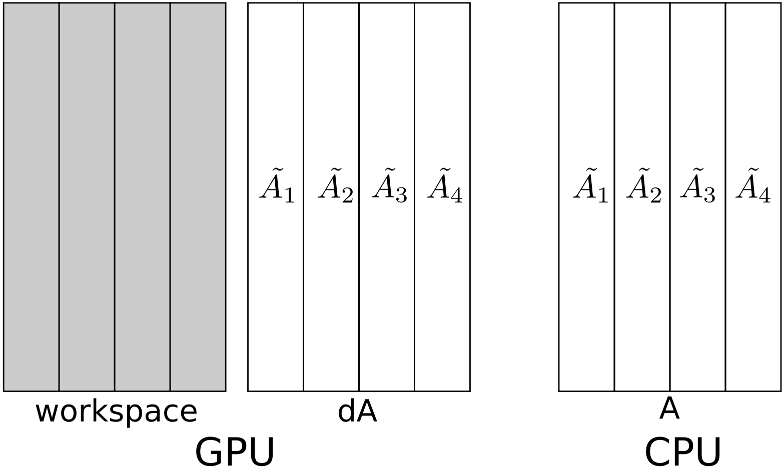
Hybrid (in-core) QR decomposition

- QR factorization of $\tilde{A}$ rich in small-sized BLAS2 operations
- Bad performance when $\tilde{A}$ is big, even on multi-core systems
- Solution: Implement panel QR factorization
- $\tilde{A}$ is divided into panels $\rightarrow$ do panel factorization on the CPU, and update on the GPU

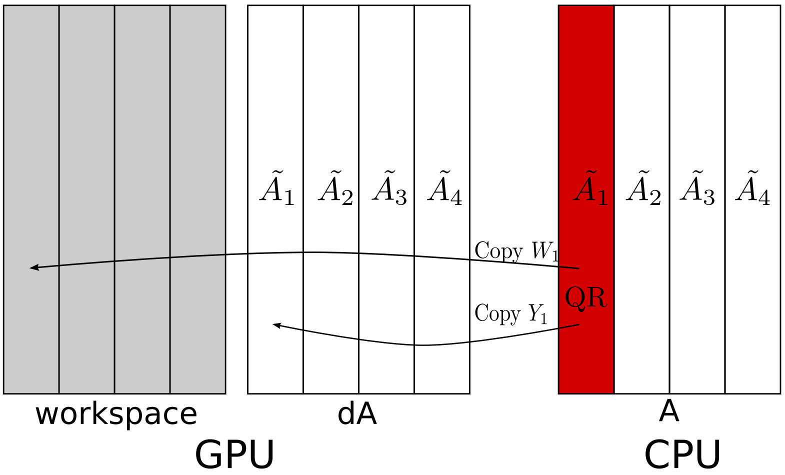
Hybrid (in-core) QR decomposition

- QR factorization of $\tilde{A}$ rich in small-sized BLAS2 operations
- Bad performance when $\tilde{A}$ is big, even on multi-core systems
- Solution: Implement panel QR factorization
- $\tilde{A}$ is divided into panels $\rightarrow$ do panel factorization on the CPU, and update on the GPU

Hybrid QR

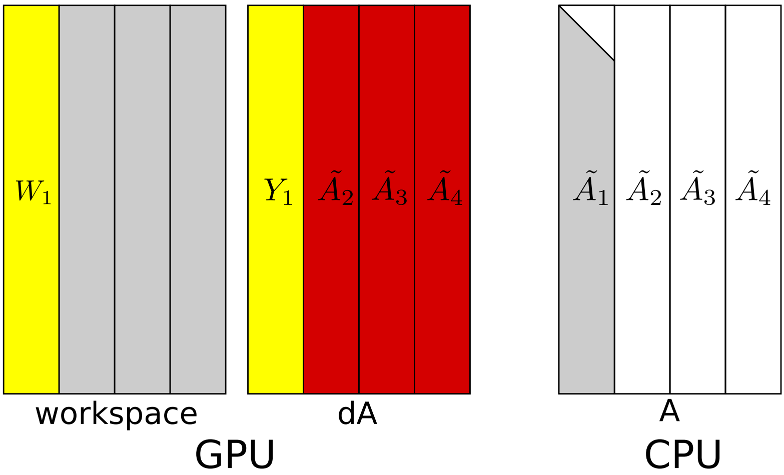

$\tilde{A}_1$ $\tilde{A}_2$ $\tilde{A}_3$ $\tilde{A}_4$
$\tilde{A}_1$ $\tilde{A}_2$ $\tilde{A}_3$ $\tilde{A}_4$
workspace
dA
A
GPU
CPU

Hybrid QR

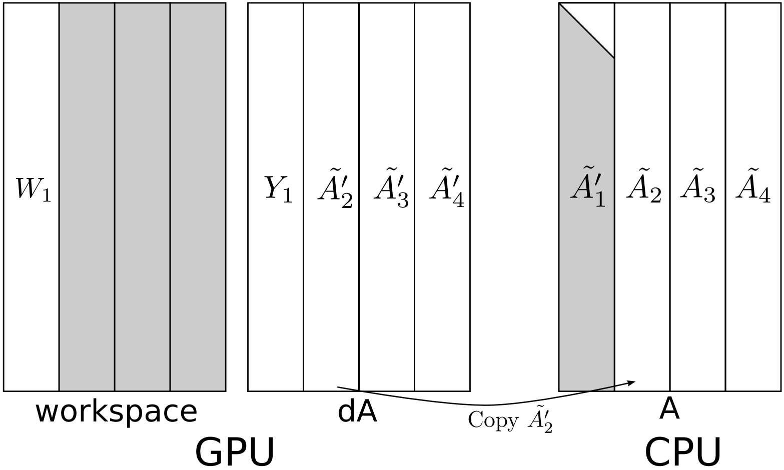


$\tilde{A}_1$ $\tilde{A}_2$ $\tilde{A}_3$ $\tilde{A}_4$
$\tilde{A}_1$ $\tilde{A}_2$ $\tilde{A}_3$ $\tilde{A}_4$
Copy W_1
QR
Copy Y_1
workspace
dA
A
GPU
CPU

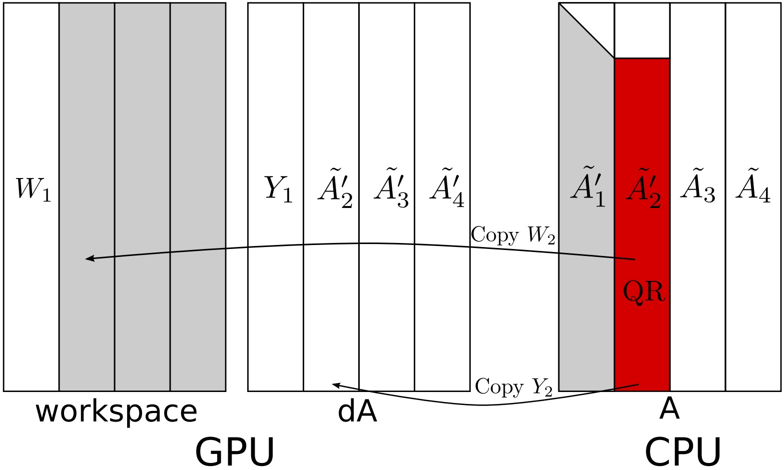
Hybrid QR


W_1
Y_1
$\tilde{A}_2$
$\tilde{A}_3$
$\tilde{A}_4$
$\tilde{A}_1$
$\tilde{A}_2$
$\tilde{A}_3$
$\tilde{A}_4$
workspace
dA
A
GPU
CPU

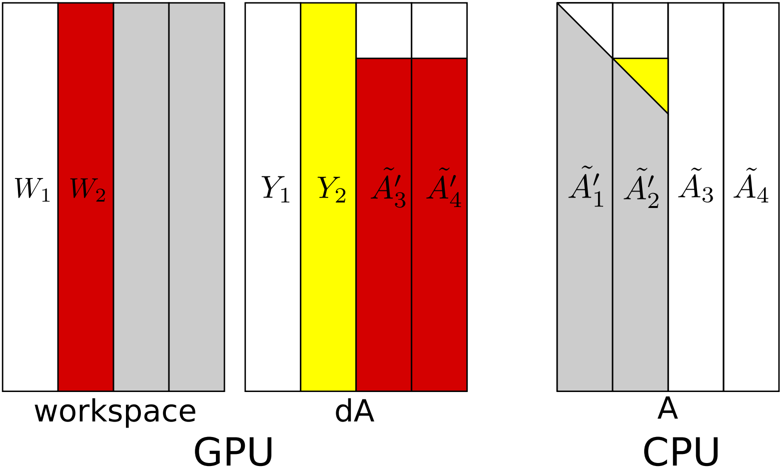
Hybrid QR


W_1
Y_1
$\tilde{A}_2'$
$\tilde{A}_3'$
$\tilde{A}_4'$
$\tilde{A}_1'$
$\tilde{A}_2$
$\tilde{A}_3$
$\tilde{A}_4$
workspace
dA
Copy $\tilde{A}_2'$
A
GPU
CPU

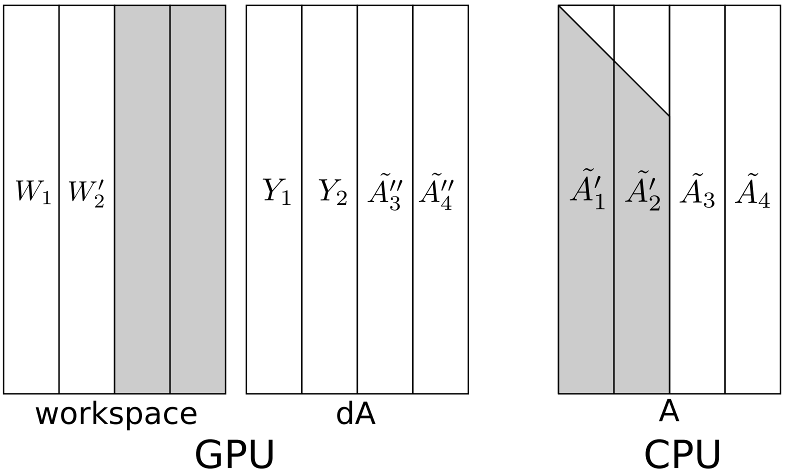
Hybrid QR


W_1
Y_1
$\tilde{A}'_2$
$\tilde{A}'_3$
$\tilde{A}'_4$
$\tilde{A}'_1$
$\tilde{A}'_2$
$\tilde{A}_3$
$\tilde{A}_4$
Copy W_2
QR
Copy Y_2
workspace
dA
A
GPU
CPU

Hybrid QR


W_1
W_2
Y_1
Y_2
$\tilde{A}'_3$
$\tilde{A}'_4$
$\tilde{A}'_1$
$\tilde{A}'_2$
$\tilde{A}_3$
$\tilde{A}_4$
workspace
dA
A
GPU
CPU

Hybrid QR


W_1
W_2'
workspace
Y_1
Y_2
$\tilde{A}_3''$
$\tilde{A}_4''$
dA
GPU
$\tilde{A}_1'$
$\tilde{A}_2'$
$\tilde{A}_3$
$\tilde{A}_4$
A
CPU

Hybrid two-sided update

Two-sided update

Applying Q to $\hat{A}$ from both sides:

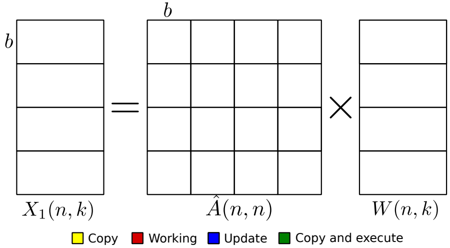
$$\begin{aligned}\hat{A} &:= Q^T\hat{A}Q = (I + WY^T)^T\hat{A}(I + WY^T) \\ &= \hat{A} + YW^T\hat{A} + \hat{A}WY^T + YW^T\hat{A}WY^T.\end{aligned} \tag{1}$$

How to efficiently compute the update

The two-sided update can be divided into 4 steps:

1. (SYMM) $X_1 := \hat{A}W$,
2. (GEMM) $X_2 := \frac{1}{2}X_1^T W$,
3. (GEMM) $X_3 := X_1 + YX_2$,
4. (SYR2K) $\hat{A} := \hat{A} + X_3Y^T + YX_3^T$.

First step $X_1 := \hat{A}W$



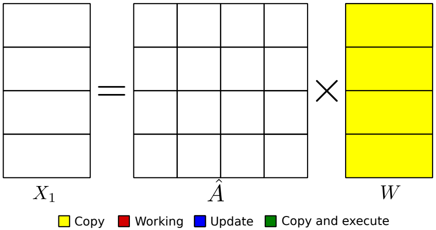
Computing X_1

1. Choose b so that blocks of size $k \times b$, b^2 and $n \times k$ fit into the GPU memory
2. Divide X_1, $\hat{A}$ and W into blocks, copy W on the GPU
3. Copy $\hat{A}_{ij}$ to GPU and update X_{1i}

$$X_{1i} = X_{1i} + \hat{A}_{ij} * W_j.$$

4. Return X_{1i} to the CPU

First step $X_1 := \hat{A}W$



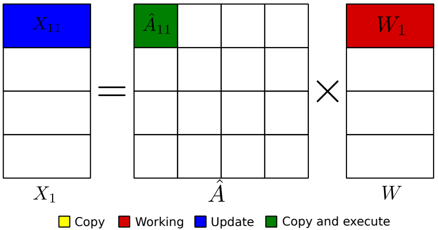
Computing X_1

1. Choose b so that blocks of size $k \times b$, b^2 and $n \times k$ fit into the GPU memory
2. Divide X_1, $\hat{A}$ and W into blocks, copy W on the GPU
3. Copy $\hat{A}_{ij}$ to GPU and update X_{1i}

$$X_{1i} = X_{1i} + \hat{A}_{ij} * W_j.$$

4. Return X_{1i} to the CPU

First step $X_1 := \hat{A}W$



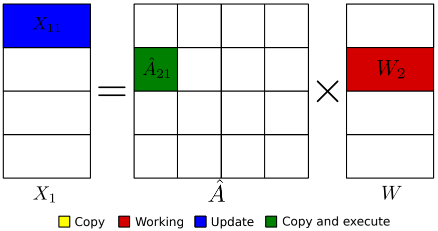
Computing X_1

1. Choose b so that blocks of size $k \times b$, b^2 and $n \times k$ fit into the GPU memory
2. Divide X_1, $\hat{A}$ and W into blocks, copy W on the the GPU
3. Copy $\hat{A}_{ij}$ to GPU and update X_{1i}

$$X_{1i} = X_{1i} + \hat{A}_{ij} * W_j.$$

4. Return X_{1i} to the CPU

First step $X_1 := \hat{A}W$



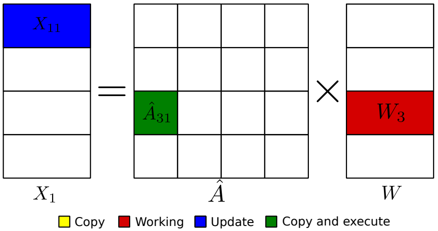
Computing X_1

1. Choose b so that blocks of size $k \times b$, b^2 and $n \times k$ fit into the GPU memory
2. Divide X_1, $\hat{A}$ and W into blocks, copy W on the GPU
3. Copy $\hat{A}_{ij}$ to GPU and update X_{1i}

$$X_{1i} = X_{1i} + \hat{A}_{ij} * W_j.$$

4. Return X_{1i} to the CPU

First step $X_1 := \hat{A}W$



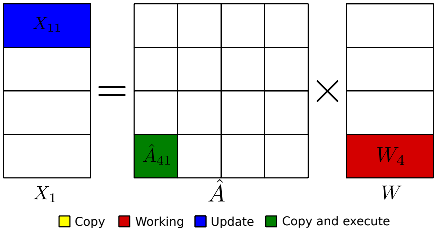
Computing X_1

1. Choose b so that blocks of size $k \times b$, b^2 and $n \times k$ fit into the GPU memory
2. Divide X_1, $\hat{A}$ and W into blocks, copy W on the GPU
3. Copy $\hat{A}_{ij}$ to GPU and update X_{1i}

$$X_{1i} = X_{1i} + \hat{A}_{ij} * W_j.$$

4. Return X_{1i} to the CPU

First step $X_1 := \hat{A}W$



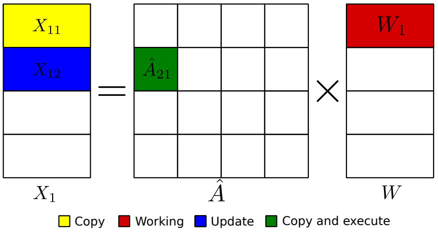
Computing X_1

1. Choose b so that blocks of size $k \times b$, b^2 and $n \times k$ fit into the GPU memory
2. Divide X_1, $\hat{A}$ and W into blocks, copy W on the GPU
3. Copy $\hat{A}_{ij}$ to GPU and update X_{1i}

$$X_{1i} = X_{1i} + \hat{A}_{ij} * W_j.$$

4. Return X_{1i} to the CPU

First step $X_1 := \hat{A}W$



Computing X_1

1. Choose b so that blocks of size $k \times b$, b^2 and $n \times k$ fit into the GPU memory
2. Divide X_1, $\hat{A}$ and W into blocks, copy W on the the GPU
3. Copy $\hat{A}_{ij}$ to GPU and update X_{1i}

$$X_{1i} = X_{1i} + \hat{A}_{ij} * W_j.$$

4. Return X_{1i} to the CPU

Steps 2 and 3

Step 2: $X_2 := \frac{1}{2} X_1^T W$

- X_2 requires $k \times k$ storage and can fit into GPU memory
- Copy block X_{1i} to the GPU at the time and update X_2:

$$X_2 = X_2 + X_{1i} W_i,$$

Step 3: $X_3 := X_1 + Y X_2$

- X_3 requires $n \times k$ space on the GPU and is update one block at the time
- Y overwrites W in the GPU memory
- Copy X_{1i} to the GPU, update X_{3i} and return it to the CPU memory

$$X_{3i} = X_{1i} + Y_i X_2.$$

Steps 2 and 3

Step 2: $X_2 := \frac{1}{2}X_1^T W$

- X_2 requires $k \times k$ storage and can fit into GPU memory
- Copy block X_{1i} to the GPU at the time and update X_2:

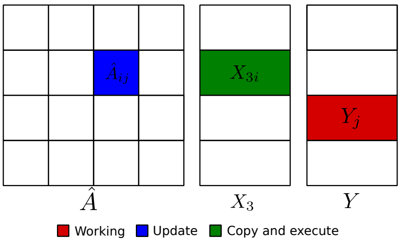
$$X_2 = X_2 + X_{1i}W_i,$$

Step 3: $X_3 := X_1 + YX_2$

- X_3 requires $n \times k$ space on the GPU and is update one block at the time
- Y overwrites W in the GPU memory
- Copy X_{1i} to the GPU, update X_{3i} and return it to the CPU memory

$$X_{3i} = X_{1i} + Y_iX_2.$$

Step 4: $\hat{A}_{ij} := \hat{A}_{ij} + X_{3i} Y_j^T + Y_i X_{3j}^T$



Update block $\hat{A}_{ij}$

1. Copy X_{3i} on the GPU and execute:

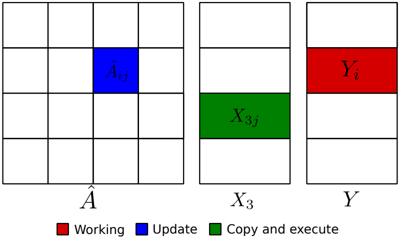
$$\hat{A}_{ij} = \hat{A}_{ij} + X_{3i} Y_j^T$$

2. Copy X_{3j} on the GPU and execute:

$$\hat{A}_{ij} = \hat{A}_{ij} + Y_i X_{3j}^T$$

3. Return $\hat{A}_{ij}$ to the CPU memory

Step 4: $\hat{A}_{ij} := \hat{A}_{ij} + X_{3i} Y_j^T + Y_i X_{3j}^T$



Update block $\hat{A}_{ij}$

1. Copy X_{3i} on the GPU and execute:

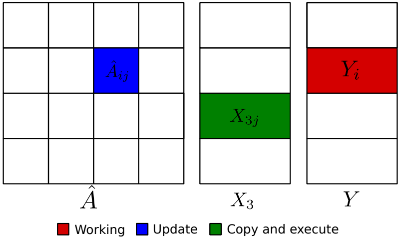
$$\hat{A}_{ij} = \hat{A}_{ij} + X_{3i} Y_j^T$$

2. Copy X_{3j} on the GPU and execute:

$$\hat{A}_{ij} = \hat{A}_{ij} + Y_i X_{3j}^T$$

3. Return $\hat{A}_{ij}$ to the CPU memory

Step 4: $\hat{A}_{ij} := \hat{A}_{ij} + X_{3i} Y_j^T + Y_i X_{3j}^T$



Update block $\hat{A}_{ij}$

1. Copy X_{3i} on the GPU and execute:

$$\hat{A}_{ij} = \hat{A}_{ij} + X_{3i} Y_j^T$$

2. Copy X_{3j} on the GPU and execute:

$$\hat{A}_{ij} = \hat{A}_{ij} + Y_i X_{3j}^T$$

3. Return $\hat{A}_{ij}$ to the CPU memory

Outline

Experimental environment

Target platform

- `peco.act.uji.es` small cluster at Univeristat Jaume I
- 8 nodes, each with 2 Intel Xeon QuadCore E5520, 24 GB memory
- GPU NVIDIA Tesla C2050, 2.6 GB global memory (ECC on)

Compilers and libraries

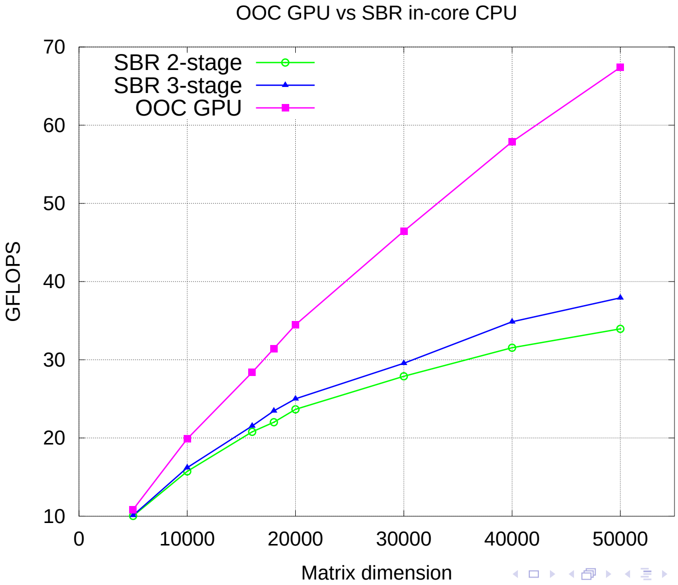
- GotoBLAS, gfortran
- Lapack 3.1.1
- CUDA 4.0, CUBLAS
- SBR Toolbox

Testing parameters

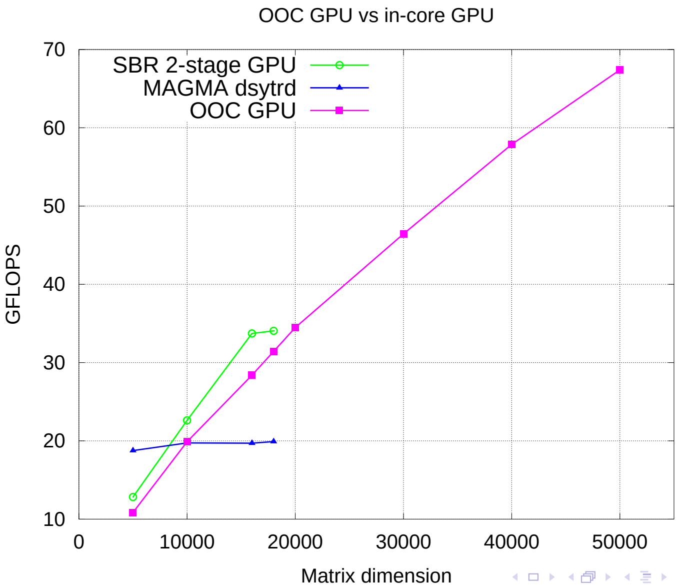
Testing parameters

- The flops count for reduction to band form: $2n^3/3$
- The total flops count for reduction from full to tridiagonal: $4n^3/3$
- We have used non-pinned (pageable) memory
- Testing were done on one node using 8 cores and one GPU in DP

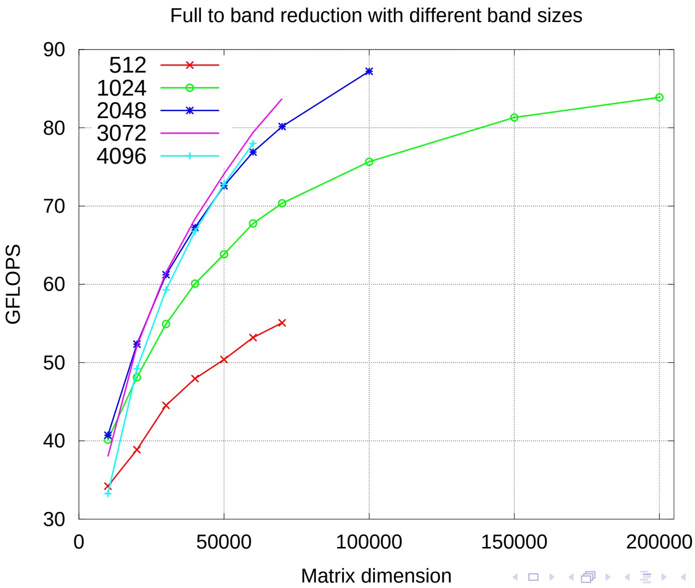
Performance: Full to tridiagonal reduction



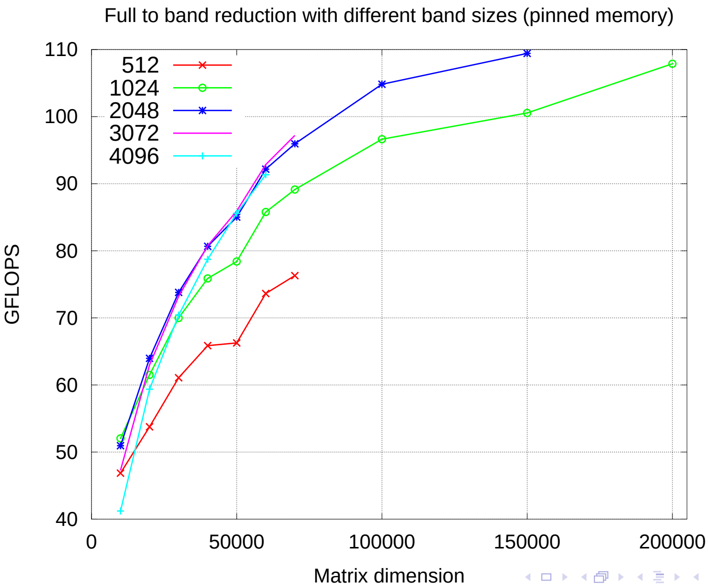
Performance: Full to tridiagonal reduction



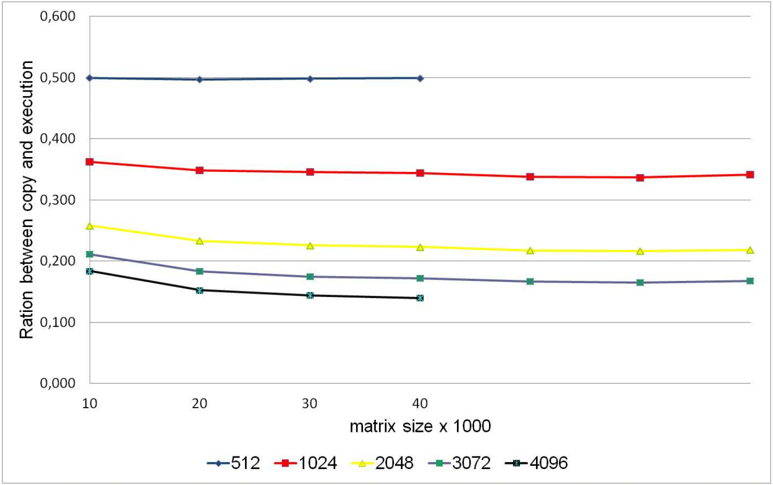
Performance: Full to band reduction



Performance: Full to band reduction



Ration between copy and execution (full → band form)



Outline

Conclusion

Current status

- We have implemented algorithm that uses OOC techniques for reducing full dense matrix to band form
- Our algorithm matched the performance of the in-core algorithm when the problem is large enough
- The algorithm is independent of the problem size

Ongoing tasks

- Overlapping copying with the computation on the GPU
- Block QR algorithm on the GPU for large matrices
- Multi-stage approach implementation on the GPU (reduction from band to narrower band)
- Accumulation of the Q when the eigenvectors are also required

Conclusion

Current status

- We have implemented algorithm that uses OOC techniques for reducing full dense matrix to band form
- Our algorithm matched the performance of the in-core algorithm when the problem is large enough
- The algorithm is independent of the problem size

Ongoing tasks

- Overlapping copying with the computation on the GPU
- Block QR algorithm on the GPU for large matrices
- Multi-stage approach implementation on the GPU (reduction from band to narrower band)
- Accumulation of the Q when the eigenvectors are also required

Conclusion

Current status

- We have implemented algorithm that uses OOC techniques for reducing full dense matrix to band form
- Our algorithm matched the performance of the in-core algorithm when the problem is large enough
- The algorithm is independent of the problem size

Ongoing tasks

- Overlapping copying with the computation on the GPU
- Block QR algorithm on the GPU for large matrices
- Multi-stage approach implementation on the GPU (reduction from band to narrower band)
- Accumulation of the Q when the eigenvectors are also required

Conclusion

Current status

- We have implemented algorithm that uses OOC techniques for reducing full dense matrix to band form
- Our algorithm matched the performance of the in-core algorithm when the problem is large enough
- The algorithm is independent of the problem size

Ongoing tasks

- Overlapping copying with the computation on the GPU
- Block QR algorithm on the GPU for large matrices
- Multi-stage approach implementation on the GPU (reduction from band to narrower band)
- Accumulation of the Q when the eigenvectors are also required

Conclusion

Current status

- We have implemented algorithm that uses OOC techniques for reducing full dense matrix to band form
- Our algorithm matched the performance of the in-core algorithm when the problem is large enough
- The algorithm is independent of the problem size

Ongoing tasks

- Overlapping copying with the computation on the GPU
- Block QR algorithm on the GPU for large matrices
- Multi-stage approach implementation on the GPU (reduction from band to narrower band)
- Accumulation of the Q when the eigenvectors are also required

Thank you for your attention!