Achieve Better Performance with PEAK on XSEDE Resources

Haihang You, Bilel Hadri, Shirley Moore

XSEDE 12
July 18th 2012
Motivations

- **FACTS**
  - ALTD (Automatic Tracking Library Database) ref Fahey, Jones, Hadri, CUG 2010
    - Numerical libraries are one of the most used packages
    - LAPACK library is linked with different package
      - Craypat, Amber, nwchem, Cactus, Abinit, Chromo and qdp
  - Kraken supports optimized version of LibSci (Cray), ACML (AMD) and MKL (Intel) with different compilers (PGI, GNU, Intel, Cray)
  - Architecture-optimized versions of libraries is not always used on Kraken by the users.

- **GOAL**
  - Study Choosing an appropriate library for a given application is essential for achieving good performance.
  - Design a framework to help researchers to determine the choice of library for better performance of their applications.
PEAK framework

- Three numerical libraries have been studied: LibSci (10.4.5), ACML (4.4.0) and Intel MKL (10.2) using dense and random matrix.

- Each numerical library is built with the following compilers: PGI (10.6.0), GNU (4.4.3) and Intel (11.1.038).

- Developed a set of scripts to build executables, submit jobs, gather data and plot results automatically.

- Store the results to build a Knowledge database
  - Function, Data size (vector/matrix),
  - For ScaLAPACK, FFT, need to add other dimensions:
    - number of cores, topology, accuracy

- To provide an API for the users
  - to query the system for suggestions of using certain library with certain compiler for better performance for their scientific applications
PEAK

Input: size, function ...

Output: compiler, library, environment...

INTERFACE

Driver generator → Test driver code → Compiler/library → Executable → Environment → Performance database

Kernel codes → Job Script generator → Job script → Environment → Performance data

Query
Cray LibSci

- Collection of numerical routines tuned for performance on Cray XT/XE systems.

- Most LibSci components contain both single-processor and parallel routines, optimized specifically to make best use of Cray processors and interconnect architectures:
  - BLAS (Basic Linear Algebra Subroutines) -- BLACS (Basic Linear Algebra Communication Subprograms)
  - LAPACK (Linear Algebra routines) -- ScaLAPACK (parallel Linear Algebra routines)
  - FFT (Fast Fourier Transforms) -- FFTW (Fastest FFT in the West)

- Three libraries unique to Cray are:
  - CRAFFT (Cray Adaptive FFT routines) - a library of serial and parallel, single- and double-precision, computing the discrete Fourier transform in 1, 2 or 3 D.
  - IRT (Iterative Refinement Toolkit) - a library of solvers and tools that provides solutions to linear systems using single-precision factorizations while preserving accuracy through mixed-precision iterative refinement.

- Usage
  - module load xt-libsci (loaded on Kraken by default)
  - man intro_libsci
  - http://docs.cray.com
AMD / MKL

• ACML (AMD Core Math Library)
  – LAPACK, BLAS, and extended BLAS (sparse), FFTs (single- and double-precision, real and complex data types).
  – APIs for both Fortran and C
  – http://developer.amd.com/acml.jsp

• MKL (Math Kernel Library)
  – LAPACK, BLAS, and extended BLAS (sparse), FFTs (single- and double-precision, real and complex data types).
  – APIs for both Fortran and C
Linking on Kraken

<table>
<thead>
<tr>
<th>Library</th>
<th>Compiler</th>
<th>Link-flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIBSCI</td>
<td>PGI</td>
<td>Environment by default: compile without adding flags</td>
</tr>
<tr>
<td></td>
<td>GNU</td>
<td>compile without adding flags</td>
</tr>
<tr>
<td></td>
<td>Intel</td>
<td>compile without adding flags</td>
</tr>
<tr>
<td>ACML</td>
<td>PGI</td>
<td>/opt/acml/4.4.0/pgi64_mp/lib/libacml_mp.a --mp</td>
</tr>
<tr>
<td></td>
<td>GNU</td>
<td>/opt/acml/4.4.0/gfortran64_mp/lib/libacml_mp.a --fopenmp</td>
</tr>
<tr>
<td></td>
<td>Intel</td>
<td>/opt/acml/4.4.0/ifort64_mp/lib/libacml_mp.a -openmp -lpthread</td>
</tr>
<tr>
<td>MKL</td>
<td>PGI</td>
<td>-WI,--start-group /opt/intel/Compiler/11.1/038/mkl/lib/em64t/libmkl_intel_lp64.a /opt/intel/Compiler/11.1/038/mkl/lib/em64t/libmkl_pgi_thread.a /opt/intel/Compiler/11.1/038/mkl/lib/em64t/libmkl_core.a -WI,--end-group -mp -lpthread</td>
</tr>
<tr>
<td></td>
<td>GNU</td>
<td>-WI,--start-group /opt/intel/Compiler/11.1/038/mkl/lib/em64t/libmkl_intel_lp64.a /opt/intel/Compiler/11.1/038/mkl/lib/em64t/libmkl_gnu_thread.a /opt/intel/Compiler/11.1/038/mkl/lib/em64t/libmkl_core.a -WI,--end-group -L/opt/intel/Compiler/11.1/038/lib/intel64/ -liomp5</td>
</tr>
<tr>
<td></td>
<td>Intel</td>
<td>-WI,--start-group /opt/intel/Compiler/11.1/038/mkl/lib/em64t/libmkl_intel_lp64.a /opt/intel/Compiler/11.1/038/mkl/lib/em64t/libmkl_intel_thread.a /opt/intel/Compiler/11.1/038/mkl/lib/em64t/libmkl_core.a -WI,--end-group -openmp -lpthread</td>
</tr>
</tbody>
</table>

- Unload module libsci when loading acml and intel and when you change the Prg-Env

- OMP_NUM_THREADS is set by the user at runtime to the number of threads desired.
- Alternatively, when the code is using Intel or the MKL libraries, the environment variable MKL_NUM_THREADS must be set.
Kraken BLAS : Level 1 : scalar vector addition

- Performance drops at about similar size of problems for all libraries as it reaches the limit at each level of memory hierarchy
- Quasi similar performance except for small data size
LibSci library has the DGEMV fastest implementation for small size by far. For larger data size, ACML and Libsci are similar.

MKL gives the worst performance and the drop out of memory before the others libraries.
Kraken BLAS Level 3: Matrix Matrix Multiplication

- ACML library has the fastest implementation for DGEMM, reaching 113 Gflop/s (91% of the theoretical peak). LibSci is 2% slower than ACML.
- Libraries compiled with Intel perform less than 40% of the theoretical peak.
- For very small matrices, MKL is the fastest implementation,
LAPACK – DPOTRF : Cholesky Factorization

- DPOTRF is based mostly on DGEMM. Both MKL and LibSci are the fastest implementation for the small and large matrix size respectively.

- ACML and LibSci built with Intel compiler gives the worst performance
LAPACK – DGELS – QR factorization and Solve

### Key Points

- **ACML** achieved the best performance with PGI and GNU.
- **LibSci** has a behavior of a library not used the optimizing BLAS. Compare to the ACML, LibSci is 3 times slower with PGI and GNU and 10 times slower with Intel.
- **Libsci** has the small block size (NB) and it is set by default to 32, while for MKL, it depends on the matrix size (it varies from 16 to 128).
LAPACK : DSYGV : generalized symmetric-definite eigenproblem

- ACML and MKL gives the fastest implementation.

- LibSci is the slowest implementation and with GNU, the performance does not exceed 0.3 Gflops while ACML reaches 16 Glops.

\[ \sim 6 \text{ times} \]
Test case using LAPACK Functions on Kraken

DFTB is a quantum chemistry molecular dynamic application solving eigenvalues of the Hamiltonian: \( H(R)\Psi = E(R)\Psi \) refs: Zheng, Morokuma, Jakowski, Int. J. Quantum Chem. (2009)

**Algorithm:**

1) Solve electronic Schrödinger equation (with DSYGV) at nuclear configuration until convergence (~10-20 iterations per MD step)
2) calculate forces (gradient of energy)
3) move nuclei classically (Newton Eq. and quantum forces from 2)
4) repeat step 1-3 several thousand of times (typically 10,000 MD steps)
### DFTB results

- Two simulations are considered: small with 121 atoms and a larger one with 363 which makes the matrix size 460 and 1460 respectively.

- Memory bound. Problem scales cubically with number of atoms $O(N^3)$

- Only 6 steps have been done. The default is the LibSci with PGI. Solving the system takes about 80% of the total execution.

<table>
<thead>
<tr>
<th>Library</th>
<th>Compiler</th>
<th>DSYGV</th>
<th>Total Time in sec</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>small</td>
<td>large</td>
<td>small</td>
</tr>
<tr>
<td>LIBSCI</td>
<td>PGI</td>
<td>31.3</td>
<td>1193</td>
<td>38.9</td>
</tr>
<tr>
<td></td>
<td>GNU</td>
<td>205</td>
<td>7034</td>
<td>221</td>
</tr>
<tr>
<td></td>
<td>Intel</td>
<td>42.3</td>
<td>1401</td>
<td>51.2</td>
</tr>
<tr>
<td>ACML</td>
<td>PGI</td>
<td>7.49</td>
<td>137</td>
<td>15.3</td>
</tr>
<tr>
<td></td>
<td>GNU</td>
<td>11.5</td>
<td>177</td>
<td>22.3</td>
</tr>
<tr>
<td></td>
<td>Intel</td>
<td>15.3</td>
<td>275</td>
<td>22.8</td>
</tr>
<tr>
<td>MKL</td>
<td>PGI</td>
<td>11.7</td>
<td>284</td>
<td>21.2</td>
</tr>
<tr>
<td></td>
<td>GNU</td>
<td>15.8</td>
<td>388</td>
<td>25.5</td>
</tr>
<tr>
<td></td>
<td>Intel</td>
<td>16.2</td>
<td>292</td>
<td>23.9</td>
</tr>
</tbody>
</table>

- Performance match the DSYGV Performance. ACML best implementation (reduced to 50% of the overall time in DSYGV) while LibSci with GNU is the worst!

- For Larger size, MKL library improves and reduce the gap from ACML.
Nautilus

- DGEMM using 16 cores
Nautilus (thread memory affinity)

- SGI Altix systems such as Nautilus provide tools for managing memory placements and thread affinity, which are necessary to optimize OpenMP/multithreaded applications:
  - Thread affinity `dplace`
    - tool is used to bind a related set of processes to specific CPUs to prevent process migrations.
    - This improves the performance since it increases the percentage of memory accesses that are local.
    - the option `-x 2` has been used for the Intel MKL to skip placement of second thread (Intel OpenMP jobs use an extra lightweight shepherd thread that is unknown to the user and need not be placed). Besides thread affinity, memory affinity is a paramount factor to achieve better performance.

  - Memory affinity `numactl`
    - Runs processes with a specific NUMA scheduling and/or memory placement policy.
    - Specifying `interleave=nodes` sets the memory interleave policy in round robin fashion on nodes.
Nautilus (environment variable with MKL)

- To get the full potential of Nautilus (a NUMA system), a few environment variables have to be set to avoid performance degradation.
  - Simply setting the number of threads by defining MKL_NUM_THREADS or OMP_NUM_THREADS is not enough.

- When using MKL and the Intel compiler, two more environment variables must be set,
  - The MKL_DYNAMIC
    - Environment variable enables the MKL to dynamically set the number of threads regardless of MKL_NUM_THREADS/OMP_NUM_THREADS values.
    - The default value of MKL_DYNAMIC is TRUE. When MKL_DYNAMIC is FALSE, Intel MKL uses the number of threads set by the user.

  - The KMP_AFFINITY
    - Environment variable provides a thread affinity mechanism for Intel OpenMP programs.
    - The default value of KMP_AFFINITY type is none such that it does not bind OpenMP threads to particular thread contexts;
    - This binding interferes with dplace on the SGI Altix platform and affects performance adversely.
    - KMP_AFFINITY=disabled must be set to prevent the OpenMP runtime library from making any affinity-related system calls.
Nautilus DGEMM with 16 cores

- Performance on 16 cores of DGEMM with MKL-Intel depending on the environment variable options.
Nautilus (numactl)

- Performance on Nautilus with and without numactl option on 64 cores with DGEMM
Best Practices

- Numerical Recipes books DO NOT provide optimized code.
  - (Libraries can be 100x faster).

- Don’t reinvent the wheel.

- Use optimized libraries!

- Don’t forget the environment variables!

- The efficient use of numerical libraries can yield significant performance benefits
  - Should be one of the first things to investigate when optimizing codes
  - The best library implementation often varies depending on the individual routine and possibly even the size of input data
  - READ the manual and/or attend the tutorials/workshops!
Conclusions

<table>
<thead>
<tr>
<th></th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAXPY</td>
<td>MKL/PGI</td>
<td>LibSci/PGI</td>
<td>LibSci/PGI</td>
</tr>
<tr>
<td>DGEMV</td>
<td>LibSci/PGI</td>
<td>LibSci/PGI</td>
<td>ACML/</td>
</tr>
<tr>
<td>DGEMM</td>
<td>LibSci/PGI</td>
<td>ACML/PGI</td>
<td>ACML/PGI</td>
</tr>
<tr>
<td>DGETRF</td>
<td>ACML/PGI</td>
<td>LibSci/PGI</td>
<td>LibSci/PGI</td>
</tr>
<tr>
<td>DGESV</td>
<td>ACML/PGI</td>
<td>LibSci/PGI</td>
<td>LibSci/PGI</td>
</tr>
<tr>
<td>DPOTRF</td>
<td>MKL/PGI</td>
<td>LibSci/PGI</td>
<td>LibSci/PGI</td>
</tr>
<tr>
<td>DGELS</td>
<td>MKL/Intel</td>
<td>ACML/PGI</td>
<td>MKL/PGI</td>
</tr>
<tr>
<td>DGEEV</td>
<td>ACML/PGI</td>
<td>LibSci/PGI</td>
<td>LibSci/PGI</td>
</tr>
<tr>
<td>DSYGV</td>
<td>ACML/PGI</td>
<td>ACML/PGI</td>
<td>MKL/PGI</td>
</tr>
</tbody>
</table>

- Different library implementations have different strong points.
  - LibSci gives generally the fastest implementation (except for DGELS and DSYGV).
  - ACML should be considered to be the safest solution to avoid weak performance.

- The best library implementation often varies depending on the individual routine and the size of input data.

- Experiment with different versions and parameters and find what works for your code.
PEAK Web Interface
PEAK Web Interface - Performance Checkout
Example: DGEEV
Future Work

• Similar performance on other architectures and other vendor libraries

• Expand the work to ScaLAPACK and FFT

• Provide an API for the users in order to query the system for suggestions of using certain library with certain compiler for better performance for their scientific applications
THANKS !