Hybrid CPU-GPU generation of the Hamiltonian and Overlap matrices in FLAPW methods

Diego Fabregat-Traver

with Davor Davidović, Markus Höhnerbach and Edoardo di Napoli

HPAC, RWTH Aachen
fabregat@aices.rwth-aachen.de

JARA-HPC Symposium, October 4th, 2016
RWTH Aachen, Germany

Motivation

- Legacy codes in SC have grown with focus on functionality
- Problems:
  - Code lacks modularity, encapsulation, code reuse, ...
  - Code is often a direct translation of mathematical formulas
- Problems get exacerbated with recent shift to heterogeneous architectures
- “Modernize or perish”
  - Yes, it is costly
  - Yes, it is necessary and benefits are substantial

Goal

In general:

- Study of the benefits of reengineering the software
- Modular design, clear layering and interfaces
- Bottom layers: standardized and highly optimized libraries

In particular:

- FLEUR as use case
- Modernize a portion of the code that takes about 40% of the computation
- Required an important initial effort
- We now give evidence of performance portability to heterogeneous CPU + GPU architectures

Outline

FLEUR & FLAPW

Disclaimer: NOT a physicist!!! :-)

- Kohn and Sham reformulate high-dimensional Schrödinger equation as

$$\hat{H}\psi_i(\mathbf{r}) = \left(-\frac{\hbar^2}{2m}\nabla^2 + V_0(\mathbf{r})\right)\psi_i(\mathbf{r}) = \epsilon_i\psi_i(\mathbf{r})$$

- Depending on the discretization, there exist different methods
- FLEUR implements the FLAPW method to approximate the Kohn-Sham equations
- FLAPW: Expansion of LAPW.

$$\psi_i(r) = \sum_{t=1}^{N_G} c_{t,i}\phi(r)$$

- Set of eigenproblems

$$H \cdot c_i = \epsilon_i S \cdot c_i$$

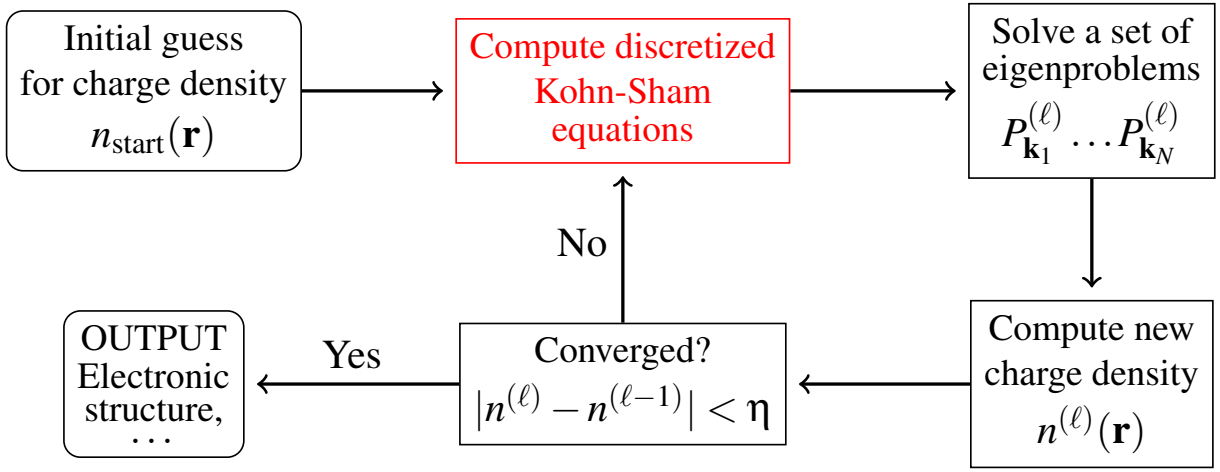
FLEUR & FLAPW

Disclaimer: NOT a physicist!!! :-)

- Iterative process (Self-Consistent Field iteration):

$$\hat{H}\psi_i(\mathbf{r}) = \left(-\frac{\hbar^2}{2m}\nabla^2 + V_0(\mathbf{r})\right)\psi_i(\mathbf{r}) = \epsilon_i\psi_i(\mathbf{r})$$

$$n(\mathbf{r}) = \sum_i^N |\psi_i(\mathbf{r})|^2$$



- We focus on the construction of H and S

Constructing H and S

$$H_{G',G} = \sum_{a=1}^{N_A} \sum_{L',L} \left(\left(A_{L'}^{a,G'} \right)^* T_{L',L;a}^{[AA]} A_L^{a,G} \right) + \left(\left(A_{L'}^{a,G'} \right)^* T_{L',L;a}^{[AB]} B_L^{a,G} \right)$$
$$+ \left(\left(B_{L'}^{a,G'} \right)^* T_{L',L;a}^{[BA]} A_L^{a,G} \right) + \left(\left(B_{L'}^{a,G'} \right)^* T_{L',L;a}^{[BB]} B_L^{a,G} \right)$$

$$S_{G',G} = \sum_{a=1}^{N_A} \sum_{L} \left(A_L^{a,G'} \right)^* A_L^{a,G} + \left(B_L^{a,G'} \right)^* B_L^{a,G} \| \dot{u}_l^a \|^2$$

Rewritten as matrix operations

$$H = \sum_{a=1}^{N_A} \underbrace{A_a^H T_a^{[AA]} A_a}_{H_{AA}} + \underbrace{A_a^H T_a^{[AB]} B_a + B_a^H T_a^{[BA]} A_a + B_a^H T_a^{[BB]} B_a}_{H_{AB+BA+BB}}$$

$$S = \sum_{a=1}^{N_A} A_a^H A_a + B_a^H U_a^H U_a B_a,$$

Constructing H and S

$$H_{AB+BA+BB} = \sum_{a=1}^{N_A} B_a^H (T_a^{[BA]} A_a) + (A_a^H T_a^{[AB]}) B_a + \frac{1}{2} B_a^H (T_a^{[BB]} B_a) + \frac{1}{2} (B_a^H T_a^{[BB]}) B_a$$

Constructing H and S

$$
\begin{aligned}
H_{AB+BA+BB} = \sum_{a=1}^{N_A} & B_a^H (T_a^{[BA]} A_a) + (A_a^H T_a^{[AB]}) B_a + \\
& \frac{1}{2} B_a^H (T_a^{[BB]} B_a) + \frac{1}{2} (B_a^H T_a^{[BB]}) B_a \\
= \sum_{a=1}^{N_A} & B_a^H (T_a^{[BA]} A_a + \frac{1}{2} T_a^{[BB]} B_a) + \\
& (A_a^H T_a^{[AB]} + \frac{1}{2} B_a^H T_a^{[BB]}) B_a
\end{aligned}
$$

Constructing H and S

$$
\begin{aligned}
H_{AB+BA+BB} = \sum_{a=1}^{N_A} & B_a^H (T_a^{[BA]} A_a) + (A_a^H T_a^{[AB]}) B_a + \\
& \frac{1}{2} B_a^H (T_a^{[BB]} B_a) + \frac{1}{2} (B_a^H T_a^{[BB]}) B_a \\
= \sum_{a=1}^{N_A} & B_a^H (T_a^{[BA]} A_a + \frac{1}{2} T_a^{[BB]} B_a) + \\
& (A_a^H T_a^{[AB]} + \frac{1}{2} B_a^H T_a^{[BB]}) B_a
\end{aligned}
$$

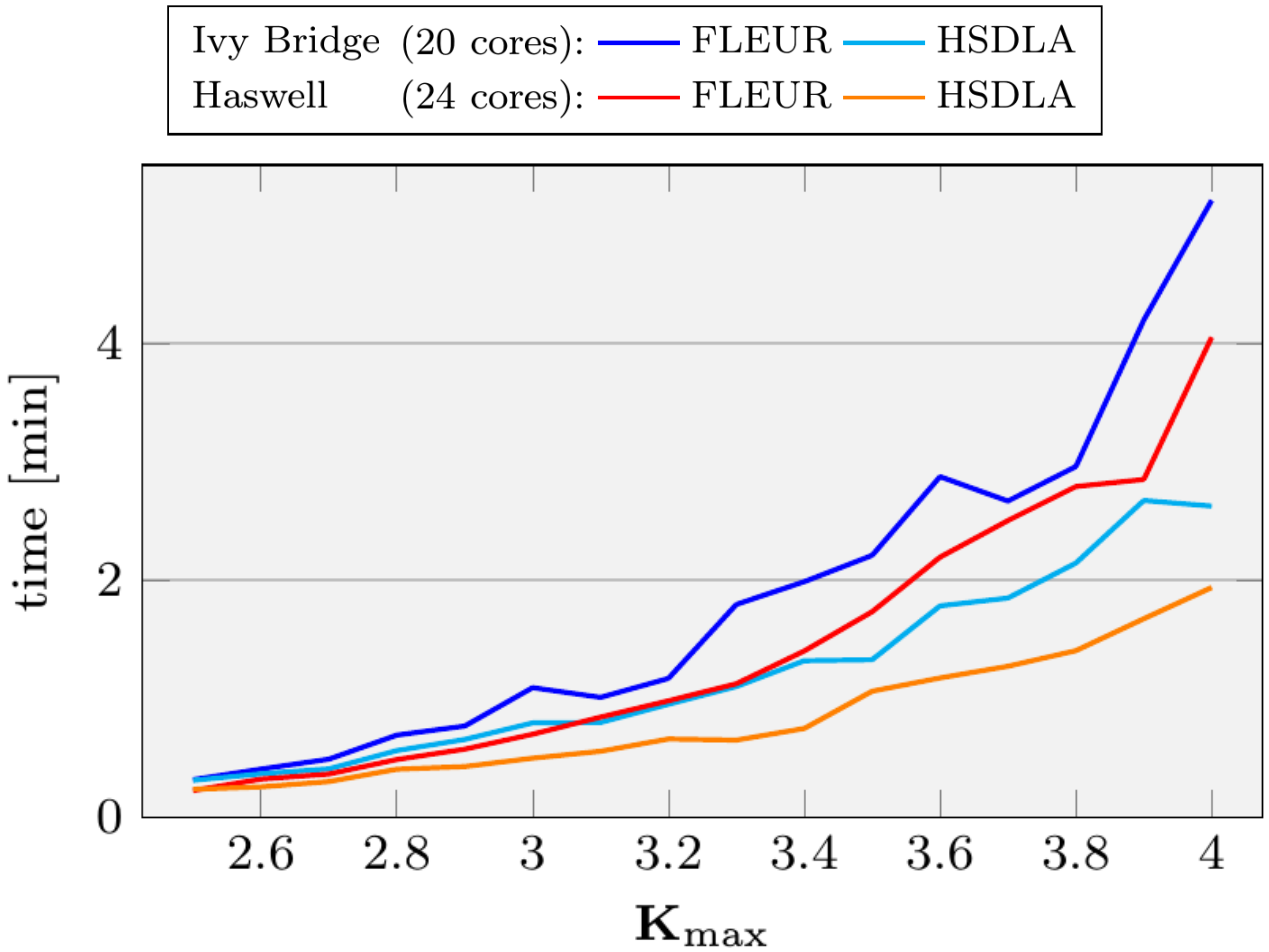
1: **for** $a := 1 \rightarrow N_A$ **do**
2: $\quad Z_a = T_a^{[BA]} A_a$ ▷ (`zgemm`: $8 N_L^2 N_G$ Flops)
3: $\quad Z_a = Z_a + \frac{1}{2} T_a^{[BB]} B_a$ ▷ (`zhemm`: $8 N_L^2 N_G$ Flops)
4: **end for**
5: $H = Z^H B + B^H Z$ ▷ (`zher2k`: $8 N_A N_L N_G^2$ Flops)

Stripped algorithm

1: Create A, B
2: // $H_{AB+BA+BB}$
3: **for** $a := 1 \rightarrow N_A$ **do**
4: $\quad Z_a = T_a^{[BA]} A_a$ ▷ (`zgemm`: $8N_L^2 N_G$ Flops)
5: $\quad Z_a = Z_a + \frac{1}{2} T_a^{[BB]} B_a$ ▷ (`zhemm`: $8N_L^2 N_G$ Flops)
6: **end for**
7: $H = Z^H B + B^H Z$ ▷ (`zher2k`: $8N_A N_L N_G^2$ Flops)
8: // S
9: $S = A^H A$ ▷ (`zherk`: $4N_A N_L N_G^2$ Flops)
10: $B = UB$ ▷ (`scaling`: $2N_A N_L N_G$ Flops)
11: $S = S + B^H B$ ▷ (`zherk`: $4N_A N_L N_G^2$ Flops)
12: // H_{AA}
13: **for** $a := 1 \rightarrow N_A$ **do**
14: $\quad$ **try:**
15: $\quad\quad C_a = Cholesky(T_a^{[AA]})$ ▷ (`zpotrf`: $\frac{4}{3} N_L^3$ Flops)
16: $\quad$ **success:**
17: $\quad\quad Y_a = C_a^H A_a$ ▷ (`ztrmm`: $4N_L^2 N_G$ Flops)
18: $\quad$ **failure:**
19: $\quad\quad X_a = T_a^{[AA]} A_a$ ▷ (`zhemm`: $8N_L^2 N_G$ Flops)
20: **end for**
21: $H = H + A^H_{\neg \text{HPD}} X_{\neg \text{HPD}}$ ▷ (`zgemm`: $8N_{A_{\neg \text{HPD}}} N_L N_G^2$ Flops)
22: $H = H + Y^H_{\text{HPD}} Y_{\text{HPD}}$ ▷ (`zherk`: $4N_{A_{\text{HPD}}} N_L N_G^2$ Flops)

Previous multi-core results

Test case: NaCl ($N_A = 512$, $N_L = 49$, $N_G = [2256 - 9273]$)



Porting to heterogeneous architectures

So ...

- You rewrote the code in terms of standardized libraries
- Got a speedup of about 2x
- Is that it?

Porting to heterogeneous architectures

So ...

- You rewrote the code in terms of standardized libraries
- Got a speedup of about 2x
- Is that it?

Not at all :)

- BLAS is the first numerical library ported to every new architecture
- On paper: quick and easy port to other architectures
- The questions are:
  - Can we port to CPU+GPU with minimal modifications?
  - How far can we get?

Porting to heterogeneous architectures

Back-of-the-envelope analysis

- 5 lines of the algorithm constitute 97% of flops
- Correspond to BLAS-3 operations (`gemm`, `herk`, `her2k`)
- High arithmetic intensity and should fit GPUs well

Porting to heterogeneous architectures

Back-of-the-envelope analysis

- 5 lines of the algorithm constitute 97% of flops
- Correspond to BLAS-3 operations (`gemm`, `herk`, `her2k`)
- High arithmetic intensity and should fit GPUs well

- First step: offload these routine calls
- All 5 are BLAS kernels. Can we use some library?
  - cuBLAS
  - cuBLAS-XT
  - MAGMA
  - BLASX

Porting to heterogeneous architectures

Back-of-the-envelope analysis

- 5 lines of the algorithm constitute 97% of flops
- Correspond to BLAS-3 operations (`gemm`, `herk`, `her2k`)
- High arithmetic intensity and should fit GPUs well

- First step: offload these routine calls
- All 5 are BLAS kernels. Can we use some library?
  - cuBLAS
  - cuBLAS-XT
  - MAGMA
  - BLASX

Porting to heterogeneous architectures

Back-of-the-envelope analysis

- 5 lines of the algorithm constitute 97% of flops
- Correspond to BLAS-3 operations (`gemm`, `herk`, `her2k`)
- High arithmetic intensity and should fit GPUs well

- First step: offload these routine calls
- All 5 are BLAS kernels. Can we use some library?
  - cuBLAS
  - cuBLAS-XT
  - MAGMA
  - BLASX

Porting to heterogeneous architectures

Additional code?

3 x wrappers around the BLAS calls:

```
void gpu_zgemm_( char *transa, char *transb, int *m,  int *n, int *k,
                 std::complex<double> *alpha,  std::complex<double> *A,  int *lda,
                 std::complex<double> *B,  int *ldb,
                 std::complex<double> *beta, std::complex<double> *C,  int *ldc )
{
    cublasOperation_t cu_transa = transa[0] == 'N' ? CUBLAS_OP_N :
                                  transa[0] == 'T' ? CUBLAS_OP_T : CUBLAS_OP_C;
    cublasOperation_t cu_transb = transb[0] == 'N' ? CUBLAS_OP_N :
                                  transb[0] == 'T' ? CUBLAS_OP_T : CUBLAS_OP_C;
    cublasXtZgemm( handle, cu_transa, cu_transb, *m, *n, *k,
                   (cuDoubleComplex *)alpha, (cuDoubleComplex *)A, *lda,
                                             (cuDoubleComplex *)B, *ldb,
                   (cuDoubleComplex *)beta,  (cuDoubleComplex *)C, *ldc );
}
```

Porting to heterogeneous architectures

Additional code?

- 3x wrappers (zgemm, zherk, zher2k)
- Init and cleanup of cuda runtime and devices
  - Get #devices, initialize devices, create handlers, ...
  - Destroy handlers, free devices, ...
- Allocate data in page-locked memory
  - Avoid “hidden” copies
  - Fast data transfer

Porting to heterogeneous architectures

Additional code?

- 3x wrappers (zgemm, zherk, zher2k)
- Init and cleanup of cuda runtime and devices
  - Get #devices, initialize devices, create handlers, ...
  - Destroy handlers, free devices, ...
- Allocate data in page-locked memory
  - Avoid “hidden” copies
  - Fast data transfer

Only around 100 lines of additional code

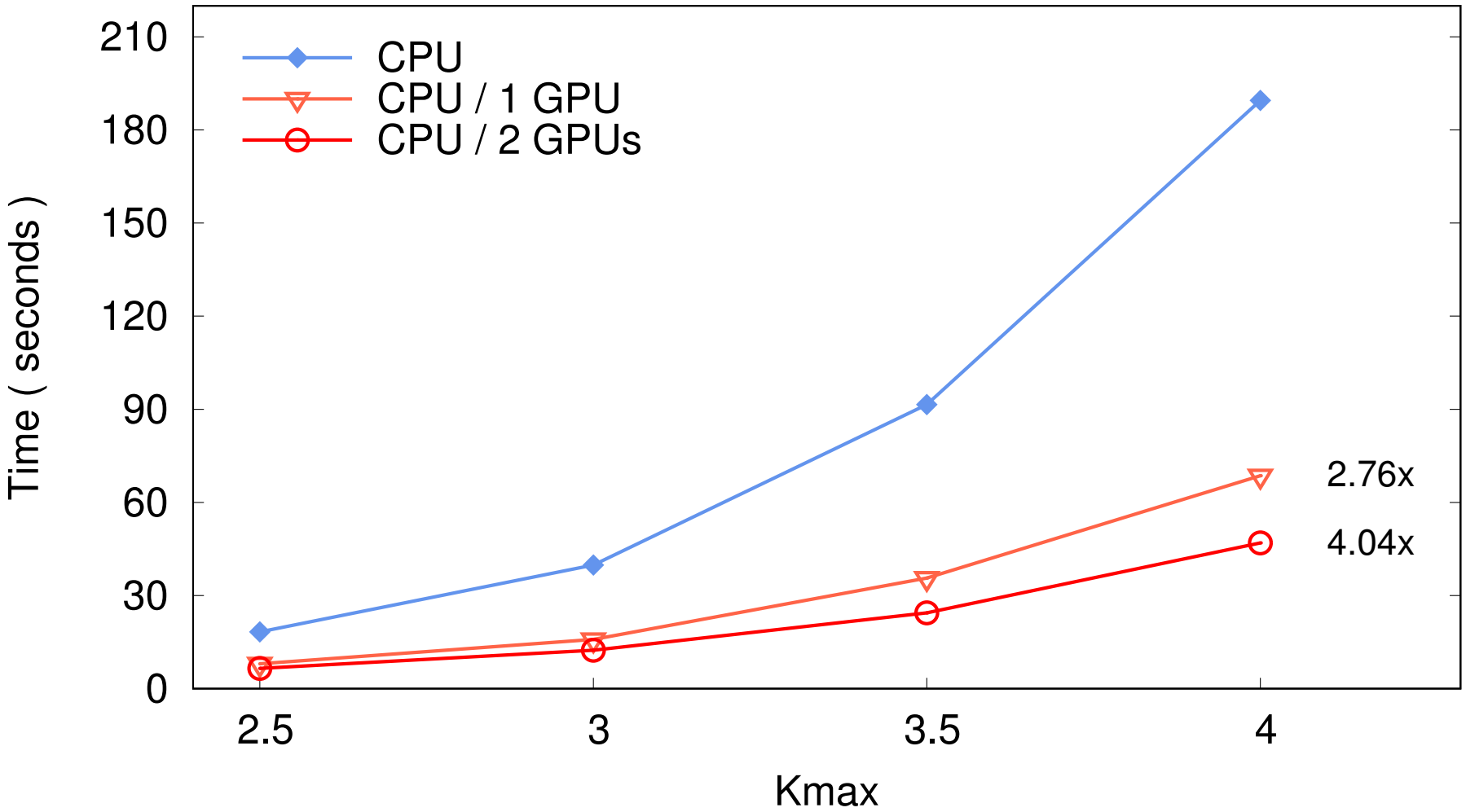
Experimental results

Sandy Bridge:

- CPU: E5-2650, 2 x 8core, 2.0GHz, 64GBs RAM
- 2 Nvidia Tesla K20X
- Peak performance: 256 GFs/s + 2 x 1.3 TFs/s

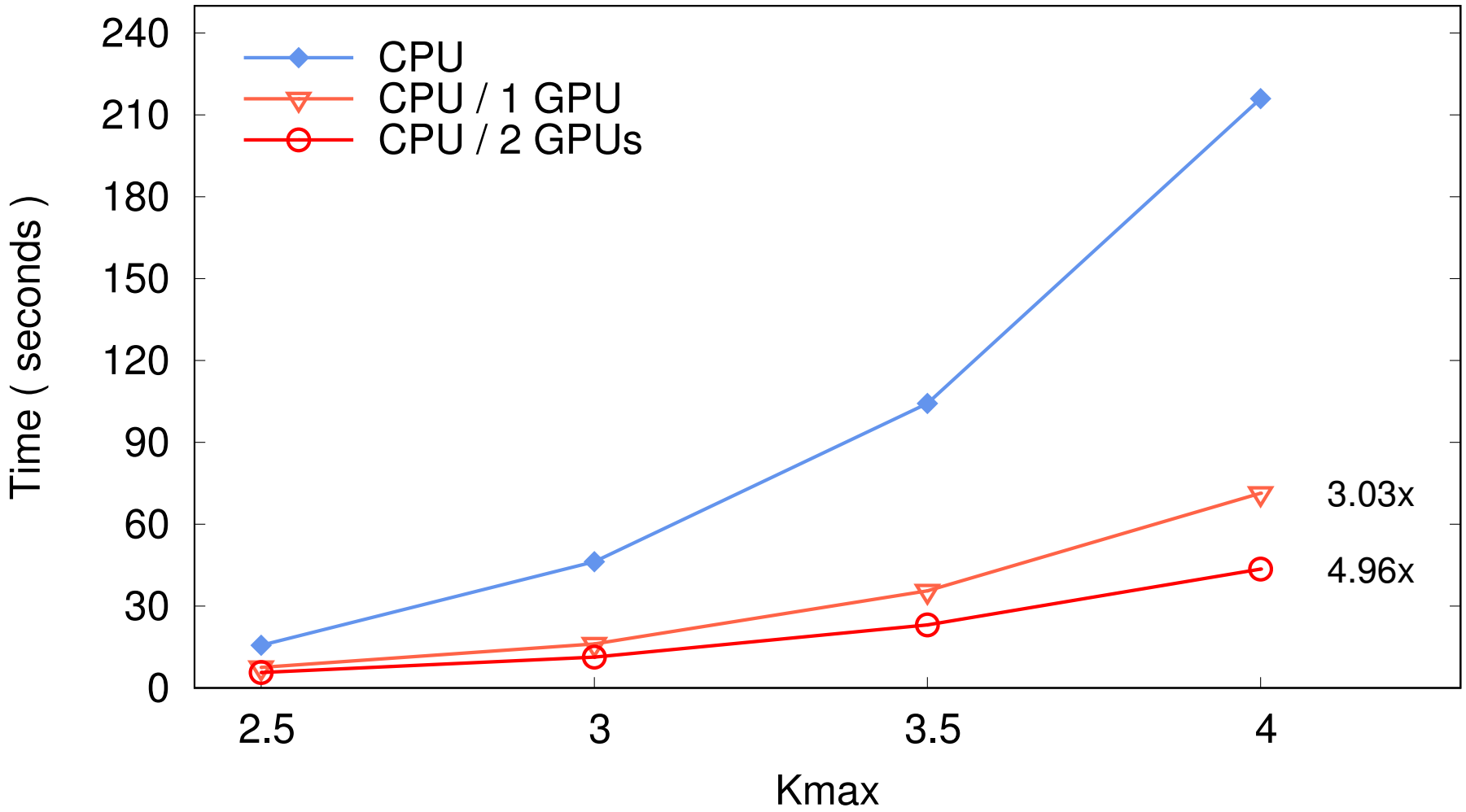
Experimental results

Test case 1: NaCl ($N_A = 512$, $N_L = 49$, $N_G = [2256 - 9273]$)



Experimental results

Test case 2: AuAg ($N_A = 108$, $N_L = 121$, $N_G = [3275 - 13379]$)



Breakdown of timings

Section (line(s))	Time	Performance	Eff (CPU/GPU)	Eff (Full system)
Loop 1 (3–6)	2.27 secs	80.35 GFlops/s	0.31	0.03
Loop 2 (13–20)	2.62 secs	34.81 GFlops/s	0.14	0.01
U norm (10)	0.23 secs	1.01 GFlops/s	0.00	0.00
S_A	4.37 secs	1974.63 GFlops/s	0.75	0.69
S_{BU}	4.41 secs	1956.72 GFlops/s	0.75	0.68
$H_{AB+BA+BB}$	9.49 secs	1818.57 GFlops/s	0.69	0.63
$H_{AA\ (HPD)}$	2.32 secs	1859.72 GFlops/s	0.71	0.65
$H_{AA\ (\neg HPD)}$	4.75 secs	1816.66 GFlops/s	0.69	0.63

Table: Results for NaCl ($K_{max} = 4.0$).

How to improve these results

- Hybrid code for the “loops”
- Tuning the execution of cuBLAS-XT
- Hybrid implementation of BLAS routines

How to improve these results

- Hybrid code for the “loops”
- Tuning the execution of cuBLAS-XT
- Hybrid implementation of BLAS routines

Currently:

- Hybrid `zgemm`
- Tuning block size for cuBLAS-XT

zgemm improvements

K_{max}	2.5	3.0	3.5	4.0
cuBLAS-XT	0.57	0.69	0.64	0.63
Hybrid us	0.68	0.75	0.77	0.70

Table: NaCl - Efficiency

zgemm improvements

K_{max}	2.5	3.0	3.5	4.0
cuBLAS-XT	0.57	0.69	0.64	0.63
Hybrid us	0.68	0.75	0.77	0.70

Table: NaCl - Efficiency

K_{max}	2.5	3.0	3.5	4.0
cuBLAS-XT	0.54	0.66	0.71	0.66
Hybrid us	0.71	0.77	0.80	0.77

Table: AuAg - Efficiency

zgemm improvements

K_{max}	2.5	3.0	3.5	4.0
cuBLAS-XT	0.57	0.69	0.64	0.63
Hybrid us	0.68	0.75	0.77	0.70

Table: NaCl - Efficiency

K_{max}	2.5	3.0	3.5	4.0
cuBLAS-XT	0.54	0.66	0.71	0.66
Hybrid us	0.71	0.77	0.80	0.77

Table: AuAg - Efficiency

Improvements from 10% to 30%

Conclusions and Future Work

Conclusions:

- Modernizing legacy code is critical
- Layered design built on top of standardized libraries
- Increase in performance
- Quick & easy performance portability
- Case of FLEUR: up to 12x speedup

Conclusions and Future Work

Conclusions:

- Modernizing legacy code is critical
- Layered design built on top of standardized libraries
- Increase in performance
- Quick & easy performance portability
- Case of FLEUR: up to 12x speedup

Future Work:

- Hybrid for `zgemm`, `zherk`, `zher2k`
- Experiments on Jureca (4 GPU devices)
- Reduction in computational cost (à la FLEUR)

Thank you for your attention!

Details on the original HSDLA:

- High-performance generation of the Hamiltonian and Overlap matrices in FLAPW methods. *Edoardo Di Napoli, Elmar Peise, Markus Hrywniak and Paolo Bientinesi*. Accepted in CPC.

DFG DAAD

Financial support from DFG through Grant BI 1533/2-1 and from DAAD through the *PPP Kroatien* program is gratefully acknowledged

We also thank the RWTH ITC Center for the computing resources