Charm++ on Heterogeneous Systems

Lukasz Wesolowski
David Kunzman

PARALLEL PROGRAMMING LAB
ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Outline

- Motivation
- Heterogeneity
- Current Work
  - IBM Cell Processor
  - NVIDIA GPU
- Future Work

Motivation

- Heterogeneous processors
  - e.g. Cell Broadband Engine Architecture (“Cell”)
  - e.g. Heterogenous CPUs: Intel and AMD to release CPUs with on die GPUs

- Heterogeneous systems
  - Bull Novascale: Hybrid CPU/GPU system
  - Intel & Cray: Using Intel’s Larrabee chips in supercomputers

- Accelerators
  - GPUs, SPEs, FPGAs, and so on being used to perform compute intensive portions of applications
  - Architecture of the accelerator can differ greatly from that of the host CPU
  - Programmer has to keep many architectural details of the accelerator in mind

GPUs & PS3s: Not Just For Gaming

- Goals
  - Allow Charm++ applications to utilize accelerators (GPUs, SPEs on the Cell, etc.)
  - Make this as easy as possible
    - Hide as many architectural details as possible
    - Allow developer to focus on the problem itself

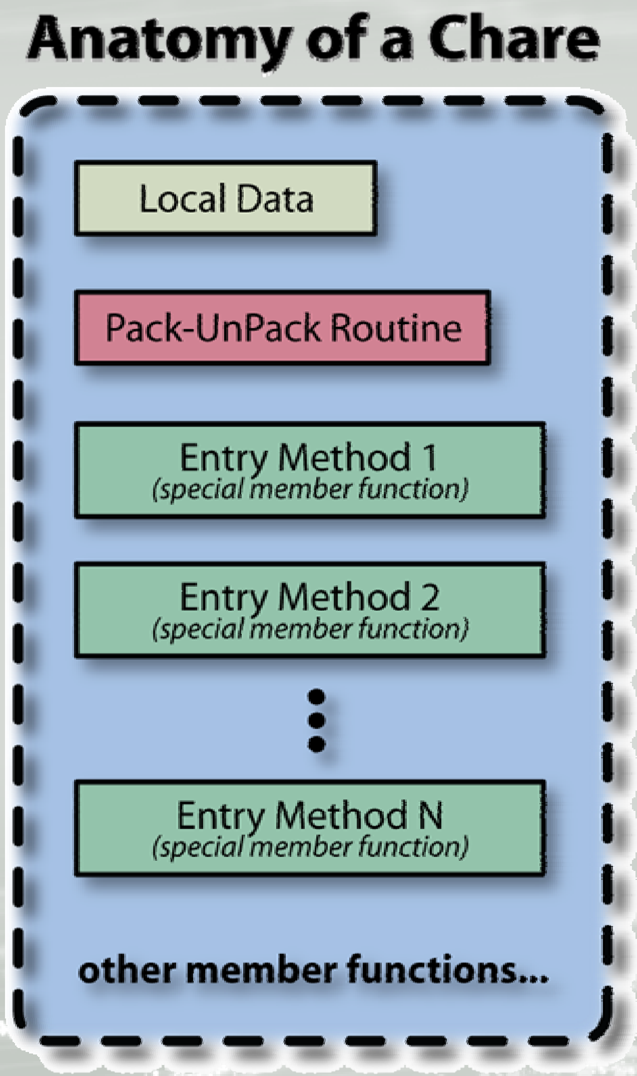
Forms of Heterogeneity

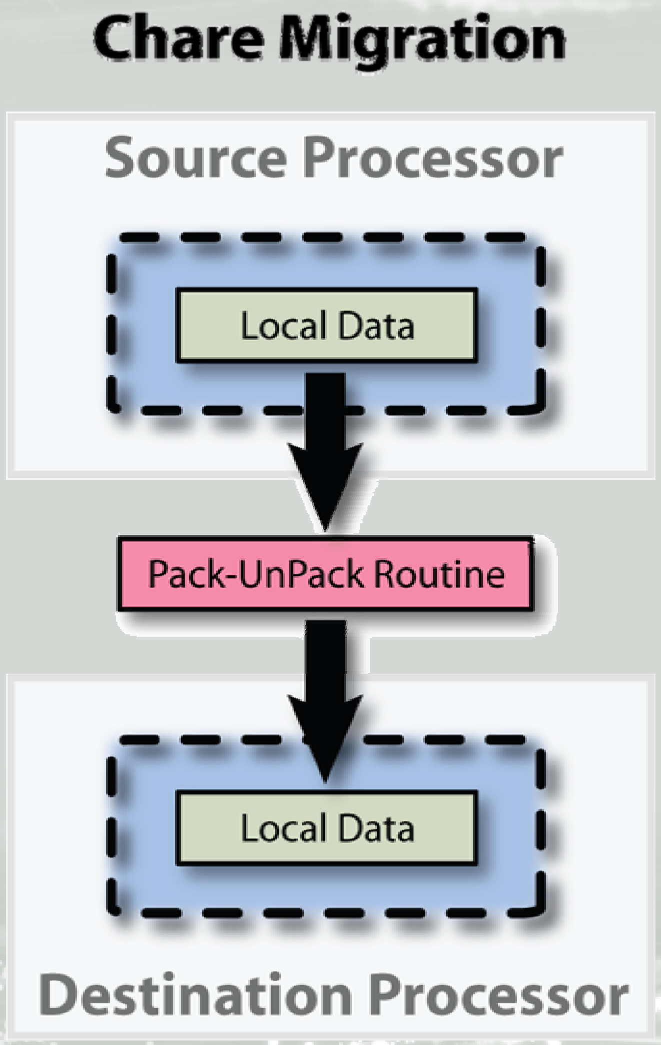
- Architectural differences between cores within a node
  - e.g. PPEs and SPEs on Cell, GPU attached to x86 CPU
  - Behavior/performance depends on which core is used

- Differences in node capabilities/resources
  - e.g. System memory, number of accelerators per CPU

- Nodes have different architectures altogether
  - e.g. Some nodes have x86 processors while others have Cells
  - Performance characteristics of each node can vary dramatically
  - Issues such as byte order as nodes communicate

Chare Objects


Anatomy of a Chare
Local Data
Pack-UnPack Routine
Entry Method 1
(special member function)
Entry Method 2
(special member function)
Entry Method N
(special member function)
other member functions...



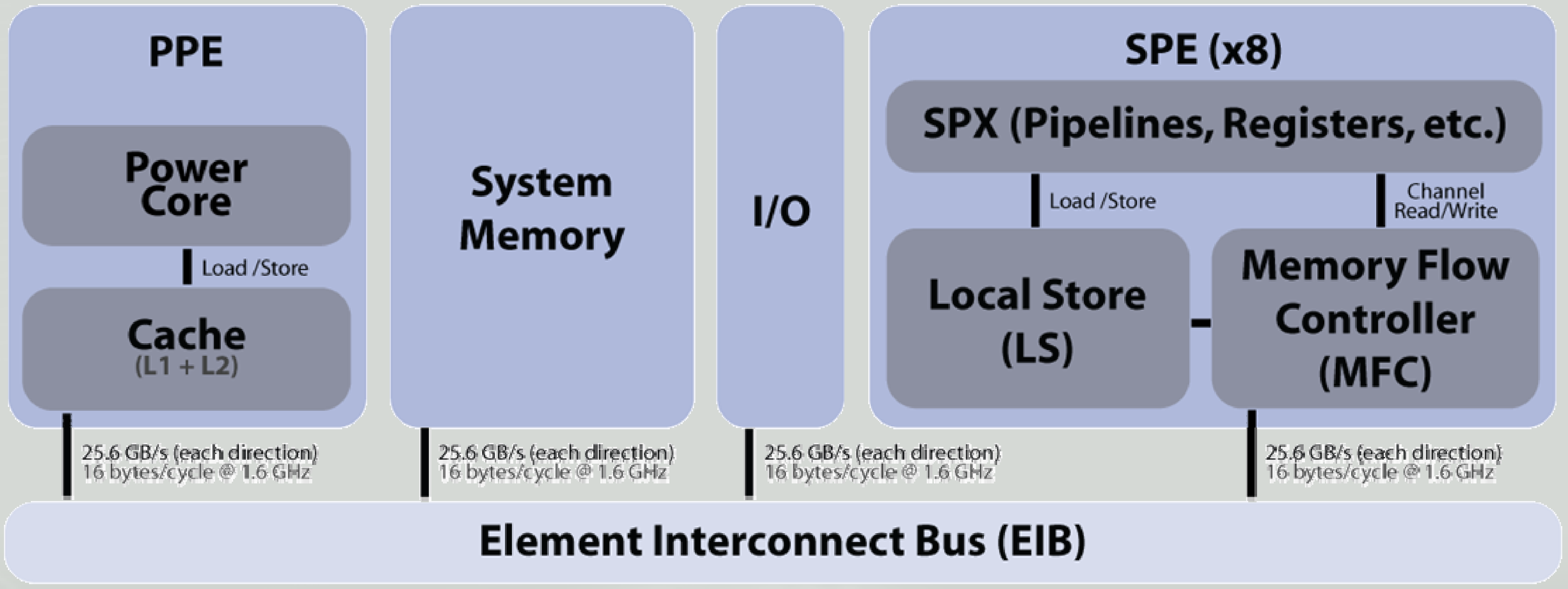
Charm++ with Accelerators

- Migration of chare objects
  - Utilize the Pack-UnPack (PUP) routines to handle issues such as byte ordering, pointer size, and so on

- Entry method handlers
  - Use handlers to direct message to specific architecture implementations of entry methods
    - Allow for optimized code for each architecture type
    - Difficulty in compiling single code block for multiple specialized architectures

- Load Balancer Framework
  - Utilize the current load balancer framework for migrating chare objects across processors

Cell Processor Architecture



Offload API

- Offload API
  - Interface for “offloading” chunks of work called “Work Requests” to the SPEs
    - General C/C++ library
    - Developed with the needs of Charm++ in mind
  - Multiple work requests in flight, automatically overlapping data movement to/from the SPEs with useful computation on the SPEs
  - User notified of work request completion through callback or work request handle (polling or blocking)
  - Gathers performance data on the SPEs which can be used to visualize program behavior in Projections

Offload API Code Example

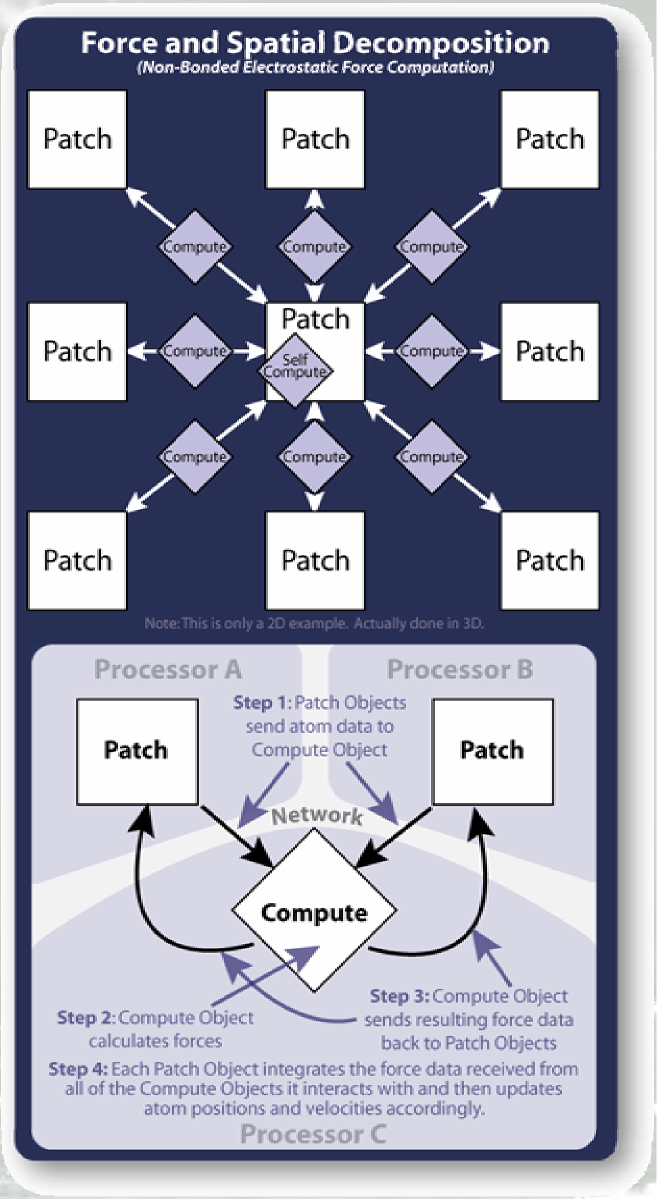
```
///// hello.cpp (PPE Only) //////////////////////
#include <stdio.h>
#include <string.h>
#include <spert_ppu.h> // Offload API Header
#include "hello_shared.h"
#define NUM_WORK_REQUESTS 10

int main(int argc, char* argv[]) {
  WRHandle wrHandle[NUM_WORK_REQUESTS];
  char msg[] __attribute__((aligned(128))) = { "Hello"
      };
  int msgLen = ROUNDUP_16(strlen(msg));

  InitOffloadAPI();

  // Send some work requests
  for (int i = 0; i < NUM_WORK_REQUESTS; i++)
    wrHandle[i] = sendWorkRequest(FUNC_SAYHI,
                                  NULL, 0,
                                  msg, msgLen,
                                  NULL, 0
                                 );

  // Wait for the work requests to finish
  for (int i = 0; i < NUM_WORK_REQUESTS; i++)
    waitForWRHandle(wrHandle[i]);

  CloseOffloadAPI();
  return EXIT_SUCCESS;
}
```

```
///// hello_spe.cpp (SPE Only) ///////////////
#include <stdio.h>
#include "spert.h" // SPE Runtime Header
#include "hello_shared.h"

inline void sayHi(char* msg) {
  printf("\"%s\" from SPE %d...\n",
         msg, (int)getSPEID());
}

void funcLookup(int funcIndex,
    void* readWritePtr, int readWriteLen,
    void* readOnlyPtr, int readOnlyLen,
    void* writeOnlyPtr, int writeOnlyLen,
    DMAListEntry* dmaList) {

  switch (funcIndex) {
    case SPE_FUNC_INDEX_INIT: break;
    case SPE_FUNC_INDEX_CLOSE: break;
    case FUNC_SAYHI:
      sayHi((char*)readOnlyPtr);
      break;
    default: // should never occur
      printf("ERROR :: Invalid funcIndex
     (%d)\n",
             funcIndex);
      break;
  }
}
```

```
///// Output //////
"Hello" from SPE 0...
"Hello" from SPE 7...
"Hello" from SPE 4...
"Hello" from SPE 5...
"Hello" from SPE 6...
"Hello" from SPE 2...
"Hello" from SPE 3...
"Hello" from SPE 0...
"Hello" from SPE 1...
"Hello" from SPE 1...
```

NAMD on Cell

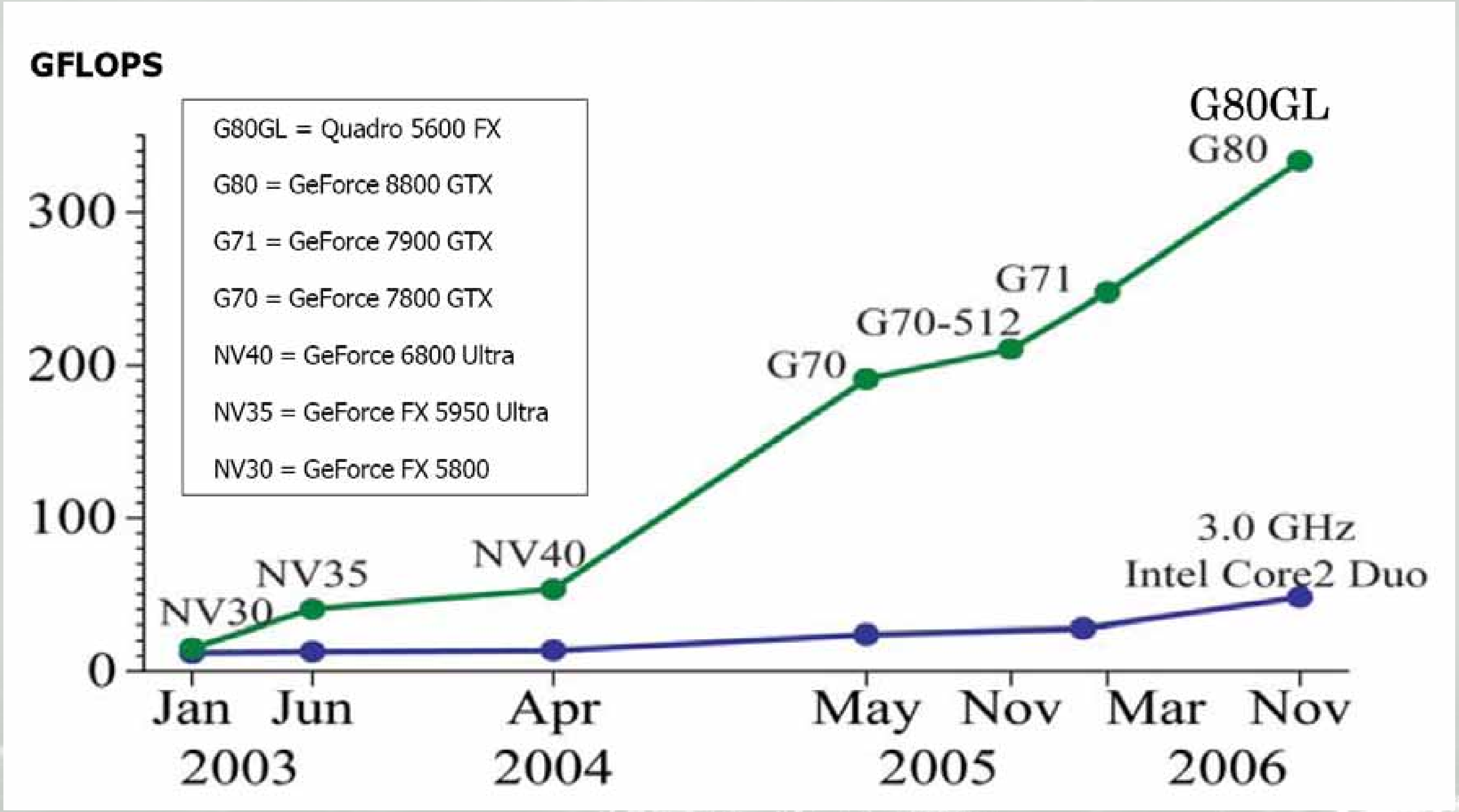
- NAnoscale Molecular Dynamics (NAMD)
- Can utilize multiple IBM Cell Blades and/or Sony PS3s in the same run
- Current performance (ApoA1)
  - 4 PS3s: approx. 540 - 560 ms/step
  - For comparison: 4 processors of…
    - BlueGene/L: ~1686 ms/timestep
    - XT3/XT4: ~452 ms/timestep
- Current limitations
  - GigE interconnect latency problems (especially during PME steps)
  - Using double precision floating point math


Force and Spatial Decomposition
(Non-Bonded Electrostatic Force Computation)
Patch
Compute
Self Compute
Note: This is only a 2D example. Actually done in 3D.
Processor A
Processor B
Step 1: Patch Objects send atom data to Compute Object
Network
Compute
Step 2: Compute Object calculates forces
Step 3: Compute Object sends resulting force data back to Patch Objects
Step 4: Each Patch Object integrates the force data received from all of the Compute Objects it interacts with and then updates atom positions and velocities accordingly.
Processor C

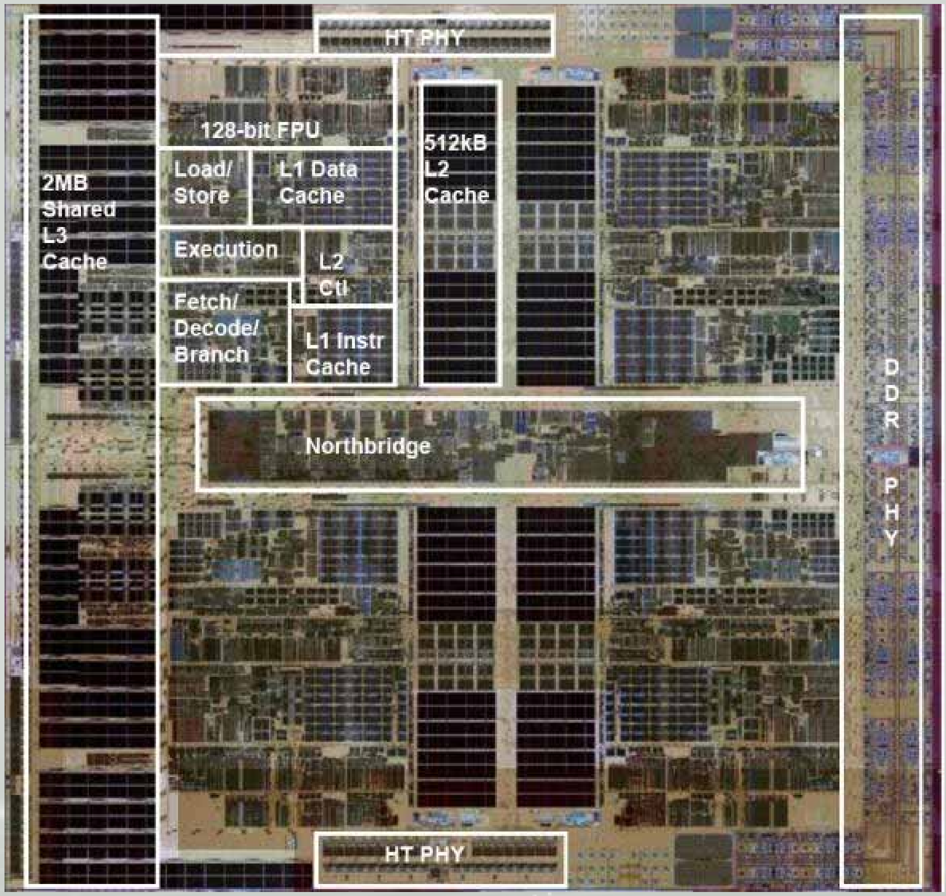
ChaNGa on Cell

- Charm++ N-body Gravity Solver (ChaNGa)
- Current performance
  - 1 PS3: approx. 3.5 seconds per iteration
  - Intel Quadcore: approx. 4 seconds per iteration

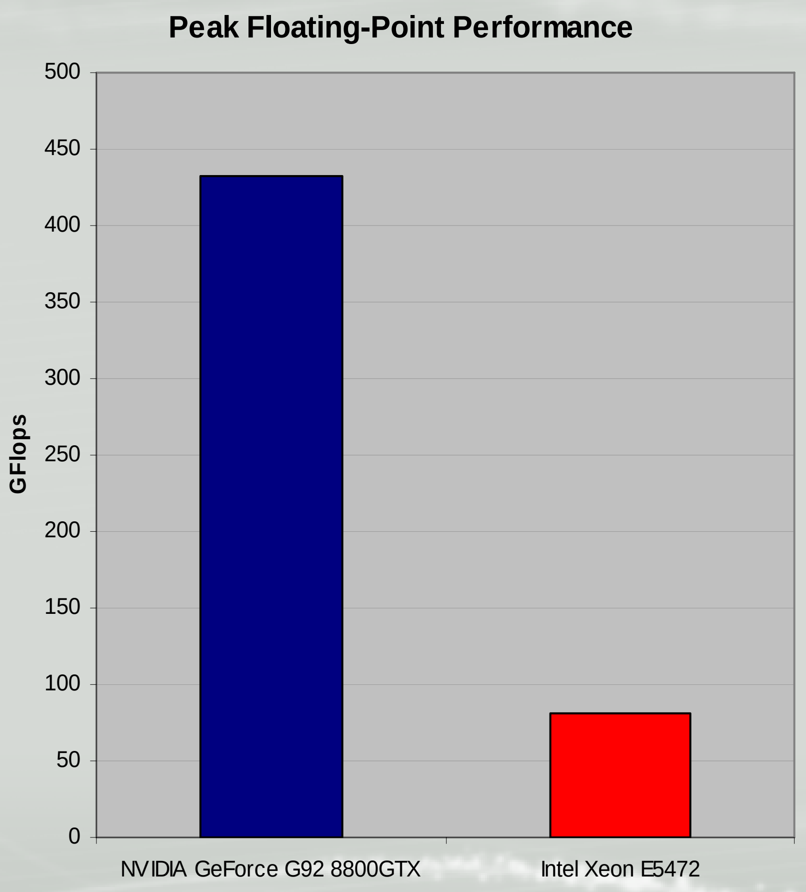
GPU Performance Trends

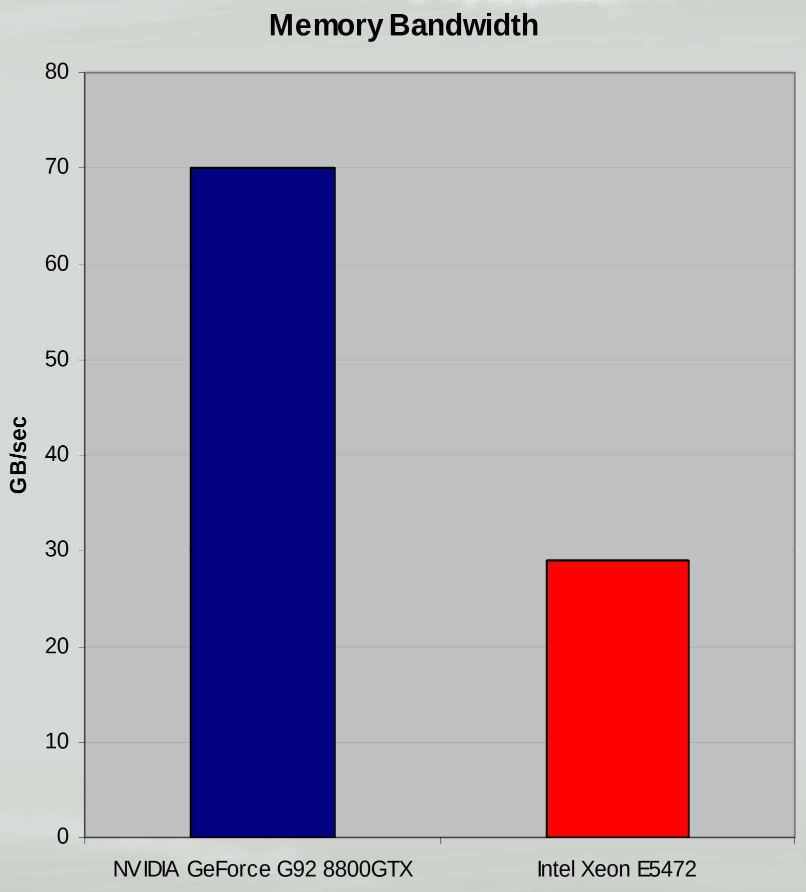


CPU Architecture



Performance Comparison





The General Purpose GPU

- “General Purpose” programming using graphics API
  - Difficult
  - Motivated the need for a special API
- APIs designed specifically for general purpose programming
  - CUDA
  - Required hardware support

CUDA

- Compute Unified Device Architecture
- C with extensions for general purpose programming on NVIDIA GPUs
- Kernels execute one by one on the GPU
- CPU in charge of data transfer and execution control

GPGPU Caveats

- Requirements for good performance
  - Large amount of data
  - High data parallelism
  - Limited communication
  - Simple control flow

GPGPUs and Hybrid HPC

- Supercomputers
  - Bull Novascale, a hybrid CPU/GPU system
    - 1068 8-core Intel CPU nodes (192 TFLOPS)
    - 96 NVIDIA GT200 GPUs (103 TFLOPS)
- Powerful workstations
  - A simple upgrade
  - Powerful platform for test runs

Charm++ on the GPGPU using CUDA

- Option 1: use CUDA calls directly inside Charm++ functions
- Option 2: place GPU work requests in a queue and let an offload API perform execution control

Direct Approach: CUDA in Charm++

- Difficulty of synchronization
  - Either GPU or CPU under-utilized
  - Kernel invocations and synchronizations in different logical units
- Poor code readability
  - Memory transfers
  - Separate files for CUDA and Charm++ code

Offload API

- User
  - Enqueues work requests to a GPU queue
  - Specifies a callback function per work request
- Runtime system
  - Automates memory transfer
  - Initiates kernel execution
  - Periodically checks for work request completion and offloads more work to the GPU

Asynchronous API

- Asynchronous data transfers to the GPU
  - Hide memory transfer time for subsequent kernels
- Simultaneous execution of multiple work requests
  - Light kernels not impeded by execution of larger kernels

Future Work

- Load balancing
  - Charm++ applications can currently execute across IBM Blades and PS3s
  - Enable Pack-UnPack routines to handle byte order, and so on
- Ease of Programming
  - Find common use scenarios for accelerators
  - Develop useful abstractions for both offloading work and managing concurrency

Summary

- New devices offer unprecedented floating point performance
- The Offload API allows programming Charm++ applications to effectively utilize heterogeneous systems

References

- http://www.intel.com/performance/server/xeon/hpcapp.htm
- http://techreport.com/articles.x/13224/3
- http://www.nvidia.com/object/geforce9.html
- http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf
- http://www.tgdaily.com/content/view/37070/135/

Questions?

Charm++ Code Example

```
///// hello.C (PPE Only) /////////////////////
#include "hello_shared.h"

class Main : public CBase_Main {

  public:
    Main(CkArgMsg* m) {
      // ...
      CProxy_Hello arr = CProxy_Hello::ckNew(nElements);
      arr[0].SayHi(17);
    };

    void done(void) {
      CkPrintf("All done\n");
      CkExit();
    };
};

class Hello : public CBase_Hello {

  public:

    void SayHi(int hiNo) {
      char buf[16] __attribute__((aligned(16)));
      sprintf(buf, "%d", thisIndex);
      sendWorkRequest(FUNC_SAYHI,
                      NULL, 0,           // RW
                      buf, strlen(buf)+1, // RO
                      NULL, 0,           // WO
                      CthSelf()
                     );
      CthSuspend();

      if (thisIndex < nElements-1)
        thisProxy[thisIndex+1].SayHi(hiNo+1);
      else
        mainProxy.done();
    }
};
```

```
///// hello_spe.cpp (SPE Only) ///////////////
#include "spert.h"
#include "hello_shared.h"

#ifdef __cplusplus
extern "C"
#endif
void funcLookup(int funcIndex,
                void* readWritePtr, int readWriteLen,
                void* readOnlyPtr, int readOnlyLen,
                void* writeOnlyPtr, int writeOnlyLen,
                DMAListEntry* dmaList
               ) {

  switch (funcIndex) {
    case FUNC_SAYHI: sayHi((char*)readWritePtr,
          (char*)readOnlyPtr); break;
    default:
      printf("!!! WARNING !!! Invalid funcIndex (%d)\n",
          funcIndex);
      break;
  }
}

void sayHi(char* readWritePtr, char* readOnlyPtr) {
  printf("\"Hi\"... \"%s\"\n", readOnlyPtr);
}
```

```
///// hello.ci //////
mainmodule hello {
  readonly CProxy_Main mainProxy;
  readonly int nElements;

  mainchare Main {
    entry Main(CkArgMsg *m);
    entry void done(void);
  };

  array [1D] Hello {
    entry Hello(void);
    entry [threaded] void SayHi(int hiNo);
  };
};
```

```
///// Output //////
“Hi“… “0”
“Hi“… “1”
“Hi“… “2”
“Hi“… “3”

…
(through nElements lines)
```