Model for Programming Heterogeneous Systems

David Kunzman  July 28th, 2009

SAAHPC 2009 - NCSA
Motivations

• Popularity of Accelerators
  – Our work on accelerated entry methods currently focuses on Cell and Larrabee
  – Difficulty in programming systems including these devices
    • Extra architecturally specific code
    • Many asynchronous events (DMAs, multiple cores)

• Heterogeneous Systems
  – Roadrunner at LANL (Opterons and Cells)
  – Lincoln at NCSA (Xeons and GPUs)
  – MariCel at BSC (Powers and Cells)
Charm++

• Charm++: An object-based message passing programming model (library based)
  – *Chare objects* (C++ objects) with *entry methods* (member functions)
  – *Entry methods*…
    • are invoked by other entry methods, regardless of where are objects are located
    • Typically access message and object data
  – Constructor of *main chare object(s)* start computation (as main function does in C++)
  – Data locality expressed through chare objects and messages
  – *Interface files* indicate which classes are chare classes and which member functions are entry methods
  – Object collections (*chare arrays*, *groups*, etc.)
Charm++

Chare A
void entryMethod_1() {
    doSomeWork();
    MyMessage msg = new MyMessage();
    B.entryMethod_2(msg); // returns immediately
doMoreWork();
}
void entryMethod_3(int var1, float var2) {...}

Chare B
void entryMethod_2(MyMessage *msg) {
    delete msg;
    int myInt = 4;
    float myFloat = 3.14f;
    A.entryMethod_3(myInt, myFloat);
}

Chare C[0]
Chare C[1]
Chare C[2]
Chare C[3]
Chare C[4]
Chare C[5]
Extensions Charm++

• Added extensions
  – Accelerated entry methods
  – Accelerated blocks
  – SIMD instruction abstraction

• Extensions should be portable between architectures
Accelerated Entry Methods

- Executed on accelerator if present
- Targets computationally intensive code
- Structure based on standard entry methods
  - Data dependencies expressed via messages
  - Code is self-contained
- Managed by the runtime system
  - DMAs automatically overlapped with work on the SPEs
  - Scheduled (based on data dependencies: messages, objects)
  - Multiple independently written portions of code share the same SPE (link to multiple accelerated libraries)
Accel Entry Method Structure

entry [accel] void entryName
  ( …passed parameters… )
  [ …local parameters… ]
  { … function body … }
callback_member_function;

objProxy.entryName( … passed parameters … )
Accelerated Blocks

- Additional code that is accessible to accelerated entry methods
  - `#include` directives
  - `#define` macros
  - Functions called by accelerated entry methods
SIMD Abstraction

• Abstract SIMD instructions supported by multiple architectures
  – Currently adding support for: SSE (x86), AltiVec/VMX (PowerPC; PPE), SIMD instructions on SPEs
  – Generic C implementation when no direct architectural support is present
  – Types: vecf, veclf, veci, etc.
  – Operations: vaddf, vmulf, vsqr.rf, etc.
// Accumulate the incoming floating point values into the local array of values
entry [accel] void ChareObj::accum(int inArrayLen, align(sizeof(vecf)) float inArray[inArrayLen])
  [ readOnly : int localArrayLen <impl_obj->arrayLen>,
    readWrite : float localArray[localArrayLen] <impl_obj->array> ] {

  if (inArrayLen != localArrayLen) return; // Make sure arrays are the same length
  vecf* inArrayVec = (vecf*)inArray; // Cast float arrays to vector float arrays
  vecf* localArrayVec = (vecf*)localArray;
  int arrayVecLen = inArrayLen / vecf_numElems; // Calc len of vector arrays (int “/” rounds down)

  // Add as many elements using SIMD operations as possible
  for (int i = 0; i < arrayVecLen; ++i) {
    localArrayVec[i] = vaddf(localArrayVec[i], inArrayVec[i]);
  }

  // Add remaining elements via scalar operations (if array length is not a multiple of vector length)
  for (int i = arrayVecLen * vecf_numElems; i < inArrayLen; ++i) {
    localArray[i] = localArray[i] + inArray[i];
  }

} accum_callback; // Call impl_obj->accum_callback() when accel entry method completes

To Invoke: myChareObj.accum(someFloatArray_len, someFloatArray_ptr);
MD Example Code

• Based on object interaction seen in NAMD’s nonbonded electrostatic force computation (fairly simplified)
  – Object Types: Patch, SelfCompute, PairCompute, PatchProxy
  – PatchProxy objects are a communication optimization in NAMD
• List of particles evenly divided into equal sized *patches*
  – Compute objects calculate forces
    • Coulomb’s Law
    • Single precision floating-point
  – Patches sum forces and update particle data
  – All particles interact with all other particles each timestep
• Makes use of extensions presented
• ~92K particles (similar to ApoA1 benchmark)
Accelerated Entry Methods

- Particle example program on Cell
  - SPE “kernel,” inner-most loop (4 particle interactions)
    - 124 Flops in 56 cycles via 54 instructions
    - Serial code max. perf: 2.21 flops/cycle or 27.7% peak
  - Performance (measured)
    - 1 Cell (QS20; 8 SPEs) : 50.09 GFlop/sec (24.46% peak)
    - 1 Cell (PS3; 6 SPEs) : 36.66 GFlop/sec (23.87% peak)**
    - 1 x86 core (Xeon E5320) : 8.72 GFlop/sec (58.56% peak)
    - 6 x86 cores (Xeon E5320) : 51.42 GFlop/sec (57.53% peak)
    - 2 Cells (QS20 & PS3) : 87.23 GFlop/sec (24.34% peak)***
  - ** Based off 2 PS3 case (1 PS3 has issues with problem size used)
  - *** Static load balancing used to account for chip differences; non-hetero build
- 8 SPEs (simple cores) ~ 6 x86 cores (complex cores)
Stepping into the Heterogeneous...
Stepping into the Heterogeneous...

Why build this...
Stepping into the Heterogeneous...

... when you could build this?
Work Since the Submission

• A Step Further: A Heterogeneous Cluster
  – Our test cluster is a mixture of
    • 4 QS20 IBM Cell Blades
    • 4 Sony Playstation 3s
    • 1 x86-based node: dual core Intel Xeon
  – Interconnect: Gigabit Ethernet (common to all)
• Goal: Use them all in a single run
  – Runtime system automatically modifies passed parameters to handle architectural differences (e.g. big endian vs. little endian)
Summary of Performance

# Blade/PS3 Pairs (+2 Xeon 5130 cores at each point)
Heterogeneous & Projections

hosts (3 timesteps)

blade_1
ps3_1
blade_2
ps3_2
blade_3
ps3_3
blade_4
ps3_4

blade_1
ps3_1
blade_2
ps3_2
Summary

• Extended Charm++ to support accelerators
  – Accelerated entry methods
  – Accelerated blocks
  – SIMD instruction abstraction
• Modified runtime system to support heterogeneous clusters
  – Removes the requirement that host cores be identical (in terms of architecture and/or resources)
• Demonstrated good performance for a simple MD code on a heterogeneous cluster where…
  – Cores have different ISAs, memory hierarchies, SIMD instruction extensions
  – Some cores (the SPEs) require DMAs, cannot directly communicate on the network
Image Credits

• Background Playstation controller image
  – Originally taken by user “wlodi” on Flickr and modified by David Kunzman
  – http://creativecommons.org/licenses/by-sa/2.0/deed.en

• UN Building
  – Originally taken by user “hmerinomx” on Flickr
  – http://www.flickr.com/photos/tukatuka/3702387380
  – http://creativecommons.org/licenses/by/2.0/

• Ray and Maria Stata Center at MIT
  – Originally taken by user “Christopher Chan” on Flickr
  – http://www.flickr.com/photos/chanc/374392584/
  – http://creativecommons.org/licenses/by-nc-nd/2.0/deed.en