Hybrid CUDA, OpenMP, and MPI Parallel Programming on Multicore GPU Clusters

東海大學 資訊工程學系 楊朝棟
Chao-Tung Yang
Department of Computer Science
Tunghai University, Taichung 40704, Taiwan
e-Mail: ctyang@thu.edu.tw
http://web.thu.edu.tw/ctyang/www
Outline

• Introduction to Parallel Computing
• CUDA, OpenMP, and MPI Programming
• Our GPU Cluster Testbed
• Performance Evaluation
• Summary
Introduction to Parallel Computing

• Multicore microprocessors are available
  – More than one processor per chip

• Requires explicitly parallel programming
  – Compare with instruction level parallelism
    • Hardware executes multiple instructions at once
    • Hidden from the programmer
  – Hard to do by programmers
    • Programming for performance
    • Load balancing issue
    • Optimizing communication and synchronization
Introduction to Parallel Computing (cont.)

• Two commonly used parallel programming models: OpenMP and the Message Passing Interface (MPI).

  – OpenMP provides for parallelization of applications running within a single multicore-based system.
  – MPI provides for parallelization of applications running across many such systems, often referred to as a computing cluster.
Massively parallel computing has become a commodity technology.

The Democratization of Parallel Computing

- GPU + CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by Nvidia.

- Using CUDA, the latest Nvidia GPUs become accessible for computation like CPUs.
CUDA, OpenMP, and MPI Programming

• CUDA is the computing engine in Nvidia GPUs that is accessible to software developers through variants of industry standard programming languages.

• CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs.
CUDA, OpenMP, and MPI Programming (cont.)

- CUDA programming characteristics
  - Scalable parallel programming model
  - Minimal extensions to familiar C/C++ environment
  - Heterogeneous serial-parallel computing
  - Hierarchy of concurrent threads
  - Lightweight synchronization primitives
  - Shared memory model for cooperating threads
CUDA, OpenMP, and MPI Programming (cont.)

CUDA, OpenMP, and MPI Programming (cont.)

int main (int argc, char **argv) {
    1. Allocate memory space in device (GPU) for data
    2. Allocate memory space in host (CPU) for data
    3. Copy data from main memory to GPU
    4. Call “kernel” routine to execute on GPU
       (with CUDA syntax that defines no of threads and their physical structure)
    5. Transfer results from GPU to CPU
    6. Free memory space in device (GPU)
    7. Free memory space in host (CPU)

    return 0;
}
CUDA, OpenMP, and MPI Programming (cont.)

- OpenMP is often the easiest way to achieve moderate parallelism on shared memory machines.

- OpenMP contains compiler directives, library routines and environment variables.
  - OpenMP is a set of extensions to Fortran/C/C++
  - OpenMP is usually used to parallelize loops
  - Find your most time consuming loops
  - Split them up between threads
CUDA, OpenMP, and MPI Programming (cont.)

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    int nthreads, tid;

    /* Fork a team of threads giving them their own copies of variables */
    #pragma omp parallel private(nthreads, tid)
    {
        /* Obtain thread number */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
        /* All threads join master thread and disband */
    }
```
CUDA, OpenMP, and MPI Programming (cont.)

- [http://openmp.org/wp/](http://openmp.org/wp/)
Hybrid CUDA and OpenMP

- Assume calculate 1024 independent size matrix multiplication by 4 GPUs

```c
omp_set_num_threads(4);
#pragma omp parallel for
for(int i=0;i<1024;i++){
    cudaSetDevice(omp_get_thread_num());
    // do matrix multiplication by CUDA
}
```
Hybrid CUDA and OpenMP (cont.)

```c
printf(" =-=-=-=-=-=-=-=-n\n\n"));
    // Print Execution Information

    return 0;
}

__global__ void vecAdd_gpu_kernel(float vecA[], float vecB[], float vecC[])
{
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    // thread linearize
    // if(id < ELEMENT_COUNT)
    vecC[id] = vecB[id] + vecA[id];
}

void vecAdd_cpu(float vecA[], float vecB[], float vecC[])
{
    for(int i=0; i<FACTOR*ELEMENT_COUNT; i++)
        vecC[i] = vecB[i] + vecA[i];
}

bool check_result(float h_res[], float d_res[])
{
    for(int i=0; i<FACTOR * ELEMENT_COUNT; i++)
        if(h_res[i] != d_res[i])
        {
            printf(" %d, %f, %f\n", i, h_res[i], d_res[i]);
            return 0;
        }
    return 1;
}
```
CUDA, OpenMP, and MPI Programming (cont.)

- **Message Passing Interface**
  - Provide message passing library
  - Similar to network programming
  - Much easier and powerful using simple APIs
  - Support language - C, C++, Fortran, JAVA
  - Program can be run on multiple machines
  - Can get high-performance and scalability
CUDA, OpenMP, and MPI Programming (cont.)

- **Six basic MPI functions**
  - `MPI_Init` – initialize MPI environment
  - `MPI_Comm_size` – determine number of processes (processors)
  - `MPI_Comm_rank` – determine process’s (processor’s) rank
  - `MPI_Send` – blocking data send
  - `MPI_Recv` – blocking data receive
  - `MPI_Finalize` – shutting down MPI environment
CUDA, OpenMP, and MPI Programming (cont.)

• [http://www.open-mpi.org/](http://www.open-mpi.org/)
Hybrid CUDA and MPI

• Assume calculate 1024 independent size matrix multiplication
• “rank” is the current process id from MPI

```c
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
cudaGetDeviceCount(&device_count);
piece_size=1024/numprocs;
start_pos=piece_size*rank;
for(int i=start_pos; i<start_pos+piece_size; i++){
    cudaSetDevice(rank%device_count);
    // do matrix multiplication by CUDA
}
```
Hybrid CUDA and MPI (cont.)

```c
#include "mpi.h"
#include <stdio.h>
#include "kernel.h"
int main(int argc, char **argv)
{
    int myid, numprocs;
    int namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs)
    MPI_Get_processor_name(processor_name,&namelen);
    printf("Hello World! Process %d of %d on %s
",myid,processor_name);
    MPI_Finalize();
    return 0;
}
```

```
__global__ void hello(char* s){
    char w[50]="Hello CUDA ~~~ =^.^=":
    int k;

    for(k=0; w[k]!=0; k++)
        s[k]=w[k];
    s[k]=0;
}
```

```
int go(int cpu_id,char* name){
    char* d;
    char h[100];
    int gpu_id = -1;

    if(cpu_id >= 4){
        cudaSetDevice(cpu_id % 4);
    } else
        cudaSetDevice(cpu_id);
    cudaGetDevice(&gpu_id);
    cudaMemcpy(h, d, 100, cudaMemcpyDeviceToHost);
    printf("\n%s from Device %d on %s\n",h,gpu_id,name);
    cudaFree(d);
    cudaMemcpy(n,dn,1,cudaMemcpyDeviceToHost);
```
CUDA, OpenMP, and MPI Programming (cont.)

• **OpenMP part**
  - https://computing.llnl.gov/tutorials/openMP/

• **MPI part**
  - https://computing.llnl.gov/tutorials/mpi/
  - http://www.osc.edu/supercomputing/training/mpi/
  - http://www.mpi-forum.org/docs/docs.html
Our GPU Cluster Testbed

- Our GPU cluster testbed consists of one C2070, two C2050 and six C1060 Tesla cards.
- All nodes are connected by a gigabit switch.
# Tesla C-Series Workstation GPUs

<table>
<thead>
<tr>
<th></th>
<th>Tesla C1060</th>
<th>Tesla C2050</th>
<th>Tesla C2070</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architecture</strong></td>
<td>Tesla 10-series GPU</td>
<td>Tesla 20-series GPU</td>
<td></td>
</tr>
<tr>
<td><strong>Number of Cores</strong></td>
<td>240</td>
<td>448</td>
<td></td>
</tr>
<tr>
<td><strong>Caches</strong></td>
<td>16 KB Shared Memory / 8 cores</td>
<td>64 KB L1 cache + Shared Memory / 32 cores</td>
<td>768 KB L2 cache</td>
</tr>
<tr>
<td><strong>Floating Point Peak Performance</strong></td>
<td>933 Gigaflops (single)</td>
<td>1030 Gigaflops (single)</td>
<td>515 Gigaflops (double)</td>
</tr>
<tr>
<td><strong>GPU Memory</strong></td>
<td>4 GB</td>
<td>3 GB</td>
<td>6 GB</td>
</tr>
<tr>
<td><strong>Memory Bandwidth</strong></td>
<td>102 GB/s (GDDR3)</td>
<td>144 GB/s (GDDR5)</td>
<td></td>
</tr>
<tr>
<td><strong>System I/O</strong></td>
<td>PCIe x16 Gen2</td>
<td>PCIe x16 Gen2</td>
<td></td>
</tr>
<tr>
<td><strong>Power</strong></td>
<td>188 W (max)</td>
<td>237 W (max)</td>
<td>225 W (max)</td>
</tr>
<tr>
<td><strong>Available</strong></td>
<td>Available now</td>
<td>Available now</td>
<td>Available now</td>
</tr>
</tbody>
</table>
Connecting Tesla S1070 to a Host Server

- **PCle Gen2 Host Interface Cards**
- **PCle Gen2 Cables (0.5m length)**

**Host Server (oct5)**

**Tesla S1070**
Connecting Tesla S1070 to a Host Server

Host System (oct5) w/ 2 PCIe slots

Tesla S1070

PCIe Host Interface Card

PCIe Host Interface Card

NVIDIA Switch

NVIDIA Switch
Some Pictures
Turbo BOX Pro

Up to 4* GPU cards
Turbo BOX Pro

- Dedicated stylish external GPU enclosure with solid metal structure (時髦兼具的專用外接式GPU機箱，內具堅固的合金結構)
- High-speed 40 Gbps PCIe 2.0 ×8 host connectivity (高速 40 Gbps PCIe 2.0 ×8 電腦連線速率)
- Capable of up to four dual-slot GPU cards (最多可安裝四張兩寬GPU卡)
- Best heat dissipation required for high performance GPU cards (優異的機箱散熱系統，支援多張高效能GPU卡)
Our GPU Cluster Testbed

- Communication bandwidth between intra-node and inter-node

<table>
<thead>
<tr>
<th></th>
<th>C2050</th>
<th>C2050-1</th>
<th>C1060</th>
<th>oct5</th>
<th>C2070</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2050</td>
<td>39776.4</td>
<td>828.806</td>
<td>771.509</td>
<td>819.762</td>
<td>833.232</td>
</tr>
</tbody>
</table>
Compiling CUDA Codes

• Linux
  – Command line. CUDA provides nvcc (a NVIDIA “compiler-driver”. Use instead of gcc
  – nvcc -O3 -o <exe> <input> -
    I/usr/local/cuda/include -L/usr/local/cuda/lib64
    -lcudart
  – Separates compiled codes for CPU and for GPU and compiles all object code. Need regular C compiler installed for CPU. Make files also provided.

• Windows
  – NVIDIA suggests using Microsoft Visual Studio 2010
Performance Evaluation

Throughput of MatrixMul

Gflop/s

<table>
<thead>
<tr>
<th></th>
<th>C1060</th>
<th>C2050</th>
<th>C2070</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput of</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MatrixMul</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Performance Evaluation

C2050+C1060 VecAdd

second

normal (3:2) (10:7) (7:5)

1024*1024
2048*2048
Performance Evaluation

![C2050+C2070 VecAdd](image)

- Blue bar: 1024x1024
- Red bar: 2048x2048

The graph shows the performance comparison between two different resolutions:
- Normal mode
- (7:5) mode
Performance Evaluation
Summary

• We propose a parallel programming approach using hybrid CUDA, MPI, and OpenMP programming.

• The experiments reveal that the hybrid parallel multi-core GPU currently processing with OpenMP and MPI as a powerful approach of composing high performance clusters.

• For more detail, please refer
Summary (cont.)

• New trends in hardware and software technologies are likely to make multicore clusters more promising.

• Multicore GPU cluster computing system are rapidly becoming the standard platforms for high-performance computing.

• Hybrid CUDA and MPI programming is the most obvious approach to take advantage of GPU clustering performance.

• Multicore GPU cluster based supercomputers can be seen everywhere!
Thanks

e-Mail: ctyang@thu.edu.tw

http://web.thu.edu.tw/ctyang/www
Referred Papers

