

PERFORMANCE ANALYSIS OF MKL FAST FOURIER TRANSFORM AND FFT-BASED POISSON SOLVER ON INTEL XEON PHI

Benjamin Ulmer, Uldis Locans and Andreas Adelman

Paul-Scherrer-Institut Villigen, Switzerland

ABSTRACT

This paper presents a short performance analysis of a three dimensional, real-to-complex Fast Fourier Transformation (FFT) implemented using Intel Math Kernel Library. Further performance analysis of an FFT-based Poisson solver is presented as the targeted application. The benchmarks are performed on an Intel Xeon Phi 5110p coprocessor card as an example for Intel many integrated core architecture systems. The benchmark code is implemented in the framework of the Dynamic Kernel Scheduler (DKS), an ongoing research project at PSI, aimed to provide an efficient software layer between host application and different hardware accelerators. Performance results are compared to an implementation for an Nvidia Tesla K20 card as an example for a general purpose GPU serving as an hardware accelerator.

1. INTRODUCTION

The Dynamic Kernel Scheduler (DKS)[1] is being developed in order to provide a software layer between host applications and different hardware accelerators. DKS handles the communication between the host and the hardware accelerators, schedules task execution, and provides a library of built-in algorithms. DKS is used for parallelization in the Object Oriented Parallel Accelerator Library (OPAL) [2], an open source tool for charged-particle optics in accelerator structures and beam lines. An important part of the OPAL framework is the included field solver using real to complex Fast Fourier Transformations. Intel Math Kernel Library (MKL) [3] provides an implementation for discrete Fourier Transformations promising good performance on Intel Many Integrated Core (MIC) architectures. An implementation using MKL is compared to a CUDA implementation for an Nvidia Tesla K20 general purpose GPU. Besides benchmarks using the standalone DKS framework also benchmarks combining DKS with the Independent Parallel Particle Layer (IPPL) [4], an object oriented framework for particle based applications, are presented. The Poisson solver implemented in OPAL [2] is based on Fourier transformations of a Greens function and the charge density field and therefore a promising application to gain speedup by performing FFTs on an accelerator card.

2. PROGRAMMING MODEL

Intel many integrated core architectures (MIC) [5] support two different programming models, the *native mode* and the *offloading model*. In native mode the code is compiled for the accelerator card and the executable is moved to the accelerator and run entirely on the coprocessor. Offload model uses compiler directives to specify code sections which should run on the MIC if it is available. In the present performance analysis the offload model is used, which allows writing code similar to CUDA code used for the GPU as an accelerator.

3. FAST FOURIER TRANSFORMATION ON MIC

As one crucial part of the targeted Poisson Solver an isolated performance analysis of FFT on Intel Xeon Phi is given. After outlining the implementation of the benchmark codes the Xeon Phi performance is discussed and compared to host only execution and a GPU accelerated version.

3.1. Implementation

To benchmark the Fast Fourier Transformations a program flow similar to the target problem of the field solver was used and is listed in Algorithm 1.

FFT(*n_times*) and IFFT(*n_times*) perform as many forward or backward Fourier Transforms on the MIC as given in the parameter *n_times*. Experiments with *n_times* set to one and ten have been performed. As an example for the FFT functions see the code for FFT real-to-complex presented in Listing 1. All offloaded sections are timed individually using Intel offload report. For comparisons with other hardware accelerators the IPPL framework was used, comparing the times needed for a complete cycle of data movement to the accelerator, forward transformation, backward transformation and data movement back to the host. This benchmark is closest to the targeted field solver application. The `copy_data` flag is used to examine the time spent for data movement between host and coprocessor and to allow for additional computations on the host between the forward and backward transformation.

Algorithm 1: Benchmark loop

```
offload: Setup environment on MIC;  
offload: Allocate Memory on MIC;  
set timer = 0;  
for  $i = 1, \dots, 11$  do  
  resume timer;  
  offload: write real data;  
  offload: perform FFT( $n\_times$ );  
  if  $copy\_data == true$  then  
    offload: read complex data;  
    offload: write complex data;  
  offload: perform IFFT( $n\_times$ );  
  offload: read complex data;  
  stop timer;  
  check correctness;  
offload: Free memory on MIC;
```

```
1  int MICFFT::mic_executeRCFFT(void *in_ptr, void  
  *out_ptr, int ndim, int N[3], int n_times)  
3  {  
  // ...  
  #pragma offload target(mic:0)  
  in(real_ptr:length(0) RETAIN REUSE)  
  in(compl_ptr:length(0) RETAIN REUSE)  
5  {  
6    for (int i=0; i<n_times;++i)  
7    {  
8      DftiComputeForward(this->getHandleRC(),  
9      real_ptr, compl_ptr);  
10     }  
11    }  
12    return DKS_SUCCESS;  
13  }
```

Listing 1: FFT for Intel MIC using MKL

3.2. Experimental Results

Experimental setup. Following the benchmark loop in Algorithm 1 the Intel MKL performance was examined on an Intel Xeon Phi 5110p Coprocessor card with the following specifications

- 60 cores @ 1.053 GHz
- 8GB DDR5 @ 5.0 GTtransfers/s
- peak performance: 1.011 Tflops/s

All offloaded sections are timed individually using Intel offload report. Algorithm 1 has been integrated into the IPPL framework [4] to compare with other implementations and hardware accelerators such as a Nvidia Tesla K20 GPU. In the latter case a timer as shown in Algorithm 1 was used to compute the wall-clock-time of the different executions of the program.

The following flags turned out to give the best performance

on Intel MIC, however they have just little influence on the overall performance.

- MIC_ENV_PREFIX=MIC
- MIC_KMP_AFFINITY=scatter,granularity=fine
- MIC_OMP_NUM_THREADS=236
- MIC_USE_2MB_BUFFERS=64K

Operation Count. Performance is evaluated in terms of floating point operations per second (flops/s). Following an Intel tuning guide for two dimensional FFT [6] we estimate the flop count for FFT and IFFT by

$$2.5 \cdot N^3 \cdot \log_2(N^3)$$

where N denotes the degrees of freedom per dimension. All experiments have been restricted to cubic problem sizes with $N \in [8, 16, 32, 64, 128, 256]$.

Results. To put the following results into context the comparison of the different implementations and hardware accelerators can be found in Figure 1. Best performance was

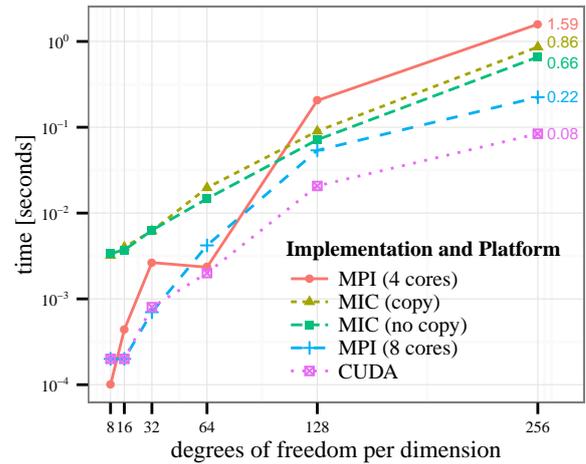


Fig. 1: IPPL benchmarks for Algorithm 1 with $n_times = 1$. MPI 4 and MPI 8 corresponds host only implementations using 4 and 8 cores. The implementations for the accelerators use one core on the host.

obtained using the CUDA implementation without data movement between forward and backward transformation. The MIC implementation without data movement between forward and backward transformation is faster than the host only implementation using 4 cores but slower than using all 8 cores on the host. To examine the relatively poor performance of the Intel MIC all offloaded parts of Algorithm 1 ($copy_data=0$) have been analyzed using Intel offload report. The first run of the benchmark loop shows worse performance than the following executions due to cold caches

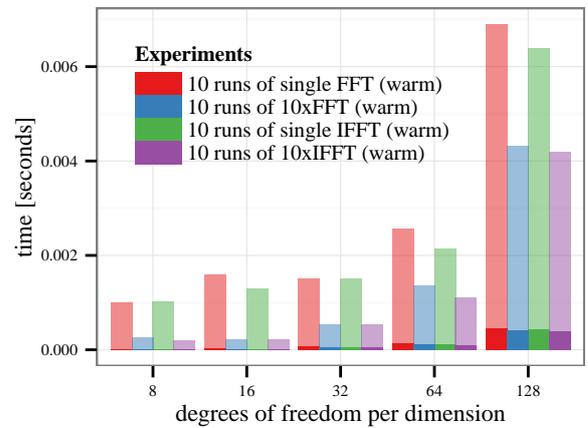
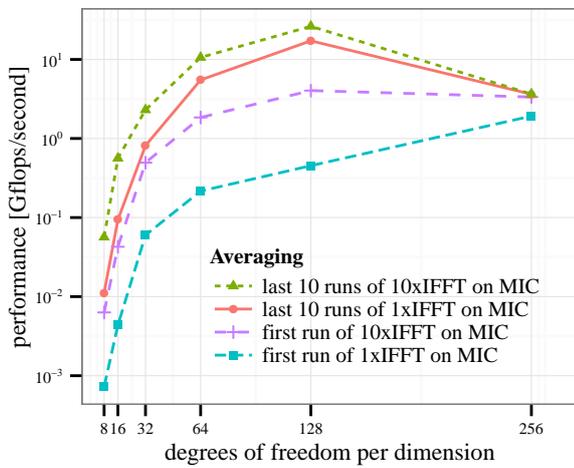
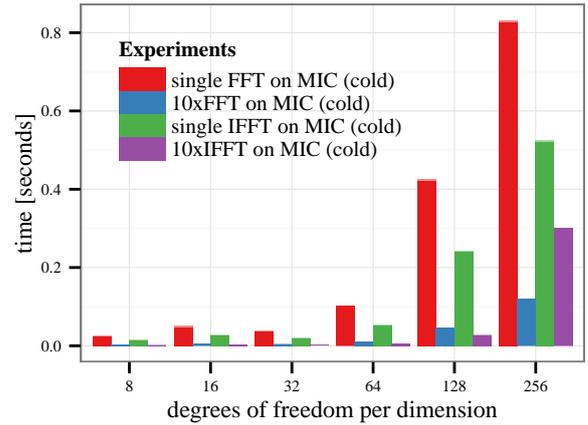
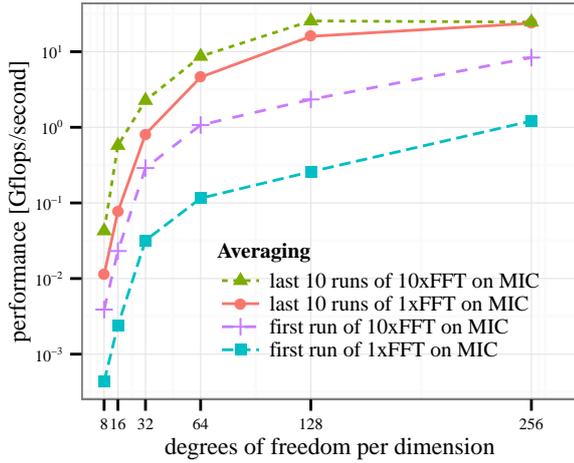


Fig. 2: FFT (top) and IFFT (bottom) performance.

on the coprocessor. Execution times for the "warm-up" run can be seen in Figure 3 (top).

As expected the execution times increase with number of degrees of freedom. Averaging over 10 transformations per FFT call decreases the execution time due to warm caches on the MIC. For problem sizes smaller than 128^3 IFFT is always faster than FFT, however with 256^3 degrees of freedom IFFT becomes significantly slower, which is further examined in the following experiments. In Figure 3 (middle and bottom) the timings averaged over the 10 runs following the warm-up run are presented.

One can observe, that averaging over 10 transformations on the coprocessor decreases runtime significantly due to caching effects. The cpu time is much higher than the actual compute time on the MIC and has to be analyzed in further work. The yet unexplained behavior of inverse transformation being orders magnitudes slower than the forward transformation for problems bigger than 128^3 can obviously be

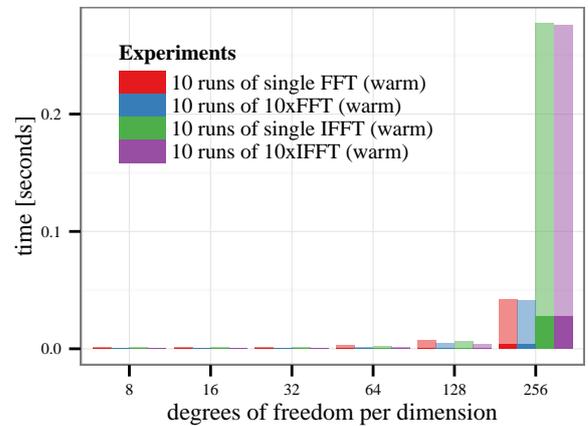


Fig. 3: Average time spent for single transformation in FFT and IFFT routines. Warming up the cache on the coprocessor (**top**). Averaged runtimes for 10 runs of the benchmark loop after the warm-up run for problem sizes up to 128^3 (**middle**) and sizes up to 256^3 (**bottom**). Solid bars correspond to execution time spent on the MIC, transparent bars correspond to host time needed for the corresponding of-fload section.

seen. Using the flop count presented above, Figure 2 show the performance of the forward and backward transformation routines on Intel MIC.

The maximal obtained performance for forward transformation is 25.5 Gflops/s and for backward transformation a maximum of 26.25 Gflops/second could be obtained. This is a factor of 5 smaller than the obtained 103 Gflops/seconds reported in the Intel tuning guide [6] for two dimensional FFT. It has to be considered that a three dimensional transformation as used in this paper needs an additional matrix transposition which takes additional time, while keeping the flop count constant.

4. POISSON SOLVER

The Poisson Solver in OPAL is used to calculate the space-charge forces in a particle beam. The solution of the Poisson equation

$$\nabla^2 \phi = -\rho/\epsilon_0 \quad (1)$$

with scalar potential ϕ and charge density ρ can be obtained by the convolution of the density field ρ with the appropriate (integrated) Green's function G [2]. Convolution can be performed in $\mathcal{O}(n \log n)$ using fast Fourier transforms. The implementation on Intel Xeon Phi is outlined and experimental results are discussed.

4.1. Implementation

The solution of the Poisson equation (1), given an interpolated charge density on a 3D rectangular grid is sketched in in Algorithm 2.

Algorithm 2: Poisson solver

- offload:** Setup environment on MIC;
 - offload:** Allocate Memory on MIC for ρ and G ;
 - set timer = 0;
 - offload:** compute integrated Greens function G on MIC;
 - offload:** FFT: $G \rightarrow \hat{G}$
 - offload:** write discrete charge density field $\rho(i, j, k)$ to MIC;
 - offload:** FFT: $\rho \rightarrow \hat{\rho}$;
 - offload:** multiply complex fields $\hat{\phi} = \hat{\rho} \cdot \hat{G}$
 - offload:** IFFT: $\hat{\phi} \rightarrow \phi$;
 - offload:** read ϕ from MIC;
 - stop timer;
 - offload:** Free memory on MIC;
-

4.2. Experimental Results

Performance of the OPAL Poisson solver was evaluated on the same hardware as for the FFT experiments (see section 3.2) using only one core on the host. As test case the PSI-Ring-Cyclotron [7] was used. In Figure 4 execution time for computing the potential field as shown in Algorithm 2 is presented. The number of particles was set to 10^5 and 100 time steps have been executed. Similar to the FFT experiments performance of MIC is inferior to the CUDA implementation and even slower than the CPU version. It has to be considered that no optimization toward asynchronous execution on the MIC or memory locality was performed. However table 1 comparing the single steps of the potential field computation shows that FFT performance has still the main impact on the overall performance on Intel MIC.

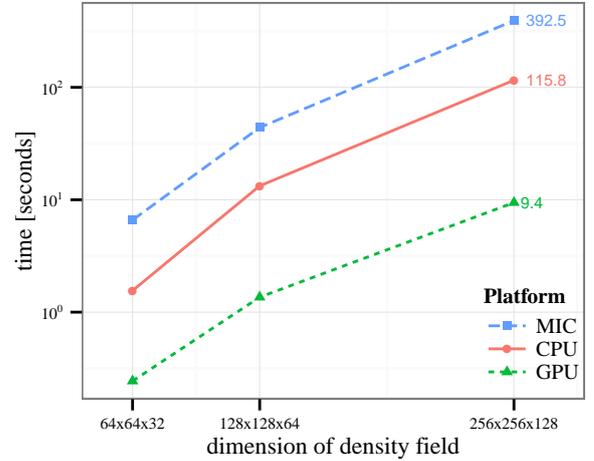


Fig. 4: Compute Potential Performance of Poisson Solver with PSI-Ring-Cyclotron for 10^5 particles and 100 time steps.

| Routine | CPU time | GPU time | MIC time |
|--------------------|----------|----------|----------|
| comp. G | 60.0856 | - | 49.575 |
| FFT+IFFT | 75.5668 | - | 233.552 |
| total comp. ϕ | 115.753 | 9.43198 | 392.54 |

Table 1: Timing in seconds of single steps of potential field calculation on a $256 \times 256 \times 128$ grid.

5. CONCLUSIONS

Intel MKL FFT implementation for MIC architecture seems to be inferior to the CUDA implementation (see Figure 1).

However the unexpected behavior for the inverse transformation on big problem sizes has to be examined further. Relatively long execution times on the host compared to the coprocessor should be evaluated and maybe the native mode for Intel MIC should be taken into account as an alternative. Performance of the Poisson solver can be increased, by optimizing the single steps on the MIC, however the main focus should be on accelerating the FFT computation on the MIC.

6. REFERENCES

- [1] U. Locans, “Dynamic kernel scheduler,” .
- [2] A. Adelmann et al., “Object oriented particle accelerator library,” online: <https://amas.psi.ch/OPAL>.
- [3] Intel(R), “Many integrated core architecture,” .
- [4] A. Adelmann, “Independent parallel particle layer,” online: <http://amas.web.psi.ch/tools/IPPL/index.html>.
- [5] Intel(R), “Intel math kernel library version 11.2,” online: <http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html>.
- [6] Y. Chao, “Tuning the intel mkl dft functions performance on intel(r) xeon phi coprocessors,” online: <https://software.intel.com/en-us/articles/tuning-the-intel-mkl-dft-functions-performance-on-intel-xeon-phi-coprocessors>.
- [7] YJ Bi, A Adelmann, R Dölling, M Humbel, W Joho, M Seidel, and TJ Zhang, “Towards quantitative simulations of high power proton cyclotrons,” *Physical Review Special Topics-Accelerators and Beams*, vol. 14, no. 5, pp. 054402, 2011.

A. MAJOR CODE CHANGES

In this section the crucial parts for using real-complex FFT on Intel MIC will be described.

A.1. Dynamic Kernel Scheduler

Setup FFT for MKL. Intel Math Kernel Library [3] requires to set up the environment for FFT calculations. The setup functions can be found in `DKS/src/MIC/MICBase.cpp`. Different setup functions for real-complex, complex-real etc. are used and stored in several distinguished handles to ensure that each setup has to be performed only once. A crucial part of the real-complex and complex-real setup is the definition of the strides used. These parameters might be tuned to increase data locality. The implemented setup follows the Intel MKL manual on <https://software.intel.com/...>

`//software.intel.com/...`

Compute FFT on Intel MIC. In `DKS/src/MIC/MICFFT.cpp` one can find the functions doing the actual computation of the Fourier transforms on Intel MIC. The handles created in the setup functions described above are reused every time a FFT is computed. The execution functions are called from `DKS/src/DKSBase.cpp` which handles the distinction between the different accelerator cards used.

Test Routines. An implementation for testing and benchmarking real to complex FFT on Intel MIC using DKS can be found in `DKS/test/testFFT3DRC.cpp`. As input parameter the domain size per dimension is requested (only cubic FFTs are tested).

A.2. OPAL

Setup FFT for Intel MIC. The FFT constructor in `OPAL/ippl/src/FFT/FFT.hpp` was augmented with two calls to setup MKL FFT on Intel MIC. Both real to complex as well as complex to real are set up at the same time. Note that the normalization factor was included to the setup for the complex to real transform. **Transform functions.** In `OPAL/ippl/src/FFT/FFT.hpp` one can find two types of transform functions. The `transform(...)` methods copy the data forth and back to the device whereas the `transformDKSRC` and `transformDKSCR` need the pointers to the data on the accelerator as function arguments as well as the DKS object. Only the real data is written to the device in `DKSRC` and the resulting real data is load from the device in `DKSCR`. However, the intermediate results remain on the accelerator.

B. COMPILATION AND EXECUTION ON KRAFTWERK

B.1. DKS

1. ssh to kraftwerk (from PSI domain):


```
[client]$ ssh <user>@kraftwerk.psi.ch
```
2. checkout svn repository for DKS:


```
[kraftwerk]$ svn co svn+ssh://<user>@savannah02.psi.ch/repos/amas/users/adelmann/Ph.D-students/Locans/work/DKS DKS
```
3. source the newest modules for e.g. Intel compiler


```
[kraftwerk]$ source /opt/psi/config/environment.bash
[kraftwerk]$ source /opt/psi/config/profile.bash
```
4. load the following modules


```
$ module use unstable
```

```
$ module load cmake/3.1.3
$ module load intel/15.3
$ module load openmpi/1.8.4
```

5. setup the CUDA environment (needed for DKS to build)

```
load_cuda() {
    PATH=$PATH:/usr/local/cuda-6.5/
    bin
    export PATH
    LD_LIBRARY_PATH=
    $LD_LIBRARY_PATH:/usr/local/cuda-
    6.5/lib64
    export LD_LIBRARY_PATH
    LIBRARY_PATH=$LIBRARY_PATH:/usr
    /local/cuda-6.5/lib64
    export LIBRARY_PATH
    CPATH=$CPATH:/usr/local/cuda
    -6.5/include
    export CPATH
}
```

6. enable in DKS/test/CMakeLists.txt the tests you want to be build

7. compile DKS following DKS/ReadMe.first

```
$ export CXX_COMPILER=mpicxx
$ export CC_COMPILER=mpicc
$ cd DKS
$ export DKS_ROOT=$PWD
$ mkdir build
$ cd build
$ export DKS_BUILD_DIR=$PWD
$ CXX=$CXX_COMPILER CC=$CC_COMPILER
  cmake -DCMAKE_INSTALL_PREFIX=
  $DKS_BUILD_DIR $DKS_ROOT
$ make
$ make install
```

8. test programs can be found in DKS_BUILD_DIR/test use e.g.

```
$ ./testFFT3DRC 100 100 100
```

B.2. OPAL

1. clone the git repository to obtain the opal sources

```
$ git clone <username>@gitorious.
  psi.ch:opal/src.git OPAL
```

2. to use Intel MIC set the following values in /OPAL/CMakeLists.txt and in /OPAL/ippl/CMakeLists.txt

```
OPTION (ENABLE_OPENCL "Enable
  OpenCL" ON)
OPTION (ENABLE_CUDA "Enable CUDA"
  OFF)
OPTION (ENABLE_OPENMP "Enable
  OpenMP + offload" ON)

OPTION (USE_OPENCL "Use OpenCL" OFF
  )
OPTION (USE_CUDA "Use CUDA" OFF)
OPTION (USE_MIC "Use MIC" ON)
```

3. export the path to your build directory of DKS (see above)

```
$ export DKS_PREFIX=~ /DKS/build/
  build
```

4. build OPAL

```
$ load_opal_toolchain_intel15() {
  module purge
  module use unstable

  module load cmake/3.1.3
  module load intel/15.3
  module load openmpi/1.8.4
  module load hdf5/1.8.12
  module load H5hut/1.99.13
  module load trilinos/11.14.3
  module load gsl/1.15
  module load boost/1.58.0
```

```
export PATH=/opt/psi/Programming/
  gcc/4.9.2/bin:$PATH
export LIBRARY_PATH=/opt/psi/
  Programming/gcc/4.9.2/lib64:
  $LIBRARY_PATH
```

```

export LD_LIBRARY_PATH=/opt/psi/
  Programming/gcc/4.9.2/lib64:
  $LD_LIBRARY_PATH
}
$ load_cuda
$ cd OPAL
$ export OPAL_ROOT=$PWD
$ mkdir build
$ cd build
$ CXX=mpicxx cmake $OPAL_ROOT -
  DENABLE_IPPLTESTS=ON
$ make -j8

```

5. The Fourier transforms using IPPL from OPAL and DKS can be tested using OPAL/src/newbuild/ippl/test/FFT by e.g.

```

$ mpirun -np 1 ./TestRCMIC -grid 16
  16 16 -Loop 10

```

6. The Poisson solver can be tested using OPAL/src/newbuild/src by e.g.

```

$ mpirun -np 1 ./opal RingCyclotron
  .in

```

C. ERRORS AND WARNINGS NOT FIXED YET

- when running IPPL test OPAL/src/newbuild/ippl/test/FFT/TestRCMIC

```

mpirun -np 1 ./TestRCMIC -grid 16
  16 16 -Loop 10
[kraftwerk:17230] mca: base:
  component_find: unable to open /
  opt/psi/Compiler/openmpi/1.8.4/
  intel/15.3/lib/openmpi/
  mca_btl_openib: librdmacm.so.1:
  cannot open shared object file:
  No such file or directory (
  ignored)
[kraftwerk:17230] mca: base:
  component_find: unable to open /
  opt/psi/Compiler/openmpi/1.8.4/
  intel/15.3/lib/openmpi/
  mca_mtl_psm: libpsm_infinipath.
  so.1: cannot open shared object
  file: No such file or directory
  (ignored)

```

- in OPAL/ippl/test/FFT/TestRCMIC.cpp icpc compiler conversion error when using

```

RFieldSPStan=real(CFieldPPStan);

```

- during compilation of OPAL using Intel compiler

```

icpc: command line warning #10006:
  ignoring unknown option '-fno-
  tree-vrp'

```

- and several times (compiling OPAL with Intel compiler)

```

skipping incompatible ...

```