Portable and Productive Performance on Hybrid Systems with libsci_acc

Luiz DeRose
Sr. Principal Engineer
Programming Environments Director
Cray Inc.
What is Cray Libsci_acc?

- Provide basic scientific libraries optimized for hybrid systems
  - Incorporate the existing GPU libraries into Cray libsci

- Independent to, but **fully compatible with OpenACC**

- Multiple use case support
  - Get the base use of accelerators with no code change
  - Get extreme performance of GPU with or without code change

- Provide additional performance and usability

- **Two interfaces**
  - Simple interface
    - **Auto-adaptation**
    - Base performance of GPU with minimal (or no) code change
    - Target for anybody: non-GPU users and non-GPU expert
  
  - Expert interface
    - Advanced performance of the GPU with controls for data movement
    - Target for CUDA, OpenACC, and GPU experts
    - **Does not imply that the expert interfaces are always needed to get great performance**
Why libsci_acc?

- Code modification is required to use existing GPU libraries!

- Several scientific library packages already exist
  - CUBLAS, CUFFT,CUSPARSE (NVIDIA), MAGMA (U Tennessee), CULA (EM Photonics)

- No Compatibility to Legacy APIs
  - cublasDgemm(....)
  - magma_dgetrf( ...)
  - culaDgetrf( ...)
  - Why not dgemm(), dgetrf()?

- Not focused on Fortran API (C/C++)
  - Require CUDA data types, primitives and functions in order to call them

- Performance
Auto-tuning

Cray Autotuning framework has been built to tune BLAS for accelerators

- GPU kernel codes are built using code generator
- Enormous offline auto-tuning is used to build a map of performance to input
- An adaptive library is built from the results of the auto-tuning
- At run-time, your code is mapped to training set of input
- Best kernel for your problem is used
Simple Interface

- Supports the standard API in the original form

- Will perform all GPU dirty-work for you
  - Initialize data structures on GPU
  - Split your problem into a CPU portion and GPU portion
  - Copy data to the GPU memory from CPU memory
  - Perform GPU and CPU operations
  - Copy data back to CPU memory

- Library-heavy codes can use GPUs with no code change

- Is not only a tool for simple usage
  - If you don’t need the data on the GPU afterwards, use the simple interface

- Simple API has automatic adaptation
Adaptation in the Simple Interface

● You can pass either host pointers or device pointers with the simple interface

● A is in host memory
  \[
  \text{dgetrf}(M, N, A, \text{lda}, \text{ipiv}, \&\text{info})
  \]
  ● Performs hybrid operation on GPU
  ● if problem is too small, performs host operation

● Pass Device memory
  \[
  \text{dgetrf}(M, N, d_\text{A}, \text{lda}, \text{ipiv}, \&\text{info})
  \]
  ● Performs hybrid operation on GPU

● BLAS 1 and 2 performs computation local to the data location
  ● CPU-GPU data transfer is too expensive to exploit hybrid execution
Libsci_acc: Simple Interface for BLAS3 and LAPACK

User Application

dgemm_();

where is the data?

Libsci_acc DGEMM_ACC

Large enough?

Libsci_acc Hybrid DGEMM

Libsci DGEMM
Expert Device & CPU Interface

- Device interface gives higher degrees of control
- Allow users to explicitly specify the execution
  - Every routine in libsci has a version with _acc and _cpu suffix
    - e.g. dgetrf_acc, dgetrf_cpu
  - Simple API for device memory and _acc API are the same
Usage - Basics

● Supports Cray and GNU compilers.

● Fortran and C interfaces (column-major assumed)
  ● Load the module craype-accel-nvidia35.
  ● Compile as normal (dynamic libraries used)

● To enable threading in the CPU library, set OMP_NUM_THREADS
  ● E.g. export OMP_NUM_THREADS=16

● Assign 1 single MPI process per node
  ● Multiple processes cannot share the single GPU

● Execute your code as normal
Libsci_acc with OpenACC

- If the code uses OpenACC, it’s possible to use the library with directives
- All data management performed by OpenACC
- Calls the device version of dgemm
- All data is in CPU memory before and after data region

```
!$acc data copy(a,b,c)

!$acc parallel
!Do Something
!$acc end parallel

!$acc host_data use_device(a,b,c)

call dgemm_acc('n','n',m,n,k,&
    alpha,a,lda,&
    b,ldb,beta,c,ldc)

!$acc end host_data
!$acc end data
```
Libsci_acc with OpenACC

- Libsci_acc is a bit smarter than this:

```
!$acc data copy(a,b,c)

!$acc parallel
!Do Something
!$acc end parallel

!$acc host_data use_device(a,b,c)

call dgemm ('n','n',m,n,k,&
    alpha,a,lda,&
    b,ldb,beta,c,ldc)

!$acc end host_data
!$acc end data
```

- Since ‘a,’ ‘b,’ and ‘c’ are device arrays, the library knows it should run on the device.

- So just dgemm is sufficient.
libsci_acc BLAS Routines Available

- **BLAS 3 - Full HYBRID Implementations**
  - [s,d,c,z]GEMM
  - [s,d,c,z]GEMM
  - [s,d,c,z]TRSM
  - [z,c]HEMM
  - [s,d,c,z]SYMM
  - [s,d,c,z]SYRK
  - [z,d]HERK
  - [s,d,c,z]SYR2K
  - [s,d,c,z]TRMM

- **The following are supported without HYBRID implementations because there is no performance advantage**
  - All BLAS 2 Routines
  - All BLAS 1 Routines
libsci_acc LAPACK Routines Available

**Full HYBRID Implementations:**
- [d,z]GETRF (LU Factorization)
- [d,z]POTRF (Cholesky Factorization)
- [d,z]GETRS (System Solver)
- [d,z]POTRS (System Solver)
- [d,z]GESDD* (Generalized Singular Values)
- [d,z]GEBRD (Generalized Bidiagonalization)
- [d,z]GEQRF* (QR Factorization)
- [d,z]GELQF (LQ Factorization)
- [d,z]GEEV (Non-symmetric Eigenvalues)
- DSYEVR* / ZHEEVR* (Hermitian/Symmetric Eigenvalues)
- DSYEV / DSYEVD (Hermitian/Symmetric Eigenvalues)
- ZHEEV / ZHEEVD (Hermitian/Symmetric Eigenvalues)
- DSYGV / ZHEGV (Hermitian/Symmetric Eigenvalue System Solver)

* Include Cray Proprietary Optimizations
Summary

- **Access to libsci_acc routines is simple**
  - No need to explicitly link - Programming Environment drivers (cc, ftn, CC) do this for you
  - Just target the GPU by loading module

- **Can automatically take advantage of threading on CPU**
  - Just set OMP_NUM_THREADS and run

- **Simple interface available to enable hybrid, CPU or GPU execution of a routine depending on where memory pointers reside and problem size**

- **Interface for advanced control is also available**
Case Study: the Himeno Benchmark

- Parallel 3D Poisson equation solver
  - Iterative loop evaluating 19-point stencil
  - Memory intensive, memory bandwidth bound

- Fortran, C, MPI and OpenMP implementations available from http://accc.riken.jp/HPC_e/himenobmt_e.html

- Strong scaling benchmark
  - XL configuration: 1024 x 512 x 512 global volume
  - Expect halo exchanges to become significant
  - Use asynchronous GPU data transfers and kernel launches to help avoid this
Porting Himeno to the Cray XK6

- Several versions tested, with communication implemented in MPI and Fortran coarrays.
- GPU version using OpenACC accelerator directives
  - Total number of accelerator directives: 27
    - plus 18 "end" directives
- Arrays reside permanently on the GPU memory
- Data transfers between host and GPU are:
  - Communication buffers for the halo exchange
  - Control value
- Cray XK6 timings compared to best Cray XE6 results (hybrid MPI/OpenMP)
Himeno performance

- XK6 GPU is about 1.6x faster than XE6
- OpenACC async implementation is ~ 8% faster than OpenACC blocking
CloverLeaf

- 2D hydro code, with several stencil-type operations

- Developed by AWE
  - Using to explore programming models
  - to be released as Open Source to the Mantevo project hosted by Sandia (miniapps)

- Current performance for 87 steps

<table>
<thead>
<tr>
<th>Mesh</th>
<th>CUDA</th>
<th>OpenACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>960x960</td>
<td>2.44</td>
<td>2.03</td>
</tr>
<tr>
<td>3840x3840</td>
<td>37.42</td>
<td>31.77</td>
</tr>
</tbody>
</table>

February 2013
Luiz DeRose - Cray Inc © 2013
GAMESS

- Computational chemistry package suite developed and maintained by the Gordon Group at Iowa State University
  - [http://www.msg.ameslab.gov/gamess/](http://www.msg.ameslab.gov/gamess/)

- **ijk-tuples kernel - Source changes**
  - CUDA - **1800 lines of hand-coded** CUDA
  - OpenACC – approximately **75 directives added** to the original source

- **Performance of ijk-tuples on 16 XK6 Nodes with Fermi**
  - CPU Only (16 ranks per node) 311 Seconds
  - CUDA – 134 seconds
  - OpenACC – 138 seconds
  - **CUDA was only ~3% faster than OpenACC**

- **Performance of ijk-tuples on 16 XK6 Nodes with Kepler**
  - CPU Only (16 ranks per node) 311 Seconds
  - CUDA – 76.6 seconds
  - OpenACC – 68.1 seconds
  - **OpenACC was ~12.5% faster than CUDA !!**
Summary

- Cray provides a high level programming environment for accelerate Computing
  - Fortran, C, and C++ compilers
    - OpenACC directives to drive compiler optimization
    - Compiler optimizations to take advantage of accelerator and multi-core X86 hardware appropriately
  - Cray Reveal
    - Scoping analysis tool to assist user in understanding their code and taking full advantage of SW and HW system
  - Cray Performance Measurement and Analysis toolkit
    - Single tool for GPU and CPU performance analysis with statistics for the whole application
  - Parallel Debugger support with allinea DDT
  - Auto-tuned Scientific Libraries support
    - Getting performance from the system … no assembly required