Hybrid Parallel Programming on GPU Clusters

Chao-Tung Yang Chih-Lin Huang Cheng-Fang Lin Tzu-Chieh Chang

Department of Computer Science, Tunghai University, Taichung, 40704, Taiwan
ctyang@thu.edu.tw clj.joe@gmail.com superfun.lin@gmail.com s942826@gmail.com

***Abstract*—Nowadays, NVIDIA's CUDA is a general purpose scalable parallel programming model for writing highly parallel applications. It provides several key abstractions – a hierarchy of thread blocks, shared memory, and barrier synchronization. This model has proven quite successful at programming multithreaded many core GPUs and scales transparently to hundreds of cores: scientists throughout industry and academia are already using CUDA to achieve dramatic speedups on production and research codes. In this paper, we propose a hybrid parallel programming approach using hybrid CUDA and MPI programming, which partition loop iterations according to the number of C1060 GPU nodes in a GPU cluster which consists of one C1060 and one S1070. Loop iterations assigned to one MPI process are processed in parallel by CUDA run by the processor cores in the same computational node.**

Keywords: CUDA, GPU, MPI, OpenMP, hybrid, parallel programming

I. INTRODUCTION

Nowadays, NVIDIA's CUDA [1, 16] is a general purpose scalable parallel programming model for writing highly parallel applications. It provides several key abstractions – a hierarchy of thread blocks, shared memory, and barrier synchronization. This model has proven quite successful at programming multithreaded many core GPUs and scales transparently to hundreds of cores: scientists throughout industry and academia are already using CUDA [1, 16] to achieve dramatic speedups on production and research codes. In NVDIA the CUDA chip, all to the core of hundreds of ways to construct their chips, in here we will try to use NVIDIA to provide computing equipment for parallel computing.

This paper proposes a solution to not only simplify the use of hardware acceleration in conventional general purpose applications, but also to keep the application code portable. In this paper, we propose a parallel programming approach using hybrid CUDA, OpenMP and MPI [3] programming, which partition loop iterations according to the performance weighting of multi-core [4] nodes in a cluster. Because iterations assigned to one MPI process are processed in parallel by OpenMP threads run by the processor cores in the same computational node, the number of loop iterations allocated to one computational node at each scheduling step depends on the number of processor cores in that node.

In this paper, we propose a general approach that uses performance functions to estimate performance weights for each node. To verify the proposed approach, a heterogeneous cluster and a homogeneous cluster were built. In our implementation, the master node also participates in computation, whereas in previous schemes, only slave nodes do computation work. Empirical results show that in heterogeneous and homogeneous clusters environments, the proposed approach improved performance over all previous schemes.

The rest of this paper is organized as follows. In Section 2, we introduce several typical and well-known self-scheduling schemes, and a famous benchmark used to analyze computer system performance. In Section 3, we define our model and describe our approach. Our system configuration is then specified in Section 4, and experimental results for three types of application program are presented. Concluding remarks and future work are given in Section 5.

II. BACKGROUND REVIEW

A. History of GPU and CUDA

In the past, we have to use more than one computer to multiple CPU parallel computing, as shown in the last chip in the history of the beginning of the show does not need a lot of computation, then gradually the need for the game and even the graphics were and the need for 3D, 3D accelerator card appeared, and gradually we began to display chip for processing, began to show separate chips, and even made a similar in their CPU chips, that is GPU.

We know that GPU computing could be used to get the answers we want, but why do we choose to use the GPU? This slide shows the current CPU and GPU comparison. First, we can see only a maximum of eight core CPU now, but the GPU has grown to 260 core, the core number, we'll know a lot of parallel programs for GPU computing, despite his relatively low frequency of core, we I believe a large number of parallel computing power could be weaker than a single issue. Next, we know that there are within the GPU memory, and more access to main memory and GPU CPU GPU access on the memory capacity, we find that the speed of accessing GPU faster than CPU by 10 times, a whole worse 90GB / s, This is quite alarming gap, of course, this also means that when computing the time required to access large amounts of data can have a good GPU to improve.

CPU using advanced flow control such as branch predict or delay branch and a large cache to reduce memory access latency, and GPU's cache and a relatively small number of flow control nor his simple, so the method is to use a lot of GPU computing devices to cover up the problem of memory latency, that is, assuming an access memory GPU takes 5

978-0-7695-4190-7/10 $26.00 © 2010 IEEE
DOI 10.1109/ISPA.2010.97

seconds of the time, but if there are 100 thread simultaneous access to, the time is 5 seconds, but the assumption that CPU time memory access time is 0.1 seconds, if the 100 thread access, the time is 10 seconds, therefore, GPU parallel processing can be used to hide even in access memory than CPU speed. GPU is designed such that more transistors are devoted to data processing rather than data caching and flow control, as schematically illustrated by Figure 1.

Therefore, we in the arithmetic logic by GPU advantage, trying to use NVIDIA's multi-core available to help us a lot of computation, and we will provide NVIDIA with so many core programs, and NVIDIA Corporation to provide the API of parallel programming large number of operations to carry out.

We must use the form provided by NVIDIA Corporation GPU computing to run it? Not really. We can use NVIDIA CUDA, ATI CTM and apple made OpenCL (Open Computing Language), is the development of CUDA is one of the earliest and most people at this stage in the language but with the NVIDIA CUDA only supports its own graphics card, from where we You can see at this stage to use GPU graphics card with the operator of almost all of NVIDIA, ATI also has developed its own language of CTM, APPLE also proposed OpenCL (Open Computing Language), which OpenCL has been supported by NVIDIA and ATI, but ATI CTM has also given up the language of another, by the use of the previous relationship between the GPU, usually only support single precision floating-point operations, and in science, precision is a very important indicator, therefore, introduced this year computing graphics card has to support a Double precision floating-point operations.

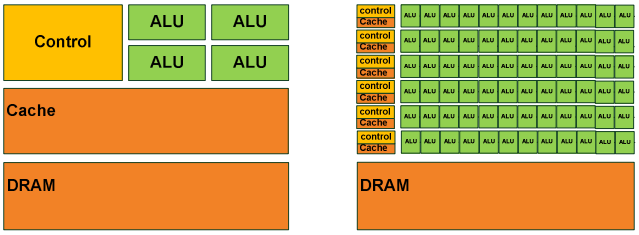

Figure 1: The CPU Devotes More Transistors to Data Processing

B. CUDA Programming

CUDA (an acronym for Compute Unified Device Architecture) is a parallel computing [2] architecture developed by NVIDIA. CUDA is the computing engine in NVIDIA graphics processing units or GPUs that is accessible to software developers through industry standard programming languages. The CUDA software stack is composed of several layers as illustrated in Figure 2: a hardware driver, an application programming interface (API) and its runtime, and two higher-level mathematical libraries of common usage, CUFFT [17] and CUBLAS [18]. The hardware has been designed to support lightweight driver and runtime layers, resulting in high performance. CUDA architecture supports a range of computational interfaces including OpenGL [9] and Direct Compute. CUDA's parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C. At its core are three key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization – that are simply exposed to the programmer as a minimal set of language extensions.

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel, and then into finer pieces that can be solved cooperatively in parallel. Such a decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables transparent scalability since each sub-problem can be scheduled to be solved on any of the available processor cores: A compiled CUDA program can therefore execute on any number of processor cores, and only the runtime system needs to know the physical processor count.

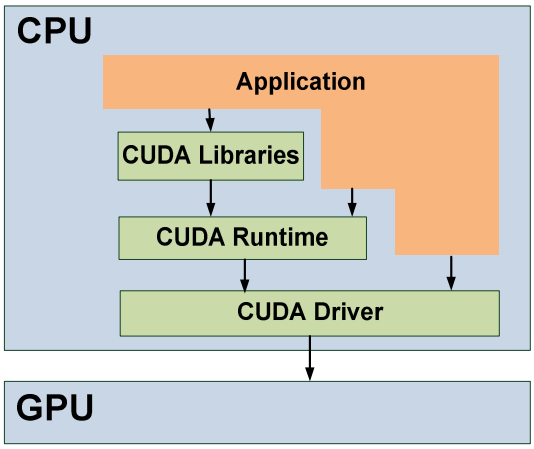

Figure 2: Compute Unified Device Architecture Software Stack

C. CUDA Processing flow

In follow illustration, CUDA processing flow is described as Figure 3 [16]. The first step: copy data from main memory to GPU memory, second: CPU instructs the process to GPU, third: GPU execute parallel in each core, finally: copy the result from GPU memory to main memory.

D. Run and Build CUDA on Ubuntu

In this session, we will describe how to build CUDA on Linux Ubuntu OS.

- Go to NVIDIA official web site to download CUDA toolkit, CUDA SDK and CUDA SDK for Linux.
- Exit X-Window and install NVIDIA Driver then execute CUDA toolkit and SDK install run files.

```
$ sudo apt-get install libglu1-mesa-dev libxmu-dev libglui-dev
libX11-dev libXi-dev build-essential gcc-4.1 g++-4.1
```

- Change directory to /usr/bin and remove gcc, g++, i486-linux-gnu-gcc-4.3, i486-linux-gnu-g++-4.3, and links simply because most of the sample codes do not support gcc 4.3, you can do this by execute the following commands.

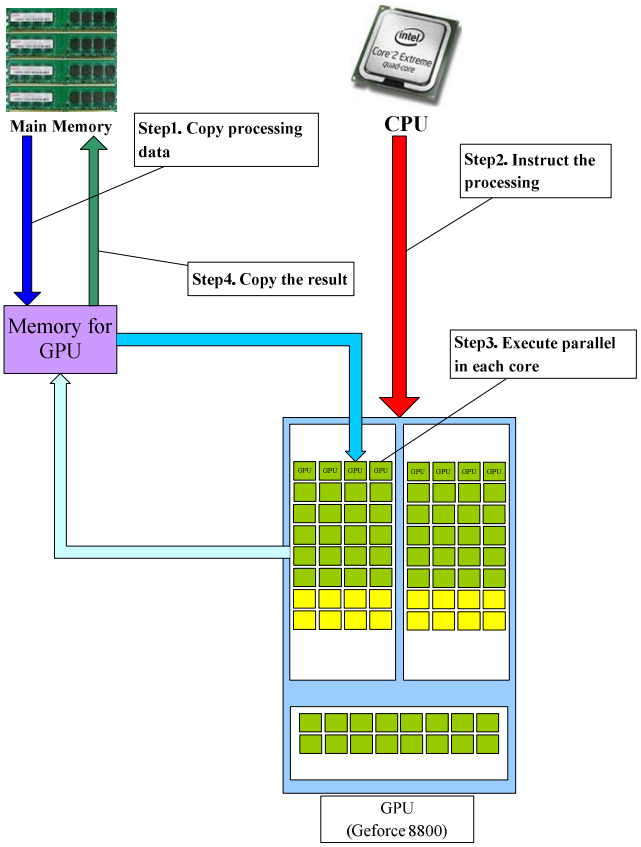

Figure 3: Processing flow on CUDA from Wiki [16]

```
$ cd /usr/bin
$ sudo rm gcc i486-linux-gnu-gcc g++ i486-linux-gnu-g++
$ sudo ln -s gcc-4.1 gcc;sudo ln -s g++-4.1 g++
$ sudo ln -s i486-linux-gnu-g++-4.1 i486-linux-gnu-g++
$ sudo ln -s i486-linux-gnu-gcc-4.1 i486-linux-gnu-gcc
$ sudo echo /usr/local/cuda/lib >>/etc/ld.so.conf
$ sudo ldconfig
```

- Go to the directory which you choose cuda SDK to install to. Here use ~/NVIDIA_CUDA_SDK as example,then execute make to compile sample code, the executable file will leave in bin/linux/release/

```
$ cd ~/NVIDIA_CUDA_SDK;make
$ cd bin/linux/release/
```

E. OpenMP Programming

In contrast, Open Multi-Processing (OpenMP) [6], a kind of shared memory architecture API [35], provides a multi-threaded capacity. A loop can be parallelized easily by invoking subroutine calls from OpenMP thread libraries and inserting the OpenMP compiler directives. In this way, the threads can obtain new tasks, the un-processed loop iterations, directly from local shared memory.

OpenMP is an open specification for shared memory parallelism. The basic idea behind OpenMP is data-shared parallel execution. It consists of a set of compiler directives, callable runtime library routines and environment variables that extend FORTRAN, C and C++ programs. OpenMP is portable across the shared memory architecture. The unit of workers in OpenMP is threads. It works well, when accessing shared data costs you nothing. Every thread can access a variable in shared cache or RAM.

The OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++ and FORTRAN on much architecture, including UNIX and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

F. Combining MPI and CUDA

The simplest way forward is to use nvcc for everything. The nvcc compiler wrapper is somewhat more complex than the typical mpicc compiler wrapper, so it's easier to make MPI code into .cu and compile with nvcc than the other way around. A sample make file might resemble:

```
    [arnoldg@ac14 mpi-gpu]$  cat Makefile
      MPICC      := nvcc -Xptxas -v
      MPI_INCLUDES    := /usr/mpi/intel/mvapich2-1.2p1/include
      MPI_LIBS   := /usr/mpi/intel/mvapich2-1.2p1/lib
    %.o : %.cu
      $(MPICC) -I$(MPI_INCLUDES) -o $@ -c $<
    mpi_hello_gpu : vecadd.o  mpi_hello_gpu.o
      $(MPICC) -L$(MPI_LIBS) -lmpich -o $@ *.o
    clean :
      rm vecadd.o mpi_hello_gpu.o
all : mpi_hello_gpu
```

Source code files as follow:

```
    [arnoldg@ac14 mpi-gpu]$  cat mpi_hello_gpu.cu
      #include <mpi.h>
      #include <stdio.h>
      #include <stdlib.h>
      #define PPN 4
      #define INTARRAYLEN 65535
      #define BCASTREPS 1000
    int main(int argc, char  *argv[])
      {
      int bcastme[INTARRAYLEN], ranksum;
      int rank, size, len;
      int gpudevice;
      int vecadd(int, int);
      char name[MPI_MAX_PROCESSOR_NAME];
```

```
            MPI_Init(&argc, &argv);

      MPI_Comm_rank(MPI_COMM_WORLD,  &rank);

      MPI_Comm_size(MPI_COMM_WORLD,  &size);

      MPI_Get_processor_name(name, &len);

         // do some MPI work, showing MPI and CUDA being run
from one  routine

      if (rank == 0) { bcastme[3]=3; }

      for (int i=0; i<BCASTREPS; i++)

      {

      MPI_Bcast(bcastme,      INTARRAYLEN,      MPI_INT,      0,
MPI_COMM_WORLD);

      }

         // modulo is useful in determining unique gpu device
ids if  ranks

      // are packed into nodes and not assigned in round robin
fashion



      gpudevice= rank % PPN;

      printf("rank %d of %d on %s received bcastme[3]=%d [gpu
%d]\n", rank, size, name,bcastme[3], gpudevice);



      vecadd(gpudevice, rank);

         // more MPI work showing MPI is functional after CUDA

      MPI_Reduce(&rank,  &ranksum,  1,    MPI_INT,  MPI_SUM,  0,
MPI_COMM_WORLD);

      if (rank == 0) { printf("ranksum= %d\n", ranksum); }

         MPI_Finalize();

      }

  [arnoldg@ac14 mpi-gpu]$
```

Parameters for passing the MPI rank and selecting a gpu were added to vecadd.

```
    [arnoldg@ac14 mpi-gpu]$  cat vecadd.cu

      // Kernel definition

      __global__ void  vecAdd(float* A, float* B, float* C)

      {

      int i = threadIdx.x;

      A[i]=0;

      B[i]=i;

      C[i] = A[i] + B[i];

      }

   #include <stdio.h>

      #define SIZE 10

      #define KERNELINVOKES  5000000

      int vecadd(int  gpudevice, int rank)

      {

      int devcheck(int, int);

      devcheck(gpudevice, rank);

        float A[SIZE], B[SIZE], C[SIZE];

      // Kernel invocation

        float *devPtrA;

      float *devPtrB;

      float *devPtrC;

      int memsize= SIZE * sizeof(float);

        cudaMalloc((void**)&devPtrA, memsize);

      cudaMalloc((void**)&devPtrB, memsize);

      cudaMalloc((void**)&devPtrC, memsize);

      cudaMemcpy(devPtrA, A, memsize,  cudaMemcpyHostToDevice);

      cudaMemcpy(devPtrB, B, memsize,  cudaMemcpyHostToDevice);

      for (int i=0; i<KERNELINVOKES; i++)

      {

      vecAdd<<<1,  gpudevice>>>(devPtrA, devPtrB, devPtrC);

      }

      cudaMemcpy(C, devPtrC, memsize,  cudaMemcpyDeviceToHost);

      // calculate only up to  gpudevice to show the unique
output

      // of each rank's kernel  launch

      for (int i=0; i<gpudevice; i++)

      printf("rank %d: C[%d]=%f\n",rank,i,C[i]);

      cudaFree(devPtrA);

      cudaFree(devPtrA);

      cudaFree(devPtrA);

      }

    int devcheck(int  gpudevice, int rank)

      {

      int device_count=0;

      int  device;     // used with cudaGetDevice() to verify
cudaSetDevice()

      cudaGetDeviceCount( &device_count);

      if (gpudevice >= device_count)

      {

      printf("gpudevice >=  device_count ... exiting\n");

      exit(1);

      }

      cudaError_t cudareturn;

      cudaDeviceProp deviceProp;

      cudaGetDeviceProperties(&deviceProp,  gpudevice);

      if (deviceProp.warpSize <= 1)

      {

      printf("rank  %d:  warning,  CUDA  Device  Emulation  (CPU)
detected, exiting\n", rank);

      exit(1);

      }

      cudareturn=cudaSetDevice(gpudevice);

      if (cudareturn == cudaErrorInvalidDevice)

      {

      perror("cudaSetDevice returned  cudaErrorInvalidDevice");

      }

      else

      {

      cudaGetDevice(&device);

      printf("rank %d:  cudaGetDevice()=%d\n",rank,device);

      }

    }

  [arnoldg@ac14 mpi-gpu]$
```

III. System Hardware

A. Tesla C1060 GPU Computing Processor

The NVIDIA® Tesla™ C1060 transforms a workstation into a high-performance computer that outperforms a small cluster. This gives technical professionals a dedicated computing resource at their desk-side that is much faster and more energy-efficient than a shared cluster in the data center. The NVIDIA® Tesla™ C1060 computing processor board which consists of 240 cores is a PCI Express 2.0 form factor computing add-in card based on the NVIDIA Tesla T10 graphics processing unit (GPU). This board is targeted as high-performance computing (HPC) solution for PCI Express systems. The Tesla C1060 [15] is capable of 933 GFLOPs/s [13] of processing performance and comes standard with 4 GB of GDDR3 memory at 102 GB/s bandwidth.

A computer system with an available PCI Express ×16 slot is required for the Tesla C1060. For the best system bandwidth between the host processor and the Tesla C1060, it is recommended (but not required) that the Tesla C1060 be installed in a PCI Express ×16 Gen2 slot. The Tesla C1060 is based on the massively parallel, many-core Tesla processor, which is coupled with the standard CUDA C programming [14] environment to simplify many-core programming.

B. Tesla S1070 GPU Computing System

The NVIDIA® Tesla™ S1070 [12] computing system speeds the transition to energy-efficient parallel computing [2]. With 960 processor cores and a standard C compiler that simplifies application development, Tesla S1070 scales to solve the world's most important computing challenges—more quickly and accurately. The NVIDIA® Tesla™ S1070 Computing System is a 1U [12] rack-mount system with four Tesla T10 computing processors. This system connects to one or two host systems via one or two PCI Express cables. A Host Interface Card (HIC) [5] is used to connect each PCI Express cable to a host. The host interface cards are compatible with both PCI Express 1x and PCI Express 2x systems.

The Tesla S1070 GPU computing system is based on the T10 GPU from NVIDIA. It can be connected to a single host system via two PCI Express connections to that host, or connected to two separate host systems via one PCI Express connection to each host. Each NVIDIA switch and corresponding PCI Express cable connects to two of the four GPUs in the Tesla S1070. If only one PCI Express cable is connected to the Tesla S1070, only two of the GPUs will be used. To connect all four GPUs in a Tesla S1070 to a single host system, the host must have two available PCI Express slots and be configured with two cables.

IV. Experimental Results

We built a heterogeneous GPC cluster consisting of one Tesla C1060 and a Tesla S1070, each with Gigabit Ethernet NIC interconnected via a D-LINK DGS-3100-24 Gigabit switch. To verify our approach, illustrate our cluster environment, and describe the terminology for our application, we implemented programs with MPI/OpenMP for execution on our test bed. We then compared the performance of our scheme with those of other static and dynamic schemes using heterogeneous and homogeneous clusters to solve problems in Matrix Multiplication, MD5 and Bubble Sorting. We also discuss performance comparisons on heterogeneous and homogeneous clusters. From Figures 4 to 6, we take log of 10 at execution time to emphasis the differences. Figure 4 shows that the performance of GPU on processing the massively parallel execution as the application of Matrix Multiplication form 256 to 2048. Figure 5 reveals that single GPU presents better performance than single CPU with multiple threads on MD5 hashing computation. Finally, Figure 6 shows that the comparison of performance on multiple GPU with MPI and OpenMP.

V. Conclusion

In conclusion, we propose a parallel programming approach using hybrid CUDA and MPI programming, which partition loop iterations according to the number of C1060 GPU nodes in a GPU cluster which consists of one C1060 and one S1070. During the experiments, loop iterations assigned to one MPI process are processed in parallel by CUDA run by the processor cores in the same computational node. 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.

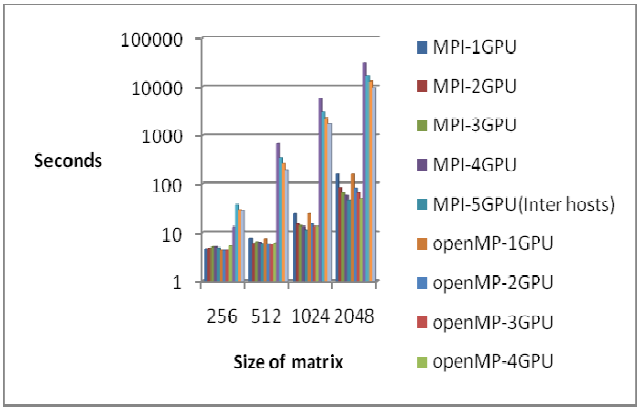

Figure 4: Matrix Multiplication with problem sizes from 256 to 2048

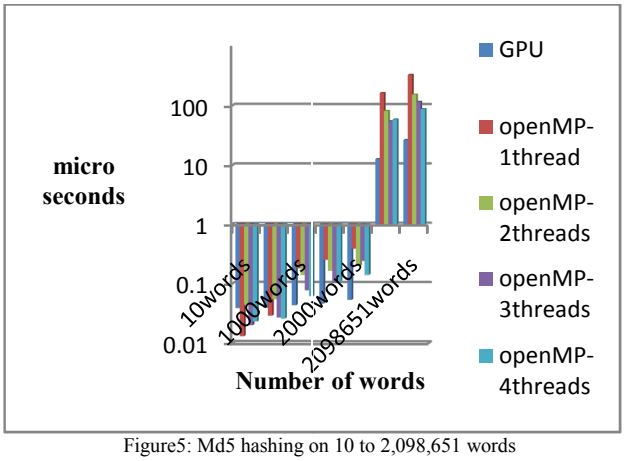

Figure5: Md5 hashing on 10 to 2,098,651 words

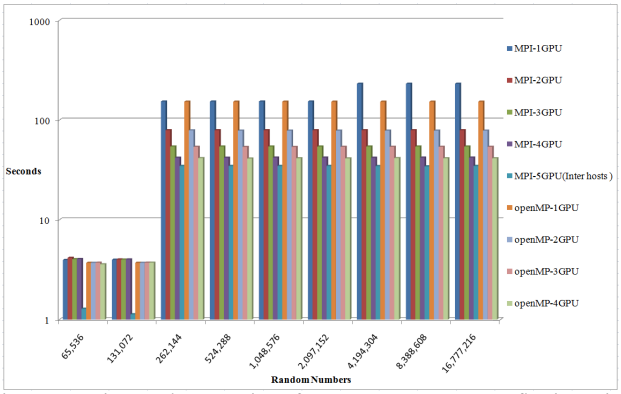

Figure 6: Sorting numbers 640 times from 65,536 to 16,777,216 floating point numbers.

ACKNOLOGEMENTS

This work is supported in part by the National Science Council, Taiwan, under grants no. NSC 98-2220-E-029-001- and NSC 98-2220-E-029-004-.

REFERENCES

[1] Download cuda, http://developer.nvidia.com/object/cuda.htm

[2] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. Buijssen,M. Grajewski, and S. Tureka, "Exploring weak scalability for FEM calculations on a GPU-enhanced cluster," Parallel Computing, vol. 33, pp. 685-699, Nov 2007.

[3] P. Alonso, R. Cortina, F.J. Martínez-Zaldívar, J. Ranilla "Neville elimination on multi- and many-core systems: OpenMP, MPI and CUDA", Jorunal of Supercomputing, http://www.springerlink.com/content/h49626615t707334/fulltext.pdf

[4] Francois Bodin and Stephane Bihan, "Heterogeneous multicore parallel programming for graphics processing units", Scientific Programming, Volume 17, Number 4 / 2009, 325-336, Nov. 2009.

[5] Specification Tesla S1070 GPU Computing System, http://www.nvidia.com/docs/IO/43395/SP-04154-001_v02.pdf.

[6] Open MP Specification, http://openmp.org/wp/about-openmp/

[7] Message Passing Interface (MPI), http://www.mcs.anl.gov/research/projects/mpi/

[8] MPICH, A Portable Implementation of MPI, http://www.mcs.anl.gov/research/projects/mpi/mpich1/index.htm.

[9] OpenGL, D. Shreiner, M. Woo, J. Neider and T. Davis, OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2 (5th Edition), Addison-Wesley, Reading, MA, August 2005.

[10] (2008) Intel 64 Tesla Linux Cluster Lincoln webpage. [Online] Available: http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64TeslaCluster/

[11] Romain Dolbeau, Stéphane Bihan, and François Bodin, HMPP: A Hybrid Multi-core Parallel Programming Environment

[12] The NVIDIA Tesla S1070 1U Computing System - Scalable Many Core Supercomputing for Data Centers http://www.nvidia.com/object/product_tesla_s1070_us.html

[13] Top 500 Super Computer Sites, What is Gflop/s, http://www.top500.org/faq/what_gflop_s

[14] NVIDIA CUDA Programming Guide, http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

[15] NVIDIA Tesla C1060 Computing Processor, http://www.nvidia.com/object/product_tesla_c1060_us.html

[16] CUDA, http://en.wikipedia.org/wiki/CUDA

[17] CUFFT, CUDA Fast Fourier Transform (FFT) library. http://developer.download.nvidia.com/compute/cuda/1_1/CUFFT_Library_1.1.pdf

[18] CUBLAS, BLAS(Basic Linear Algebra Subprograms) on CUDA http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUBLAS_Library_2.0.pdf