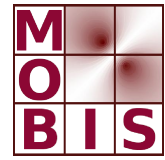




SpezialForschungsBereich F 32



Karl-Franzens Universität Graz
Technische Universität Graz
Medizinische Universität Graz



Hands on KNL with the Sandbox Hackathon example Techreport

S. Rosenberger G. Haase

SFB-Report No. 2017-005

April 2017

A-8010 GRAZ, HEINRICHSTRASSE 36, AUSTRIA

Supported by the
Austrian Science Fund (FWF)



SFB sponsors:

- **Austrian Science Fund (FWF)**
- **University of Graz**
- **Graz University of Technology**
- **Medical University of Graz**
- **Government of Styria**
- **City of Graz**



Hands on KNL with the Sandbox Hackathon example Techreport

Stefan Rosenberger and Gundolf Haase

April 4, 2017

CONTENTS

1	General	3
1.1	Improofment steps	3
2	Bios settings for KNL	4
2.1	All-to-all	4
2.2	Quadrant / Hemisphere	5
2.3	SNC-4 / SNC-2	5
3	Memory bandwith optimization	5
3.1	Out of the Box	5
3.2	Tell Compiler to tune for Xeon Phi: -xMIC-AVX512	5
3.3	Reduce Variablility by setting Bandwidth	6
3.4	Number of Threads	7
3.4.1	Methode 1 (OpenMP standard)	7
3.4.2	Methode 2 (Only for Intel OpenMP and KNL)	7
3.5	Additional notes for Memory handling	7
3.6	Finally apply to our CODE	7
4	Bonus for Memory with the Package memkind	8
5	Vectorization with AVX-512	8
5.1	Find Conflicts in vectorized loops	8
5.2	Use OpenMP	8

6	Mathlibrary	9
6.1	Nested loops	9
6.2	Optimize the linear Operator, or try's which didn't work	9
6.2.1	First try	10
6.2.2	Second try	11
6.2.3	Third try	11
7	AMG on KNL	12
7.1	First without spetial flags	12
7.2	Second with High bandwidth memory	12
7.3	Third with Vectorization	12
7.4	Forth with definition of Threads	12
8	Short first Conclusion	13
9	Improvement for KNL	13
9.1	Thread Pinning	13
9.1.1	Tests with the Environment Variables	13
9.1.2	Thread Pinning with Jacobi	14
10	Test with KMP_AFFINITY	15
11	Summary of this Techreport	15

1 GENERAL

1.1 IMPROOFMENT STEPS

We consider only the different improofments for the scalar product, and the influence to our performance. Note, we consider only the parts of `_OPENMP`:

The basic C++ code:

```
template<class S>
void scalar_product_acc(const toolbox_vector<S> &x, const toolbox_vector<S> &y, S &s)
{
    S s = 0.0;

    const S *__restrict x = _x.data(), *__restrict y = _y.data();
    const int x_size = _x.size();
    const int y_size = _y.size();
    int i;
    //cout << "atest= " << _s << endl;
    #ifdef _OPENACC
        #pragma acc parallel loop vector_length(_block_size) pcopyin(x[0:x_size], y[0:
            y_size])
    #elif defined(_OPENMP)
        // No parallelization
    #endif
    for(i = 0; i < x_size; i++)
    {
        s += x[i] * y[i];
    }
    _s = s;
}
```

🕒 Timing:

- Program Solve : 10.3973s
- Scalar Product: 0.35599s

Pragma Type 1:

```
#pragma omp parallel for reduction(+:s)
```

🕒 Timing:

- Program Solve : 10.1233s
- Scalar Product: 0.0614021s

Pragma Type 2:

```
#pragma omp parallel for private(i) shared(x,y) schedule(static) reduction(+:s)
```

🕒 Timing:

- Program Solve : 9.38401s
- Scalar Product: 0.060s

👉 Note that we observe ± 0.02 s time *jumps* for the measurement of the scalar product.

Numa Nodes: We use the following command to execute the program:

```
numactl --membind 1 ./sandbox.ICC_KNL_ genBi_ellmat.bin genBi_rhs_1e
```

Means in the KNL Introduction: *Binding the Application to HBM (Flat/Hybrid)*¹

👉 Timing:

- Program Solve : 12.226s
- Scalar Product: 0.07409s

👉 Leads to slow down!

ATTENTION: It force the code to one core. Therefore, MPI is useless!!!!

Malloc not working

```
// Includes special for Intel KNL in toolbox.h:  
#include <hbwmalloc.h>  
.  
.  
.  
hbw_free(X);
```

wherein we compile and link the program with: -lmemkind

👉 We get an error: *error: identifier hbw_free is undefined.*

👉 We need to download the open source library *memkind* at:

```
memkind.github.io/memkind
```

More info at

```
colfaxresearch.com/knl-mcdram
```

2 BIOS SETTINGS FOR KNL

One should change the Clustering mode in the bios setting for the application we use.

2.1 ALL-TO-ALL

Basically for debugging.

¹HMB ... High bandwidth memory

☞ *All-to-all* is a bad choice! Long paths for the data transfer!
Suggestion from Intel: **never use it!**

2.2 QUADRANT / HEMISPHERE

For programmer interesting choice.

Quadrant is good if we do not know which to use!

2.3 SNC-4 / SNC-2

Cluster the nodes in 4 or 2 Numa Nodes. **We need to modify code!** The tiles have to work with local memory!

One can do that via Nested openMP loops, or split MPI threads (4) and apply openMP!

3 MEMORY BANDWIDTH OPTIMIZATION

3.1 OUT OF THE BOX

On-platform memory (DDR4) Simply compile with OpenMP and run the code.

☞ Can not reproduce the memory output from the colfax tutorial!!

```
... / mpicxx -cxx=icpc -c -IPT_C -DFAST_AMG -DP2P_v1 -std=c++11 -DNDEBUG -qopenmp -  
DOPENMP -o sandbox_example.o sandbox_example.cpp  
.  
./ sandbox.ICC_KNL_ genBi_ellmat.bin genBi_rhs_Ie
```

☞ Timing:

- Program Solve : 12.5508s
- Scalar Product: 0.122052s

On Package HBM (MCDRAM)

```
numactl -m 1 ./ sandbox.ICC_KNL_ genBi_ellmat.bin genBi_rhs_Ie
```

☞ Timing:

- Program Solve : 11.1585s
- Scalar Product: 0.047848s

3.2 TELL COMPILER TO TUNE FOR XEON PHI: -xMIC-AVX512

On-platform memory (DDR4) Compile with -xMIC-AVX512 and run the code.

☞ Can not reproduce the memory output from the colfax tutorial!!

```
.../mpicxx -cxx=icpc -c -IPT_C -DFAST_AMG -DP2P_v1 -std=c++11 -DNDEBUG -qopenmp -
DOPENMP -o sandbox_example.o -xMIC-AVX512 sandbox_example.cpp
.
.
./sandbox.ICC_KNL_ genBi_ellmat.bin genBi_rhs_Ie
```

👉 Timing:

- Program Solve : 12.2986s
- Scalar Product: 0.100564s

On Package HBM (MCDRAM)

```
numactl -m 1 ./sandbox.ICC_KNL_ genBi_ellmat.bin genBi_rhs_Ie
```

👉 Timing:

- Program Solve : 11.0357s
- Scalar Product: 0.0484693s

3.3 REDUCE VARIABILITY BY SETTING BANDWIDTH

On-platform memory (DDR4) Compile with -DSTREAM_ARRAY_SIZE=64000000 (max number) and run the code.

👉 Can not reproduce the memory output from the colfax tutorial!!

```
.../mpicxx -cxx=icpc -c -IPT_C -DFAST_AMG -DP2P_v1 -std=c++11 -DNDEBUG -qopenmp -
DOPENMP-DSTREAM_ARRAY_SIZE=64000000 -o sandbox_example.o -xMIC-AVX512
sandbox_example.cpp
.
.
./sandbox.ICC_KNL_ genBi_ellmat.bin genBi_rhs_Ie
```

👉 Timing:

- Program Solve : 12.3616s
- Scalar Product: 0.101277s

On Package HBM (MCDRAM)

```
numactl -m 1 ./sandbox.ICC_KNL_ genBi_ellmat.bin genBi_rhs_Ie
```

👉 Timing(5 runs, all between):

- Program Solve : 11.07s – 11.1356s
- Scalar Product: 0.0466938s – 0.048238s

Without this option, we have 0.047 – 0.049. The Tutorial example is very small, and calles stream.c . Possibly this option is not useful for us.

3.4 NUMBER OF THREADS

By default: OpenMP uses # threads = # logical CPU's

Bandwidth sensitive applications prefer 1thread /core!

3.4.1 METHODE 1 (OPENMP STANDARD)

Use export OMP_NUM_THREADS=64. But this is platform dependent!

3.4.2 METHODE 2 (ONLY FOR INTEL OPENMP AND KNL)

On-platform memory (DDR4) Compile with *export KMP_HW_SUBSET=<cores>C,<threads>t* and run the code.

We request 1 thread per core

```
export KMP_HW_SUBSET=1 t
.
.
./ sandbox.ICC_KNL_ genBi_ellmat.bin genBi_rhs_Ie
```

 Timing:

- Program Solve : 3.58638s
- Scalar Product: 0.0872905s

On Package HBM (MCDRAM)

```
numactl -m 1 ./ sandbox.ICC_KNL_ genBi_ellmat.bin genBi_rhs_Ie
```

 Timing(5 runs, all between):

- Program Solve : 3.23264s
- Scalar Product: 0.0423851s

3.5 ADDITIONAL NOTES FOR MEMORY HANDLING

One can optimize the prefetching via *-opt-prefetch-distance=64,8*.

3.6 FINALLY APPLY TO OUR CODE

On-platform memory (DDR4) Compile with *export KMP_HW_SUBSET=<cores>C,<threads>t* and run the code.

We request 1 thread per core

```
export KMP_HW_SUBSET=1 t
.
.
... / mpicxx -cxx=icpc -c -IPT_C -DFAST_AMG -DP2P_v1 -std=c++11 -fmax-errors=1 -O3 -fma -
DNDEBUG -fargument-noalias -fargument-noalias-global -ansi-alias -align -qopt-
dynamic-align -mkl -qopenmp -DOPENMP -xMIC-AVX512 -DSTREAM_ARRAY_SIZE=64000000 -o
sandbox_example.o sandbox_example.cpp
.
```

```
.
... / mpicxx -cxx=icpc -O3 -mkl -qopenmp -DOPENMP sandbox_example.o -o sandbox.ICC_KNL_
.
.
numactl --membind 1 ./ sandbox.ICC_KNL_ genBi_ellmat.bin genBi_rhs_le
```

🕒 Timing:

- Program Solve : 2.77367s
- Scalar Product: 0.0420296s

That's an improvement!!!!

4 BONUS FOR MEMORY WITH THE PACKAGE MEMKIND

Up to now, not installed!!

5 VECTORIZATION WITH AVX-512

5.1 FIND CONFLICTS IN VECTORIZED LOOPS

With the option `-qopt-report=5` the Intel-Compiler produce an outputfile for the vectorization (5 highest repootry)! Compiler version 15+ requiered. (Similar to OpenACC PGI_ACC_NOTIFY)

`-xMIC-AVX512` to get grid of vector dependence!

It is better to use multiplikations instead of division in the loops (if possible, create invers before the vectorized region).

If one use `-S` instead of `-c` for compiling, we get the assambling code!

Test the ellpack code for KNL! should improve the vectorization.

alternative, create an addional loop over the vectorized region, with *jumps* `ii+=vector_length !!`

5.2 USE OPENMP

For reduction, one can use instead of

```
#pragma omp parallel for
for() {

#pragma omp atomic
}
```

the parallelization

```
#pragma omp parallel
{
    int container[n];
    container[0:n] = 0;
    .
    .
#pragma omp for
    for() {

    }
```

```

    for() {
#pragma omp atomic
    }
    .
    .
}

```

But that is obvious!

6 MATHLIBRARY

Note that the option -mkl uses OpenMP behind the scenes. For explicit pragmas use -qopenmp.

6.1 NESTED LOOPS

Kill the loops and use DftiSetValue instead of the loop! One passes a batch int mkl!
Change the allocator malloc to mkl_malloc!!

6.2 OPTIMIZE THE LINEAR OPERATOR, OR TRY'S WHICH DIDN'T WORK

We start with this code:

```

#pragma omp parallel for schedule(guided)
for(int i = 0; i < size; i++)
{
    S s = v_data[i];
    const T *__restrict p_col = col + dsp[i];
    const S *__restrict p_ele = ele + dsp[i];
    const T csize = cnt[i];

    for(T j = 0; j < csize; j++)
    {
        T q = p_col[j];
        S a = p_ele[j];
        s += a * u_data[q];
    }
    v_data[i] += s;
}

```

We get the following information from the intel compiler if we compile with -qopt-report=5:

```

Report from: OpenMP optimizations [openmp]

PT_C/toolbox/generic_operator/generic_operator.h(866:4-866:4):OMP:
_ZNK19linear_operator_acclidE13matrix_vectorERK14toolbox_vectorIdERS2_: OpenMP DEFINED
LOOP WAS PARALLELIZED

Report from: Loop nest, Vector & Auto-parallelization optimizations [loop, vec, par]

LOOP BEGIN at PT_C/toolbox/generic_operator/generic_operator.h(867,4)
remark #25096: Loop Interchange not done due to: Imperfect Loop Nest (Either at Source or
due to other Compiler Transformations)
remark #25451: Advice: Loop Interchange, if possible, might help loopnest. Suggested
Permutation : ( 1 2 ) --> ( 2 1 )

```

```

remark #15542: loop was not vectorized: inner loop was already vectorized

LOOP BEGIN at PT_C/toolbox/generic_operator/generic_operator.h(876,5)
<Peeled loop for vectorization>
  remark #15389: vectorization support: reference p_ele[j] has unaligned access [ PT_C/
    toolbox/generic_operator/generic_operator.h(879,12) ]
  remark #15389: vectorization support: reference p_col[j] has unaligned access [ PT_C/
    toolbox/generic_operator/generic_operator.h(878,12) ]
  remark #15381: vectorization support: unaligned access used inside loop body
  remark #15305: vectorization support: vector length 8
  remark #15309: vectorization support: normalized vectorization overhead 1.162
  remark #15301: PEEL LOOP WAS VECTORIZED
  remark #25015: Estimate of max trip count of loop=1
LOOP END

LOOP BEGIN at PT_C/toolbox/generic_operator/generic_operator.h(876,5)
  remark #15389: vectorization support: reference p_ele[j] has unaligned access [ PT_C/
    toolbox/generic_operator/generic_operator.h(879,12) ]
  remark #15389: vectorization support: reference p_col[j] has unaligned access [ PT_C/
    toolbox/generic_operator/generic_operator.h(878,12) ]
  remark #15381: vectorization support: unaligned access used inside loop body
  remark #15415: vectorization support: irregularly indexed load was generated for the
    variable <u_data>, part of index is read from memory [ PT_C/toolbox/
    generic_operator/generic_operator.h(880,15) ]
  remark #15305: vectorization support: vector length 16
  remark #15309: vectorization support: normalized vectorization overhead 0.932
  remark #15300: LOOP WAS VECTORIZED
  remark #15442: entire loop may be executed in remainder
  remark #15450: unmasked unaligned unit stride loads: 2
  remark #15462: unmasked indexed (or gather) loads: 1
  remark #15475: --- begin vector cost summary ---
  remark #15476: scalar cost: 11
  remark #15477: vector cost: 2.750
  remark #15478: estimated potential speedup: 3.410
  remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at PT_C/toolbox/generic_operator/generic_operator.h(876,5)
<Remainder loop for vectorization>
  remark #15389: vectorization support: reference p_ele[j] has unaligned access [ PT_C/
    toolbox/generic_operator/generic_operator.h(879,12) ]
  remark #15389: vectorization support: reference p_col[j] has unaligned access [ PT_C/
    toolbox/generic_operator/generic_operator.h(878,12) ]
  remark #15381: vectorization support: unaligned access used inside loop body
  remark #15305: vectorization support: vector length 8
  remark #15309: vectorization support: normalized vectorization overhead 1.162
  remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END
LOOP END

```

6.2.1 FIRST TRY

I tried (following a online tutorial):

```

{
  const int vecLen_KNL = 64;

  for(int ii = 0; ii < size; ii+=vecLen_KNL)

```

```

{
    // Note: we need the size of sizeof(S) * vecLen_KNL space
    S value[vecLen_KNL] __attribute__((aligned(sizeof(S) * vecLen_KNL)));
    for(int jj=ii; jj<ii+vecLen_KNL && jj<size; jj++)
    {
        value[jj-ii] = v_data[jj];
        const T *__restrict p_col = col + dsp[jj];
        const S *__restrict p_ele = ele + dsp[jj];

        const int csize = (int) cnt[jj];

        for(int j = 0; j < csize; ++j)
        {
            T q = p_col[j];
            S a = p_ele[j];
            value[jj-ii] += a * u_data[q];
        }
        v_data[jj] = value[jj-ii];
    }
}

```

Do not work for us! (But the online description of the use of *aligned* was for a loop with inner race condition!.)

6.2.2 SECOND TRY

```

#pragma omp parallel for default(none) private(i) shared(v_data, u_data, s, size, col, ele,
    dsp, cnt)
for(i = 0; i < size; i++)
{
    s = v_data[i];
    const T *__restrict p_col = col + dsp[i];
    const S *__restrict p_ele = ele + dsp[i];
    const T csize = cnt[i];

    for(int j = 0; j < csize; j++)
    {
        T q = p_col[j];
        S a = p_ele[j];
        s += a * u_data[q];
    }
    v_data[i] += s;
}

```

Not efficient! Compiler does by default a better work!

6.2.3 THIRD TRY

```

#pragma omp parallel for
for(int i = 0; i < size; i++)
{
    S s = v_data[i];
    const T *__restrict p_col = col + dsp[i];
    const S *__restrict p_ele = ele + dsp[i];
    const T csize = cnt[i];

```

```

for(int j = 0; j < csize; j++)
{
    T q = p_col[j];
    S a = p_ele[j];
    s += a * u_data[q];
}
v_data[i] += s;
}

```

Is efficient!

7 AMG ON KNL

We try the 3 versions of optimization with the AMG solver.

7.1 FIRST WITHOUT SPETIAL FLAGS

We run the program out of the box:

Timing:

- 👉 • Program Solve : 2.50801s
- Scalar Product : 0.00983429s
- Linear Operator: 0.0603058s

7.2 SECOND WITH HIGH BANDWIDTH MEMORY

We run the program with *numactl --membind 1 ./sandbox.ICC_KNL_genBi_ellmat.bin genBi_rhs_1e*

Timing:

- 👉 • Program Solve : 2.09236s
- Scalar Product : 0.00527334s
- Linear Operator: 0.0439417s

7.3 THIRD WITH VECTORIZATION

We run the program with *-xMIC-AVX512 -DSTREAM_ARRAY_SIZE=64000000* Timing:

- 👉 • Program Solve : 2.03113s
- Scalar Product : 0.00489283s
- Linear Operator: 0.0391483s

7.4 FORTH WITH DEFINITION OF THREADS

We run the program with *KMP_HW_SUBSET=1 and OMP_NUM_THREADS=256* Timing:

- 👉 • Program Solve : 25.3131s
- Scalar Product : 0.276269s

- Linear Operator: 1.70215s

htop shows, we do not use all the cores (only fore, but I do not know why.²)!

8 SHORT FIRST CONCLUTION

I observe, that KNL is very useful for the Conjugate gradient method with Jacobian. But we do not get height rise of efficiency if we use the Algebraic multigrid. Possibly, we should use a larger example.

9 IMPROVEMENT FOR KNL


We follow the Book from [1] to improve our code.

9.1 THREAD PINNING

We set the following environment variables:

- export KMP_PLACE_SUBSET=4T
- export OMP_NUM_THREADS=36

Timing:

- 
- Program Solve : 0.981186s
 - Scalar Product : 0.00470805s
 - Linear Operator: 0.0720277s

That is more than a factor 2 of improvement!

Lets try some different configurations of the variables:

9.1.1 TESTS WITH THE ENVIRONMENT VARIABLES

1.
 - export KMP_PLACE_SUBSET=2T
 - export OMP_NUM_THREADS=16
 - Program Solve : 1.96236s
 - Scalar Product : 0.00728059s
 - Linear Operator: 0.162728s
2.
 - export KMP_PLACE_SUBSET=2T
 - export OMP_NUM_THREADS=32
 - Program Solve : 1.11158s
 - Scalar Product : 0.0049665s
 - Linear Operator: 0.0831602s
3.
 - export KMP_PLACE_SUBSET=2T

²Possibly, since the Chip is splitted into 4 x 64 parts

- `export OMP_NUM_THREADS=64`
 - Program Solve : 0.686413s
 - Scalar Product : 0.00366664s
 - Linear Operator: 0.0411553s
4. • `export KMP_PLACE_SUBSET=2T`
- `export OMP_NUM_THREADS=128`
 - Program Solve : 0.862599s
 - Scalar Product : 0.00404763s
 - Linear Operator: 0.0359199s
5. • `export KMP_PLACE_SUBSET=4T`
- `export OMP_NUM_THREADS=16`
 - Program Solve : 1.90819s
 - Scalar Product : 0.00724268s
 - Linear Operator: 0.159435s
6. • `export KMP_PLACE_SUBSET=4T`
- `export OMP_NUM_THREADS=64`
 - Program Solve : 0.641385s
 - Scalar Product : 0.00371933s
 - Linear Operator: 0.0373378s
7. • `export KMP_PLACE_SUBSET=4T`
- `export OMP_NUM_THREADS=128`
 - Program Solve : 0.889766s
 - Scalar Product : 0.00421667s
 - Linear Operator: 0.0363829s

Therefore for our small problem, we get the best performance with

- `export KMP_PLACE_SUBSET=4T`
- `export OMP_NUM_THREADS=64`

9.1.2 THREAD PINNING WITH JACOBI

I found, that thread pinning with Jacobi has a negative effect!

10 TEST WITH KMP_AFFINITY

We try the run with *export KMP_AFFINITY=compact*.

Jacobi:

- `export KMP_PLACE_SUBSET=4T`
- `export OMP_NUM_THREADS=64`
- Program Solve : 8.57932s
- Scalar Product : 0.0873878s
- Linear Operator: 1.38823s

AMG:

- `export KMP_PLACE_SUBSET=4T`
- `export OMP_NUM_THREADS=64`
- Program Solve : 1.98916s
- Scalar Product : 0.00723767s
- Linear Operator: 0.141118s

This flag is usually to improve the all-to-all communication, which is not efficient in our case. Therefore we get (as to be expected) a slower execution time.

11 SUMMARY OF THIS TECHREPORT

If one use KNL *out of the box* (with OpenMP parallelization), we get (a reference) solution time as 12.5s for CG with Jacobean precondition-er. To use the *High bandwidth memory* (HBM) do not improve an *out of the box* example (only ~ 10 %). Also a simple flag for vectorization (*-xMIC-AVX512*) does not lead to an improvement on KNL.

The use of *on-platform memory* (DDR4) leads to a real improvement. With the environment variable *KMP_PLACE_SUBSET=4T* we get a three times faster code. If we use (additionally) the HBM memory, we get again a speed up of ~20%. We get the best result if we use *KMP_PLACE_SUBSET=1T*, HBM, vectorization with *-xMIC-AVX512* and *-DSTREAM_ARRAY_SIZE=64000000* (wherein the last environment variable fixes the max length of the bandwidth). With this we get the solve time 2.77s (4.5 times faster than out of the box).

If we consider the CG method with AMG preconditioner, we get out of the box a reference solve time as 2.51s. We get the best result is we use the same parameters as for the CG method with Jacobean except on *KMP_PLACE_SUBSET=4T*, with a solution time of 0.64s (3,9 times faster than out of the box).

REFERENCES

- [1] Jim Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming. Knights landing edition*. Elsevier / Morgan kaufmann, 2016.