Software for Heterogeneous HPC Systems

September 12th, 2014

David Richie (Brown Deer Technology, HPCMP PETT ENS Onsite)
drichie@browndeer technology.com
Acknowledgments and Disclaimers

The following people and projects contributed to material used for this presentation:

- James Ross (Engility Corporation)
- Christopher Kung (Engility Corporation, HPCMP PETTT CEA Onsite)
- Dale Shires and Song Park (ARL/CISD)
- Jamie Infantolino (SMS, ARL Contractor)
- ARL Advanced Computing investigation of heterogeneous HPC systems

*Opinions expressed in this presentation are my own, and may not be those of the DoD HPCMP; no such endorsement is implied.*
Outline of Talk

Overview of HPC Accelerator Programming

• Programming model for HPC accelerators
• Available APIs
• Considerations for a software project

Software Design Issues in Detail

• Fork-join revisited for accelerators
• Mapping the programming model to APIs
• Refactoring
• Optimization
• Frameworks
• Future considerations

*This is not a programming tutorial; syntax may be notional*
My Background

Research into HPC accelerators for last 10 years

- Application areas: electromagnetics, seismic, quantum chemistry, physics, encryption, image processing, ballistic ray-tracing, others
- Devices: FPGAs, DSPs, GPUs, MIC, RISC arrays

Developed software, middleware, and tools for accelerators

- COPRTHR SDK, STDCL, CLETE, CLRPC, compiler tools
- Implemented OpenCL for x86, ARM, and Epiphany RISC arrays

PI for ARL Advanced Computing project

- Heterogeneous computing for large-scale and deployable HPC

AMD/TopCoder OpenCL Innovation Challenge Winner

- Innovative app using APU (CPU+GPU)
- “Ghost Rider” – real-time feature tracking in an FPS interface
Overview of HPC Accelerator Programming
Speedup: Myths, Legends and Realities

“3,456,000x speedup of PostGIS using a GPU”
- Wow (actual claim from MIT researcher)

“Using a GPU we saw a speedup of 100x”
- Boilerplate papers ca. 2010
- Debunked by Intel paper for people needing a tour through reality

Good rule of thumb: work hard see 2x-5x speedup
- Accelerators provide a small boost
- Accelerators do not turn your workstation into a supercomputer

Speedup is actually an ill-defined quantity
- Problem is the quality of CPU reference code
- One person’s optimized code is another person’s starting point

In 2012 Nvidia admitted large speedups were due to bad code
Programmer’s View of the Architecture

Conceptual challenges:
• Distributed memory management
• Asynchronous parallel execution
• Parallel work distribution

Practical challenges:
• Cross-compiler co-design environment/workflow
• Non-standard (unnecessary) programming languages/extensions
• Complex software stack (everything is a work-in-progress)
• Heterogeneous architectures are simply complicated
Programming Model

General Case: 3-Level Parallelism (MPI + X + Y)

- MPI is orthogonal to the problem, it is of no immediate concern
- X+Y nearly always combined into a single two-faceted API, e.g., CUDA
- Still reflects 2 distinct levels of parallelism

Accelerator-specific: Offload of fine-grain parallelism (X+Y)

- Offload is asymmetric parallelism between host CPU and accelerator(s)
- Fine-grain parallelism maps to accelerator device
Where does MPI Fit In?

Impact of MPI

- Level-1 of a 3-Level Parallelism
- Orthogonal to the problem of accelerators
- Same balancing act between communication and compute
- Only special issue might be sharing of PCIe BUS for traffic to/from accelerators and network cards
- Typical (not only) model is to use one MPI process per accelerator

Increase compute density here

Creates communication bottleneck here
Difficult to Program – Why?

General Issue - heterogeneous system inherently complex

• More parameters to balance
• Employ multiple programming/optimization strategies

Device Issue - burden mostly shifted to programmer

• Modern CPU designed to support software (software-centric)
• Accelerator needs software to support hardware (hardware-centric)

Restricted programming environment

• Hyper-specialized architectures –supportable code restricted
• Performance depends critically on this hyper-specialization

Available software support immature

• Vendors push short-term pragmatic programming solutions
• No satisfying programming semantic has emerged
• Sufficient compiler tools simply do not exist
Divergence of HPC Programming

Good to see the community converge on a solution

- After 7 years no clear convincing path exists for software development
- We have pragmatic APIs targeting some or all devices
- Presently we must choose from what is available to use the hardware
- Programmers are accustomed to greater maturity and stability

HPC development using C++/Fortran, MPI, OpenMP (the calm before the storm)

CUDA (Nvidia only)
STDCL (portable)
OpenCL (portable)
OpenMP-4 (portable)
Intel Offload Pragmas (Intel only)
OpenACC (Nvidia focused)

DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.
Laundry List of Significant API Issues

Pthread-style APIs (explicit offload threading)

- CUDA - Nvidia flagship API, only supports Nvidia GPUs
- OpenCL - portable ... and a clumsy verbose API not terribly well designed, serious industry support varies
- STDCL - portable, designed for HPC, not an industry standard and not pushed by vendors

OpenMP-style APIs (implicit pragma markup)

- OpenMP w/Offload extensions - Intel flagship API for MIC, only supports MIC, shows limitations of pragma markup semantics
- OpenACC - supported by PGI, Nvidia owns PGI, was intended to seed OpenMP, and is now at odds with it
- OpenMP-4 - an attempt at portability, but is not same as OpenACC or Intel API, implementation is a work in progress
Considerations for a Software Project

Goals

• Hero project, experimental design exercise with accelerator?
• Porting or building production ready heterogeneous code?
• Timeframe – development time? designed for today or tomorrow?

Features/Capabilities/Design

• Portability – target specific architectures or all architectures?
• Performance – extreme or good enough?
• Where to place the abstraction boundary?
• Some designs can focus so much on future they are doomed to fail

Investment

• Do your resources realistically match your objectives?

These considerations are not new, but they take on greater importance today when considering heterogeneous HPC codes
Why Are Things More Complicated Today?

Heterogeneous is inherently more complicated

• The obvious answer

Software technology you will rely on is immature

• Vendor-driven, work-in-progress development tools and APIs
• Tasks more complicated than compiling optimized C code for a CPU are met with tools that are well behind in terms of maturity

Employing an accelerator can slow your code down

• There is a special burden to do things right since failure is measurable
• Accelerators require, and typically receive, more optimization effort

Your source code base must fragment, only decision is where

- Code main branch
  - CPU
  - Nvidia GPU
  - Intel MIC

DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.
Impact of Overall Code Structure

Code Structure can drive complexity for heterogeneous systems

- Heterogeneous software must contain code multiplicity, e.g., code for the GPU, code for Phi, etc.
- Model variations outside common solver limit multiplicity
- Model variations within solver increases multiplicity

Example: FDTD code

Example: MD code
Software Design Issues in Detail
Offload-Fork-Join Model for Accelerators

Fork-Join model used to describe parallelism of OpenMP

- Perfect for even simple cases using multi-core CPU

Offload-fork-join model has many constraints for performance:

- $t_c >> t_1 + t_2 + t_3 + t_4$
- $n$ must be large, $m$ well-chosen for good utilization of compute device
- Fine-grain parallelism in your code is necessary but not sufficient for the use of an accelerator

**Simple Fork-Join Model**

```
\[ n \]
```

**Offload-Fork-Join Model for Accelerators**

- Host (CPU) thread
- Copy to device ($t_1$)
- Launch threads ($t_2$)
- Compute ($t_c$)
- Shutdown threads ($t_3$)
- Copy from device ($t_4$)

$n$ threads grouped into blocks of size $m$
Explicit Offload Threading APIs

Conceptually Pthreads for a co-processor

- Examples: CUDA, OpenCL, STDCL

Advantages

- Precise control over memory, host CPU and accelerator execution

Disadvantages

- Co-design workflow, mixes discontinuous language support

```c
for( i=0; i<n; i++ ) {
    c[i] = a[i] * b[i];
}
```

```c
my_kern(float* a, float* b, float* c) {
    i = get_thread_id();
    c[i] = a[i] * b[i];
}
```

```c
device float *a, *b, *c; // device_malloc(n)
copy_to_device(a);
copy_to_device(b);
offload_threads( my_kern, n);
copy_from_device(a);
```
Pragma Markup APIs

Conceptually similar to OpenMP

- Examples: OpenACC, Intel Offload pragmas, OpenMP-4, HMPP

Advantages

- Single source development
- Extends an existing API “transparently”

Disadvantages

- The apparent simplicity does not actually work in practice
- The markup syntax becomes an interlaced language

```c
for( i=0; i<n; i++ ) {
    c[i] = a[i] * b[i];
}
```

```c
#pragma offload target(device) in(a: length(n)) \ in(b: length(n)) out(c: length(n))
{
    #pragma parallel for
    for( i=0; i<n; i++ ) {
        c[i] = a[i] * b[i];
    }
}
```
Which Approach is “Better”?

```c
#pragma offload target(device) \
in(a: length(n) ALLOC RETAIN) \
in(b: length(n) ALLOC RETAIN) \
nocopy(c: length(n) ALLOC RETAIN)
{ /* empty scope inserted for 
distributed memory management */ }
...
#pragma offload target(device) \
nocopy(a: length(n) REUSE RETAIN) \
nocopy(b: length(n) REUSE RETAIN) \
nocopy(c: length(n) REUSE RETAIN)
{
    #pragma parallel for
    for( i=0; i<n; i++ ) { c[i] = a[i] * b[i]; }
}
...
#pragma offload target(device) \
nocopy(a: length(n) REUSE FREE) \
nocopy(b: length(n) REUSE FREE) \
out(c: length(n) REUSE FREE)
{ /* empty scope inserted for 
distributed memory management */ }
```

```c
device_malloc(a,n);
device_malloc(b,n);
device_malloc(c,n);

copy_to_device(a);
copy_to_device(b);

offload_threads( n, my_kern, a,b,c);

my_kern(float* a, float* b, float* b) {
    i = get_thread_id();
    c[i] = a[i] * b[i];
}

copy_from_device(c);

device_free(a);
device_free(b);
device_free(c);

... And did I mention __declspec(target(mic)) ?
```
Finding Parallel Work

Where to find parallel work for accelerators in existing code?

• MPI? – No, not typically
• OpenMP? – Yes, good starting point
• Refactored code? – Yes, using accelerators is not effortless

Can we just recycle OpenMP code? ... No.

• OpenMP programming model not sufficient for accelerators

Compared with multi-core CPUs, parallelism for accelerators ...

• Must be “increased, localized and flattened”
• Must outweigh substantially higher overhead
• Must be re-coded to compensate for weak language support
Code Refactoring Goals

Restructure code to:

- Concentrate and optimize parallelism for offload
- Compensate for primitive compilation model
- Insert distributed memory management operations

Re-code parallel regions to:

- Conform to limited programming language support available
- Prepare for device-specific optimizations that will be required

Restructure data layout to:

- Insert shadow memory allocations if necessary
- Simplify data layout for accelerators
- Optimize data layout for accelerators
Code Refactoring Strategy

Exploit asymmetry in capability of CPU vs. accelerator

- Create CPU version in an “accelerator style”
- Nearly everything can be “faked” and even instrumented for study
- Then drop-in real offload code for accelerator
- Have CPU work through every step of execution you intend to target for accelerator in order to verify correctness
- Try to heavily leverage the CPU during accelerator development since it provides a much better environment in which to work

Original CPU code → GPU-style CPU code → New GPU code

Do not use this version as a reference for reporting speedup
Refactoring Scenarios

Flatten nested parallelism

• Generally not supported or not efficient

Challenges

• Structure may spread out over multiple files
• Dynamic memory allocation within outer loop must be moved outside; larger allocation required could be a problem
• Example of larger issue with concurrency – widening parallelism can require increased memory use

move temporary storage allocation, can increase memory use
Refactoring Scenarios

Loop fusion/fission to concentrate parallelism

- Increase work per thread to mitigate overhead

Challenges

- Can get carried away, fusing too much work into a single thread can be as bad as not having enough

Code here can be a problem and require non-trivial code movement when refactoring (for example, memory allocation, “heavy” serial calls)
Refactoring Scenarios

Separate memory management from parallel work

- Moving data immediately before and after the parallel offload is generally not efficient and leads to unnecessary thrashing
- This is where naive OpenMP markup can meet reality
- Also the problem with trying to hide complexity in libraries

Challenges

- Deciding when data should move, and correctly implementing the movement can be complicated
- Optimum data movement might be into another source file far removed from computation, adding to the complexity
Refactoring Scenarios

Eliminate call indirection

• Unlikely to be supported or efficient, compilation model limited

Challenges

• Code expansion beyond what is supportable by accelerator
• Moving *everything* associated with the call can be problematic since there can be no back-references to the CPU host process
• Code is modular for a reason, undermines good design

```c
for i
    call foo()
    call bar()
```

```c
for i
    inline foo()
    inline bar()
```
Optimization

General Issues

• Use of an accelerator implies performance matters
• Places a greater than normal emphasis and burden on programmer
• Accelerators provide performance at great expense of effort
• Performance is fragile and volatile, significant disparity between optimized and unoptimized code
• Performance is not portable even for portable APIs

Programmer psychology

• Accelerator will receive greater investment of effort for optimization to “prove” the initial assumption: accelerators speedup code
• How many papers are written discussing the speedup of CPU code?
• Primary source of exaggerated speedup claims
• Any team investigating performance advantage of accelerators should have an expert in CPU optimizations
Optimization: Tuning and Coding

Two distinct issues for optimization

• Coding – algorithms must be recoded to match the hyper-specialized accelerator architecture
• Tuning – many run-time parameters must be set just right to get good performance
• And the good news - they are interrelated(!)

Coding

• By definition coding optimizations are target specific
• Vendors can and do undermine your efforts with each software or architecture revision

Tuning

• Generally deals with parameters for mapping parallelism
• Normally has nothing to do with the computation itself
Optimization of Offload-Fork-Join Model

Parallel distribution must be optimized for the accelerator

- Balance competing factors
- Increasing work per thread ($t_c$) reduces parallelism ($n$)
- Thread launch overhead ($t_2$) has fixed and scalable cost
- Optimum balance of the number of threads and work per thread will be different for each accelerator
Optimization: ARL Autotuning Experiments

Compare autotuned kernels with DOE SHOC benchmark

- Uses *parameterized kernels* with brute force sweeps

- Kernels written for Tesla are improved
- In some cases faster than CUDA
- Kernels for Phi are significantly improved
- Intel optimized code using offload pragmas still untouchable
- Issues is weakness in Intel OpenCL implementation
Portability Revisited

Two arguments:

• Portable APIs promote consistency if not portability of code; host-side code (level-2 parallelism) can be mostly portable even if kernels must be specialized

• Portable APIs are an illusion; since performance requires architecture-specific optimization, why not use whatever API is pushed by vendor and build very high abstraction layers

Which argument is right?

• Unclear – this is an issue to be worked out by developers

• Absent a truly good API, the decision is challenging and today everything is in flux
Frameworks

Have no fear, just use a “framework”

- Frameworks are often dreamed up to solve everything and in the end solve nothing – so care should be used
- Good way to abstract parallelism complexity and drive details and API uncertainty to low-level layers
- Keep specific API-of-the-month away from your larger code base

A framework is a way to hedge against the future when developing software for heterogeneous systems

- However its complicated
- Similar to the problems encountered when trying to push accelerator support into libraries, which does not work
- It would be good to have a stable semantically rich way of programming heterogeneous systems, but that is not reality today
ARL Design for Tactical HPC Architectures

Software stack designed for portability across diverse architectures
- We do not port our code to different architectures, its designed to be portable

Abstraction without compromising performance
- High-level code is extensible and flexible
- Resource allocation is handled by the framework transparently
- Optimized computational algorithms are isolated at a lower level

XML-driven C++ Builder Pattern
- Flexible scenario construction

High-Level C++ Framework
- Resource allocation/management

Abstract Evaluator Objects: Host + Kernel Code
- Compute tasks isolated to this layer

STDCL
- Compute API for HPC applications

OpenCL
- Portable low-level API for devices

CLRPC
- Support for networked devices

X86 CPU
- Targeted compute devices

ARM CPU

RISC Array

Xeon Phi

AMD GPUs

Nvidia GPUs
Future Considerations – What If ...

What if a good API emerges?
  • Why this might eventually happen?

What if architecture dominance shifts? (Remember SGI?)
  • What happens to all your specialized code
  • Entry of Phi shows how serious this issue is – all of the CUDA
code is useless for Phi; with the release of a single product the
landscape was upended

What if equivalent of modern optimizing compilers
emerges for heterogeneous platforms (super-compilers)?
  • How badly deprecated will the code we must write now become?

Can HPC developers hedge against the future during a
volatile time of change?
  • Is it safe to write a lot of code? What will be the utility over the
long term? How can abstraction be used effectively?
Summary

• Heterogeneous HPC systems bring unprecedented challenges for software development.

• Present support for heterogeneous systems remains immature by comparison to support for traditional monolithic HPC systems.

• A clear path, in terms of programming semantics and APIs, has yet to emerge; instead developers are provided with pragmatic vendor solutions.

• Even with a good API (well-designed, stable, mature implementations) significant software design challenges would remain due to the many parameters that must be balanced for heterogeneous systems.

• The challenge for software development today is designing large-scale applications with appropriate abstractions to support code longevity while simultaneously supporting architecture-specific optimizations that are critical for performance.