ACRC How-To: Using Accelerators—GPUs and Co-processors—on BlueCrystal Phase 3

Table of Contents

Introduction

BlueCrsytal phase 3 was designed to support accelerator technologies. These include GPUs (Graphics Processor Units), from manufactures such as NVIDIA and AMD, as well as co-processors, such as Intel's Xeon Phi. At the date of writing, the cluster contains 70 NVIDIA Kepler GPUs, which have a peak processing power of around 70 Tflops. This is a significant computational resource when we compare it to the figure of around 112 Tflops provided by the standard X86 CPUs contained within the system. The number and type of accelerators installed on the cluster is likely to evolve through time.

This document aims to provide you with an overview of the physical make-up of these accelerators, their computational strengths and weaknesses and how you can practically make use of them.

A common feature of all accelerators is that they are comprised of processor cores that are far less complex than the x86 CPUs employed by the host machine. These simpler cores can efficiently perform a narrower range of calculations than those that can be performed on their x86 counterparts. However, because they are simpler, they occupy far less space on a silicon die and so many more of them can be placed on a single silicon chip. This is the strength of the accelerator. If you have the right kind of task, the many processor cores can be set to work in parallel and by working in tandem can complete the required calculations in less time. The details of the different accelerators may vary, but this general principle holds for all of them.

So far so good, but what is the *right kind of task* for an accelerator? If your task involves a loop over many iterations and the calculations in that loop are independent—i.e. the inputs for one iteration are not dependent upon the outputs of another—then there is a good chance that you can profitably employ an accelerator.

However, there are two more key factor that we must also attend to when using an accelerator. Since the compute device is physically separated from the host memory, typically by a PCIe bus, any data that is required for the calculations must first be moved over to the accelerator's own memory and then carried back once all the required computations are complete. The time taken by these data transfers is significant and can upset the unwary by nullify any gains made through concurrent calculations. We must also be sure not to exceed the memory capacity of the accelerator, which is typically far less than the capacity of the host's memory.

The rest of the document is given over to providing tasters of how you can make use of the accelerators installed in BlueCrystal phase 3. The approaches range from using accelerator enabled applications and libraries, through to incorporating compiler directives in your existing code or

writing new code in languages designed to help the programmer express the parallelism in a task.

Applications and Packages

Many packages and applications are becoming accelerator-enabled. Molecular dynamics simulators have been quick to exploit accelerator technologies. Packages such as GROMACS, AMBER and CHARMM, can all make use of GPUs. Applications from other fields that can use GPUs include Abaqus and LS-DYNA, used in structural mechanics, the BUDE protein docking engine, and the quantum chemistry simulator MOLPRO.

A useful list of GPU accelerated applications has been provided by NVIDIA.

Some general purpose applications are also accelerator-enabled, notably MATLAB. Many of MATLAB's built in functions can be run on a GPU. The **gpuArray()** and **gather()** functions are provided to transfer data to and from the accelerator. For more information about using GPUs in MATLAB, see the ACRC *How-To* document for MATLAB.

Libraries

The call to a library function can provide a very useful separation between bespoke, domain specific code and more common, general purpose routines which are involved in performing the overall task. The task specific code need only conform to the interface for a library routine, and the details of routine's implementation can remain opaque. As new compute technologies emerge, libraries can evolve to exploit those devices. If the interface to the library remains constant, however, the domain specific code need not change. Instead it can reap the rewards of adopting the new technologies but *without any code changes*. An example of such a library evolution is the LAPACK linear algebra package. Implementations exist in serial, multi-threaded and accelerator-enabled form.

Examples of libraries that have been accelerator-enabled include; CULA, cuBLAS and MAGMA, which perform linear algebra routines; the Intel Math Kernel Library and AMD's Core Math Library, which provide not only linear algebra, but also FFTs and random number generators; the PETSc solver library; and OpenMM, for molecular simulation.

A list of GPU enabled libraries, also provided by NVIDIA.

Compiler Directives

If you have some source code that you wish to run directly on an accelerator, the addition of compiler directives can be a convenient way to generate a suitable executable with minimal code changes. Compiler directives are particularly useful when working with a language such as Fortran, which has immature bindings to other, accelerator oriented languages such as OpenCL or CUDA.

Version 4.0 of the OpenMP API contains many features which enable the use of accelerators. However the implementation of this standard by popular compilers, such as those from GNU and Intel, are still incomplete.

The OpenACC standard is supported by the PGI compiler, which is available on BlueCrystal phase 3.

Below is an example code, **vecAdd.f90**, (from Oak Ridge National Laboratory), which mixes Fortran90 and OpenACC. Instructions for compilation on BlueCrystal phase 3 and an example submission script are also given.

vecAdd.f90:

```
program main

   ! Size of vectors
   integer :: n = 100000

   ! Input vectors
   real(8),dimension(:),allocatable :: a
   real(8),dimension(:),allocatable :: b
   ! Output vector
   real(8),dimension(:),allocatable :: c

   integer :: i
   real(8) :: sum

   ! Allocate memory for each vector
   allocate(a(n))
   allocate(b(n))
   allocate(c(n))

   ! Initialize content of input vectors, vector a[i] = sin(i)^2 vector b[i] = cos(i)^2
   do i=1,n
      a(i) = sin(i*1D0)*sin(i*1D0)
      b(i) = cos(i*1D0)*cos(i*1D0)
   enddo

   ! Sum component wise and save result into vector c

   !$acc kernels copyin(a(1:n),b(1:n)), copyout(c(1:n))
   do i=1,n
      c(i) = a(i) + b(i)
   enddo
   !$acc end kernels

   ! Sum up vector c and print result divided by n, this should equal 1 within error
   do i=1,n
      sum = sum +  c(i)
   enddo
   sum = sum/n
   print *, 'final result: ', sum

   ! Release memory
   deallocate(a)
   deallocate(b)
   deallocate(c)

end program
```

The two lines of OpenACC compiler directives are highlighted in bold. We can see that the surrounded loop iterations are obligingly independent.

To compile use:

```
module add pgi/64/13.5
pgf90 -acc vecAdd.f90 -o vecAdd.exe
```

and an example submission script:

```
#!/bin/bash

#! Submit job to the GPU queue and include a GPU in the resource request.
#PBS -q gpu
#PBS -l nodes=1:ppn=1:gpus=1

#! change the working directory (default is home directory)
cd $PBS_O_WORKDIR

#! Record some useful job details in the output file
echo Running on host `hostname`
echo Time is `date`
echo Directory is `pwd`
echo PBS job ID is $PBS_JOBID
echo This jobs runs on the following nodes:
echo `cat $PBS_NODEFILE | uniq`

#! Load the cuda and PGI compiler modules.
module add cuda50/toolkit/5.0.35
module add pgi/64/13.5

#! run the example code.
vecAdd.exe
```

However, if you compare the runtime of vecAdd.f90 solely on the host to the time taken when also employing a GPU, you will discover that this simple loop does contain enough work to justify the data transfer overheads and, as a result, the accelerator-enabled code takes longer to run.

Languages

If you are happy to write your own code, designed for use with an accelerator, you stand the best chance of achieving a useful speed-up. The two dominant languages in this area are OpenCL, and CUDA.

An attractive property of OpenCL is that it is highly portable. OpenCL code can be run on host CPUs, GPUs from any vendor and co-processors alike. In contrast, CUDA code can only be run on NVIDIA GPUs.

Introductions to programming in OpenCL and CUDA can be found at:

- HandsOnOpenCL, and
- Introduction to CUDA C

OpenCL has bindings to several languages, including Python via a package called PyOpenCL. The boxes below provide a minimal set of instructions for installing PyOpenCL in your user space and running an example job.

First download, unpack and install the python package:

```
mkdir ~/pyopencl
cd ~/pyopencl
wget https://pypi.python.org/packages/source/p/pyopencl/pyopencl-2013.2.tar.gz
tar -xzf pyopencl-2013.2.tar.gz
cd pyopencl-2013.2
module add languages/python-2.7.5
python setup.py install –user
```

and use the following submission script to submit an example job to a GPU node:

```
#!/bin/bash

#! Submit job to the GPU queue and include a GPU in the resource request.
#PBS -q gpu
#PBS -l nodes=1:ppn=1:gpus=1

#! change the working directory (default is home directory)
cd $PBS_O_WORKDIR

#! Record some useful job details in the output file
echo Running on host `hostname`
echo Time is `date`
echo Directory is `pwd`
echo PBS job ID is $PBS_JOBID
echo This jobs runs on the following nodes:
echo `cat $PBS_NODEFILE | uniq`

#! Load the cuda and python modules.
module add cuda50/toolkit/5.0.35
module add languages/python-2.7.5

#! run the example code.
python examples/benchmark.py
```

where the example script, **benchmark.py**, is provided as part of the PyOpenCL distribution.

The output should look something like:

```
=============================================================
Platform name: NVIDIA CUDA
Platform profile: FULL_PROFILE
Platform vendor: NVIDIA Corporation
Platform version: OpenCL 1.1 CUDA 6.0.1
-------------------------------------------------------------
Device name: Tesla K20m
Device type: GPU
Device memory:  4799 MB
Device max clock speed: 705 MHz
Device compute units: 13
Device max work group size: 1024
Device max work item sizes: [1024, 1024, 64]
Data points: 8388608
Workers: 256
Preferred work group size multiple: 32
Execution time of test: 0.000754048 s
Results OK
```

Summary

In this document, I have attempted to concisely describe; how accelerators are physically constructed as the computational strengths and weaknesses which emerge as a direct consequence. I have given a brief breakdown of the several different ways in which you may use accelerators on BlueCrystal, together with as many useful links as I could find. Finally, I have include several short examples to help you practically get started using accelerators on cluster.