Performance-Portable High-Level Accelerator Programming

Michael Wolfe
The Portland Group, Inc.

ABSTRACT

The OpenMP API provides a portable model for efficient, high level thread-parallel programming across platforms, vendors, operating systems. We are developing a model with the same advantages to address compute accelerators. In this talk, we explore today’s accelerator landscape, along with the perils of current programming methods. We demonstrate why OpenCL, while impressive and important, doesn’t already solve the programming problem. We close with a summary of the PGI Accelerator Model and the closely-related model being developed by the OpenMP Accelerator subcommittee.

1. WHY ACCELERATORS?

An accelerator is additional hardware added to a computer system in order to do some task faster than the computer could do without the accelerator. At one point, hardware floating point units were designed as an accelerator to a single-chip microprocessor. We use accelerators to stay within a power × cost × performance envelope. You can often get two out of those three, but to get all three often requires a hardware accelerator.

A more iconic accelerator would be the attached processors of the 1970s and 1980s, such as array processors from Floating Point Systems. These were designed to attach to a minicomputer, like a Digital VAX, and to deliver the performance of a mainframe at the cost of a mini.

More recently, Clearspeed designed a single chip parallel processor as a compute accelerator for the high performance and embedded market. IBM used a variant of the Cell processor as an accelerator, most notably for the first petaflop system, the Roadrunner. Convey Computer has designed a computer system using Intel microprocessors and a tightly integrated reconfigurable computer accelerator; an interesting feature of the Convey is the accelerator is implemented using FPGAs, though most of the FPGA programming is hidden from all but the most aggressive programmers.

The accelerators most on people’s minds today are high performance GPUs from NVIDIA or AMD. GPUs themselves are really graphics accelerators, relative to doing graphics on the central processors. GPUs as compute accelerators have the distinct advantage that the cost of developing the chip itself is amortized over its whole market, including all the graphics customers. Since the cost of designing a large computer chip can be on the order of a billion dollars, the design must have a large market to make it affordable. Some upcoming processors from Intel and AMD will incorporate GPU capability on the chip itself; we will discuss how that affects the compute accelerator world.

In the next year or so, Intel has promised to deliver a manycore chip, the Knights Corner, to address the same highly parallel, scalable applications that are now being ported to GPUs. The big potential advantage of this chip is it shares most of its instruction set with the Intel x86 processors, so initial code porting may be quite a bit easier.

A key point with accelerators is that we choose them to get performance. If performance were not a goal, we could find another solution that didn’t involve the complexity of designing and programming the accelerator. Since performance is the goal, we have to pay attention to performance when we design our applications and write our programs, and we are going to have to be willing to spend the time to tune these programs.

2. COMMON THEMES

The landscape of compute accelerators has several common architectural themes, even across such diverse designs as GPUs and Intel manycore. First and foremost, the memory hierarchy for a system with an accelerator is both more important and more exposed to the programmer. Accelerators need their own memory interface; accelerators are designed as bandwidth engines, to solve large problems over large datasets, requiring high bandwidth access to that data. This is a very different interface from that used by the host, which is optimized for low latency access and high locality. Today’s GPUs have separate physical graphics memory (GDDR); moving data from the host to the GPU means communicating across the IO bus (PCI express). The prototype Intel manycore, Knights Ferry, is similar, though the manycore chip can share the address space with the host processor. The operating system in use can migrate virtual pages from the host to the manycore memory on demand, but since this migration crosses the IO bus, it is necessarily slow and should be avoided as much as possible. The Convey system is more tightly integrated, but the coprocessor still uses a separate memory controller to disjoint physical
memory for high bandwidth access to large datasets; the application must allocate data in the proper memory space to take advantage of that high bandwidth. Managing the memory hierarchy is the most challenging aspect of making effective use of accelerators, if only because we have less experience doing this at the application level.

Most accelerators use a variation of a manycore design. An NVIDIA GPU has 16 or 32 streaming multiprocessors; an AMD GPU has 10 or 20 SIMD engines; the Knights Corner is described by Intel as "over 50 cores." The notable exception is the Convey coprocessors, which appears as a single vector engine. Along with a manycore design is the corresponding memory hierarchy, either a hardware or software-managed data cache for each core. Effective use of data locality is required to take advantage of the cache, which may be critical for the highest performance.

Within the core, accelerators are implemented using a high degree of SIMD or vector parallelism. GPUs are programmed as sets of scalar cores, but are in fact implemented for SIMD execution. Intel’s manycore design has 512-bit vector registers and instructions, essentially quadrupling the size of the x86 SSE registers and instructions. SIMD or vector instructions allow for the processor to produce more results per clock with more silicon real estate expended on function units, and less on control. Many irregular applications are inherently nonvectorizable, and will lose this significant factor of performance. Along with SIMD or vector instructions are the SIMD or vector registers, the highest level of the memory hierarchy. This is typically well-managed by compilers, so hopefully we don’t have to expose this in the application.

3. PORTABILITY

An important goal for any target computer system is portability, for some definition of portability. High level languages mostly provide source level portability; we can recompile or rebuild our applications from the program source on any new computer system and expect it to work, as long as the new system has appropriate compilers. This has developed over many years with explicit language standards as well as some implicit standards (byte addressability for C programs).

Instruction set and operating system ABI (application binary interface) standards allow us to build a program or program libraries on one system and use them across a wide family of systems. This dates back at least to the IBM System 360, a family of computers that shared a common instruction set. Today we have many implementations of the x86 instruction set from Intel and AMD, and many implementations of the Power instruction set from IBM, and it is common to be able to build a program that will execute correctly across a whole family of systems.

However, functional portability is insufficient in our particular realm. Our reason for using accelerators is performance, so what we want is performance portability. In the best case, we want to be able to write a single program that runs well across a wide range of accelerators, from today’s systems to tomorrow’s and hopefully through to the future.

We as a community have some experience with portable parallel programming. The two most widely used approaches for parallel programming are MPI and OpenMP. MPI is a library allowing parallel programming across distributed memories, and OpenMP is a set of directives to support shared-memory multiprocessor and multicore parallel programming. MPI programs are portable because there are open-source implementations; moreover, vendor implementations are often optimized for the specific transport layer, so users can expect good performance on any target system. OpenMP programs are portable because every major parallel and multicore vendor supports the standard, and good performance is important to those vendors. OpenMP has the advantage that is (almost) preserves sequential equivalence, so it’s usually easy to reason about the meaning of the parallel program.

4. ACCELERATOR PROGRAMMING

Most accelerator programming today is done with special languages to expose the features of the architecture that deliver performance. This can be viewed as a negative (you must rewrite your program) or a positive (you can take advantage of the architecture). Clearspeed added data parallel extensions to C, creating Cn. Convey systems are programmed using a vectorizing compiler and user-defined intrinsics for special operators. NVIDIA GPUs are mostly programmed (for computing) using the CUDA C and CUDA Fortran languages, which separate the data-parallel device code and device data from the host code and data. An industry group has designed OpenCL as a portable analog to CUDA, where data-parallel in OpenCL is very similar to CUDA.

The goal of performance portable programming is for a strategy that will allow a program to run on a single core, a multicore, or a CPU with one of several accelerators, and achieve good performance in each of those, taking advantage of the features of each, without having to write different versions of each algorithm for the different targets. OpenMP already allows this for the first two (single core and multicore) cases. OpenCL claims to provide "a uniform programming environment for software developers to write efficient, portable code for high-performance computer servers, desktop computer systems and handheld devices using a diverse mix of multi-core CPUs, GPUs, Cell-type architectures and other parallel processors such as DSPs." Can either OpenMP or OpenCL solve our problem for accelerators?

4.1 OpenCL?

I claim OpenCL fails to solve the portable parallel programming problem. It’s an important effort, but there are two problems. First, OpenCL is a library attacking a language problem. The constraints placed on the design, that it be a library, make such things as a kernel launch unfortunately clumsy and error prone. On a host, a procedure call to a matrix multiplication routine would take one line:

\[
\text{matrixMul}(C, A, B, N1, N2, N3);
\]

In CUDA, the kernel launch is a procedure call augmented with the launch configuration:

\[
\text{matrixMul}<<<\text{grid, block}>>>(C, A, B, N1, N2, N3);
\]

CUDA is supported by compilers that can parse the language extensions, but OpenCL is not. Hence, OpenCL must expose the details of the kernel launch as function calls, one for each kernel argument and one to launch the kernel:

\[
\text{clSetKernelArg(matrixMul, 0, sizeof(cl_mem), (void*)&(C))};
\]

\[
\text{clSetKernelArg(matrixMul, 1, sizeof(cl_mem), (void*)&(A))};
\]
the processor from IBM. That compiler redefined OpenMP so that example was an OpenMP compiler to program the Cell advantage. those features that give the accelerator its performance ad-
tors, and that allows the program and compiler to optimize that will be used effectively across a wide range of accelera-
that encourages programming with the types of parallelism between memories on the accelerator itself. T o get perfor-
the host and accelerator, and often explicit data movement 
time, including data movement between 
ignores the memory hierarchy entirely, leaving cache local-
timized for multidimensional parallel constructs. OpenMP 
omp ignores the memory hierarchy entirely, leaving cache loca-
ty to the hardware. Accelerators require optimized use of 
memory hierarchy, including data movement between 
and accelerator, and often explicit data movement 
memories on the accelerator itself. To get performance on the 
accelerator, we need a programming strategy 
that encourages programming with the types of parallelism 
that will be used effectively across a wide range of acceler-
ators, and that allows the program and compiler to optimize those features that give the accelerator its performance advantage. There are examples today of using OpenMP to program 
accelerators, but each suffers from a serious deficiency. One example was an OpenMP compiler to program the Cell processor from IBM. That compiler redefined OpenMP so that 
the master thread ran on the PPE, and slave threads ran work-sharing constructs on the SPEs. This is a significant departure from the OpenMP definitions, where the master thread becomes a member of the team executing work-sharing constructs. Another is an academic project compiler 
that compiles OpenMP programs for GPUs. Only a subset of OpenMP is supported, which is understandable, but the compiler also may ignore some directives, such as the par-
allelism mapping, if it thinks the parallelism should be in another direction. The most aggressive example (to date) has been the Intel compiler for the prototype systems using the Knights Ferry accelerator. The few examples shown use standard OpenMP, with a few added directives to tell the compiler to offload some (or all) parallel computations to the accelerator.

5. EFFECTIVE ACCELERATOR PROGRAMMING

The Portland Group, Inc. (PGI) has designed and implemented a set of language directives, in the same spirit as OpenMP directives, for programming accelerators. The directives allow a user to write a program that will run efficiently on today’s accelerators as well as tomorrow’s. We’ve implemented this in our C and Fortran compilers.

The directive language uses block-structured regions. A compute region is used to offload computation to the accelerator; the program fragment in the compute region will be compiled for execution on the accelerator. The model also allows for a runtime test whether to use an accelerator or not, based on either presence or absence of the accelerator, or based on some user condition, such as the size of the loop.

A data region is used to offload data to the accelerator. Within a data region, the data remains on the accelerator, minimizing data traffic between the host and accelerator. Additional directives can be used to optimize how the compiler should map loop parallelism onto the hardware parallelism, and where to synchronize data between the host and the accelerator. We’ve ported several large applications and seen quite good performance, particularly when data regions are used globally.

A very brief example showing the key directives would be a solver with the following structure (shown in Fortran):

```fortran
use adi_module
!$acc data region copy(rhs,u,us,vs,...))
call adi(rhs,u,us,vs,...))
!$acc end data region
module adi_module contains
subroutine adi(rhs,u,us,vs,...))
!$acc reflected(rhs,u,us,vs,...))
call compute_rhs(rhs,...))
call x_solve(rhs,...)
end subroutine

subroutine x_solve(rhs,rhp,...)
!$acc reflected(rhs,rhp,...))
!$acc region
  do i = 1, n_i
do j = 1, n_j
  rhs(i) = rhs(i) - lval(i,j)*rhp(j)
endo enddo
!$acc end region
end subroutine
end module
```

In this example, the subroutines appear in a module to make the interfaces explicit to all callers. The reflected attribute is used for argument arrays to tell the compiler that the
caller must have a data region to transfer those arguments to the device surrounding the call. It also tells the compiler that those arrays may be used in a compute region with no additional data transfers.

5.1 OpenMP Accelerator Directives

The OpenMP language committee has a subgroup working on accelerator directives with the same goals as the PGI Accelerator model. There will be some spelling changes as well as other more substantive differences, based on experiences with the PGI model and other mechanisms. The basic concepts of compute regions with implicit and guided parallelism mapping are retained, though there may be differences in the how data gets moved between the host and device memories. A particular challenge is how to distribute data and share work across multiple accelerators, and how to split work between an accelerator and a multicore host. The latest CUDA release allows more seamless support for multiple GPUs, and OpenCL allows a program to split data and work across multiple devices of different type, including the host processor, though, as mentioned, both are explicit and relatively low level.

6. CONCLUSIONS

Making accelerators mainstream requires that the programming model conforms to modern standards. Rather than developing new languages and requiring wholesale rewrites of large pieces of an application, we are designing a directive-based programming model, and have a working first implementation. Our belief is that our model will allow performance as well as functional portability across a wide family of accelerators, and still allow for efficient execution on a host without accelerators.