Vol.2012-HPC-133 No.2
2012/3/26

A precise measurement tool for power dissipation of CUDA kernels

Luo Cheng,[*1,*2] Kamil Rocki[*1,*2] and Reiji Suda[*1,*2]

Power dissipation has become an important factor for evaluating application performance. To help programmers have a further understand on the power dissipation of their applications, we are investigating tools to achieve precise and easily using measurement tools. In this paper we discuss power measurements of GPU, propose an API tool which include two parts: host part and monitor part. The host part is used for measurement initial, communication with monitor machine and data process. The monitor part is used for real-time voltages and current data collection. By using this tool, programmers can get accurate power dissipation of their applications.

1. Introduction

Power dissipation has become one of the most important factors in the development of high performance computers. Along with the growing demand for high performance computers from scientific computing, power dissipation of high performance computers should be reduced effectually while ensuring performance. This problem demands many efforts of power reduction in many possible fators such as cooling hardware, disk, processor and software.

Graphics Processing Unit(GPU) now is considered as high performance computing accelerators and is wildly used in high performance computers. With the advantage for massively parallel processing and vector computation, GPU can achieve high performance comparing to CPU. Although GPUs can consujme more power than CPUs, however the performance ratio is much higher than the power ratio when comparing GPU to CPU. Therefore, GPU can be more energy efficient than CPU from the performance/power aspect[1]. Despite of GPUs' energy efficient, GPUs still consume significant power which have reached 300W and will still increase in the future. Toward green computing on high performance computers, more power-efficient software methodologies should be applied.

Our research aims to develop a precise measurement tool for power disspation of CUDA kernels running on GPUs. Our measurement tool is basing on a precise meaurement method for power disspation of CUDA kernels provided by Suda[2] which not only measure the power supply from PSU(Power Supply Unit) but also the power supply from PCI-Express bus. Our approach use NI probers[3] to measure the currents and voltages of GPU power supplies during CUDA kernels execution. In our tool, a labview project is set in monitor machine to control the measurement process and collect the measurement data. After the execution of CUDA kernels, measurement data will be processed and transfered to host machine where the GPU is equipped. The host machine receive the data and store them into the local database. Finally, a set of API is applied for the power measurement control and database access. With this measurement tool, CUDA programmers can easily know the precise power dissipation of their CUDA kernels.

2. Background

In this section, we will briefly introduce some technical details of CUDA and GPU hardware which related to our work. More details can be found in the CUDA programming guide[4].

2.1 GPU architecture and CUDA programming model

The GPU architecture contains a scalable number of streaming multiple processors(SM). Each SM contains multiple streaming processors(SPs). There 8 SPs in each SM in GTX 260[5] and 32 SPs in each SM in Fermi chips[6].

CUDA(Compute Unified Device Architectrue) is a C language liked parallel computing architecture for programming on NVIDIA GPU. One *kernel* functions are considered as the computations on one GPU which will be executed by a number of threads. The numbers of blocks and threads will be specified when the kernel is launched by the host. A *block* means a set of a certain number of threads, and all blocks in the kernel launch have the same numbers of threads. A *warp* is formed by 32 consecutive threads in a block. Threads within one warp are

∗1 Presently with The University of Tokyo
∗2 Presently with CREST,JST

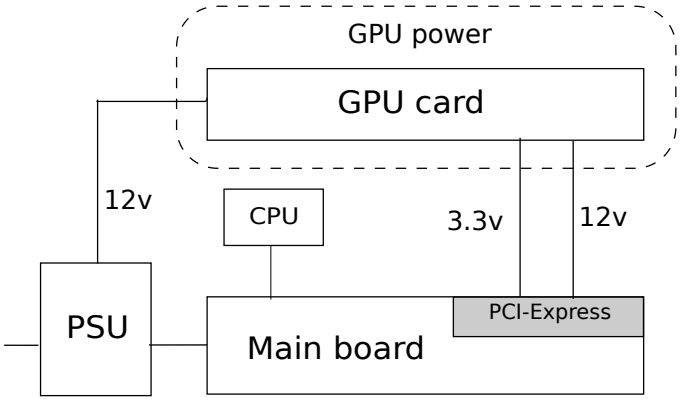

Fig. 1 GPU power measurement

executed in SIMD(Single Instruction Multiple Data) manner, and warps within one block are executed concurrently in SPMD(Single Process Multiple Data) manner.

2.2 Power Measurements of GPU

In this paper, we assume GPU to be provided on a *video card* which includes video memory, fan and other circuit components. Therefore, the power measurement of a GPU should include the power dissipation of all circut components on the GPU.

A GPU card is connected to the main board of the host processor via PCI-Express. Parts of power for a GPU card is supplied through the PCI-Express bus, and other power is supplied directly from the PSU as shown in **Fig. 1**. Let us assume the power from PCI-Express bus as *PCI-E power* and the power from PSU as *PSU power*. The PCI-E power has two voltages: 12 V and 3.3V while the voltage of PSU power is 12V.

To measure the current of the line between PSU and GPU card, a clamp probe is used to measure the line. To measure the voltage of the line, parts of the coating of two of the PSU lines is removed(one is 12V and one is GND), a voltage probe is connect to the two lines.

To measure the power supply from the PCI-Express connector, the currents and voltages should be measured. However, it is impossible to remove parts of

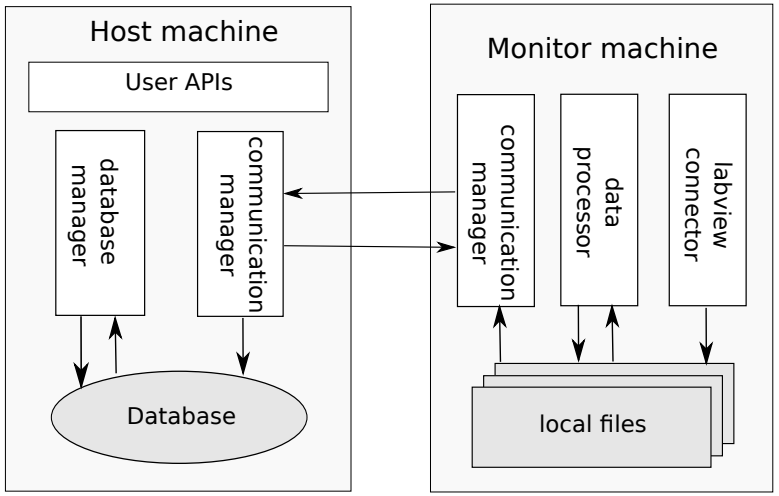

Fig. 2 Architecture of measurement tool

the coating of the PCI-Express lines. Therefore, a riser card is used to connect the GPU card and the PCI-Express bus. The 12V and 3.3V power supply lines of the riser card are separated from others. And parts of the coating of the two line and one GND line of the riser card are removed. Then, two clamp probes are used to measure the currents of the two lines and two voltage probes are connect to the three lines to measure the voltages of the 12V power line and 3.3V power line.

3. Architecture

In our precise measurement tool for power dissipation, two machines are used. One machine is called *host machine* and another is called *monitor machine*. The host machine is installed with Linux OS and equipped with one GPU to run CUDA kernels. The monitor machine is installed with windows OS as labview[7] is needed for data collection. The host machine is responsible for CUDA kernel launching, finial power dissipation data storage and monitor machine control. The monitor machine is responsible for the GPU currents and voltages measurement, power dissipation data collection, data process and data transmission with

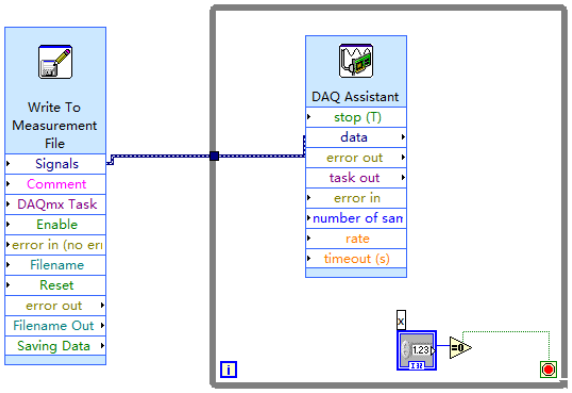

Fig. 3 labview project for data acquisition

host machine. The architecture of our measurement tool is shown in **Fig. 2**.

In monitor machine, there are three components: labview connector, data processor and communication manager. The labview connector is used to control the acquisition and collection of currents and voltages of GPU card. The data processor is used to remove the useless data(As the data acquisition starts before the CUDA kernel lanuch, there exists some useless data which should be removed). The communication manager is used to receive control signals from host machine and transfer power dissipation data to host machine.

In host machine, there are also three components: communication manager, database manager and user APIs. The communication manager is used to send control signal to monitor machine and receive power dissipation data from monitor machine. As one database is set to store all the power dissipation data, the database manager is used to manage the data storage and apply database access inferface for users. The user APIs is used to apply interface for users to control measurement task.

4. Design and Implementation

4.1 Data acquisition and process

To measure the currents and voltages of GPU, we use three FLUKE i30s current probes[9] and three YOKOGAWA 700925 voltage probes[10] to connect to the

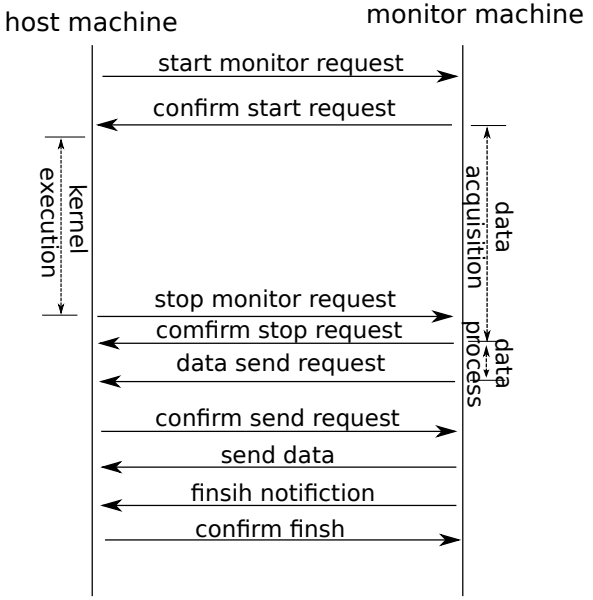

Fig. 4 The communication process between host machine and monitor machine

power lines. All the probes are connected to a NI USB-6259 BNC device[11] which is a USB high-performance data acquisiton module optimized for superior accuracy at fast sampling rates. The device connects with the monitor machine by USB and will send all the acquisited data to the monitor machine. To recieve and process the data from NI device, a labview project is set up as illustrated in **Fig. 3**. The labview project includes a DAQ Assistant and a write to measurement file module. The former module receives the data from NI device and the latter module write the data into local files. This process is continuous and controlled by a switch. We encapsulate the labview project into a DLL(Dynamic Link Library) which can be called from C or C++ program. By this way, we can control the start and stop of the data acquisition from program.

Here we face one problem is that not all the received data is useful. Because the data acquisition time must be longer than the CUDA kernel execution, so there must be some unwanted data written into the local files. These unused data

```
//open database connection
int dbOpen(char* dbName);
//read records by task ID
int dbReadByID(int ID);
//read records by task name
int dbReadByTaskName(char* name);
//qury all the user name in database
int dbGetUser();
//qury all tasks information of user
int dbGetTaskByUser(char* user);
//delete records by task ID
int dbDeleteByID(int ID);
//delete records by task name
int dbDeleteByTaskName(char* taskName);
//delete records by user name
int dbDeleteByUser(char* user);
//close database connection
int dbClose();
```

Fig. 5 The database access interface

should be removed. To identify the CUDA kenrel, we design a marker with two kernels with different intensities of memory accesses. As a kernel consumes higher power when accesses memory intensively[12], the marker will present a regular up-down power dissipation line. Therefore, we can add this marker before and after the execution of measurement kenerls. After acquisiting all the data, we just need to keep the data between the two markers.

4.2 Communciation between host machine and monitor machine

The communication between host machine and monitor machine can be classified into two types: control command and data transmission. The control command is used to control monitor start and stop while the data transmission communication is used for the data transmission between two machine as illustrated in **Fig. 4**.

When users want to measure the power disspiation of their applications, first of all, a *start monitor request* message will be sent to monitor machine from host machine. Then, the monitor machine will start the monitor process and return a *confirm start request* message to host machine. With the received *confirm start request* message, the host machine will launch the CUDA kernel. When

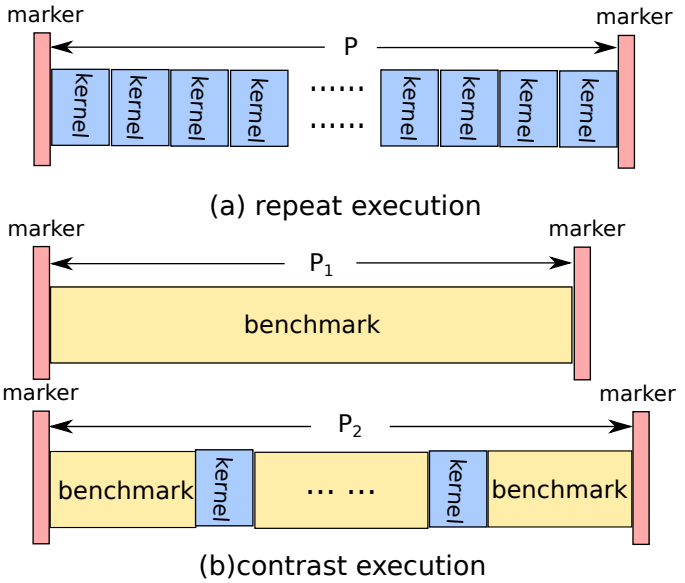

Fig. 6 Measurement of Tiny Kernel

the CUDA kernel execution is over, the host machine will send a *stop mointor request* message to monintor to stop the data acquisition and return one *confirm stop request* message. As the monitor process begins before the CUDA kernel execution process and is over after the CUDA kernel execution process, therefore, parts of the collected data is unuseful and should be removed. Thus, the monitor machine will take for a while to analyse and process with the data. After this, the monitor machine will send a *send data request* to host machine to set up a data transmission connection. When the host machine receives the message, it will receive the data transmission connection and return a *confirm send request* to monitor machine. Then, the mointor machine begins to transfer the data to the host machine. After sending all the data, the monitor machine will send a *finish notification* message to the host machine. With the message, the host machine will close the data transmission connection and send back a *confirm notification* message to monitor machine. After receiving the message, the monitor will close the connection and the measurement task is over.

4.3 Database manager

We use database in host machine to store all the power dissipation data from monitor machine. As in our case, the total data size is not very big and we hope the data can be easily shared by users on different machines. Therefore we choose a light-weight open source database SQLite[8] to store the data. SQLite is a software library that implememts a self-contained, serverless, zero-configuration SQL database engine. It create one file for each database and this database file can be moved and used by any machine with SQLite installed. In our case, we set up one database called power.db with one table named *data*. The table includes twelve attributes as shown in **Table 1**. The primary key is formed by *taskID* and *seqNum*.

To apply an easy access to the database, we set up a set of APIs as shown in **Fig. 5**. We provide interfaces to open and close database connection. For read operation, we also provide interfaces to read records by specified attribute such as taskID, taskName. As each user can have many tasks, we provide interfaces to get the relationship information between users and tasks. The function *dbGetuser()* will print out all the user names in the database and the function *dbGetTaskByUser* will print out all the task ID of the specified user name.

4.4 Tiny CUDA kernel solution

In our measurement system, there are a lot of deviation factors such as communicaton delay, marker identification to affect the accuracy. Especially for the CUDA kernels with very short execution time, it is very difficult to precisely measure its power dissipation without any auxiliary method. There are two methods to measure the power dissipation of such tiny CUDA kenrel as shown in **Fig. 6**.

Table 1 The data table of power dissipation

attribute	description	data type
taskID	task'ID	integer
taskName	the name of task	text
user	the user name of task	text
power	the power dissipation	real
current1	the current of PSU power line	real
current2	the current of 3.3V power line in PCI-Express bus	real
current3	the current of 12V power line in PCI-Express bus	real
voltage1	the voltage of PSU power line	real
voltage2	the voltage of 3.3V power line in PCI-Express bus	real
voltage3	the voltage of 12V power line in PCI-Express bus	real
taskTime	the task lanuching time	text
seqNum	the ID of record within one task	integer

The first method is repeat execution as shown in Fig. 6.(a). In this method, the tiny kernel will be executed repeatedly by n times and we can get the total power dissipation P. Then we can calculate the power dissipation of one single tiny kernel by the following equation:

$$P_{tiny} = P/n. \quad (1)$$

P_{tiny}: the power dissipation of the tiny kernel;
n: the number of repeated execution of the tiny kernel;
P: the power dissipation of n tiny kernels.

Although this method is simple, the accuracy of result can be very low. Because the first execution situation of the tiny kernel can be different of the latter execution situation which will cause greate different in the result. For example, when running the tiny kernel in the first time, there is no cache for the data. However, as the tiny kernel is very small, all the required data may be cached in the following repeated executions. With data cached, the execution time will reduced a lot and the power consumption of cache access is less than global memory access. In this case, the measured result will be much smaller than the actual result with this method. Therefore, we give up this method.

The second method is contrast execution as shown in Fig. 6.(b) which is optimized basing on the first method. To test the tiny kernel more accurately, we combine the tiny kernel with the benchmark that we designed to execute. First of all, we run the benchmark and measure the power dissipation P_1. Then, we insert the tiny kenrel into the benchmark for m times and keep enough distance between tiny kernels to avoid cache as much as possible. Running the modified benchmark, we can have the power dissipation P_2. Then, we can use the following equation to calcuate the power dissipation of the tiny kernel:

$$P_{tiny} = (P_2 - P_1)/m. \quad (2)$$

P_1: the power dissipation of the benchmark without tiny kernel embedded;
P_2: the power dissipation of the benchmark with tiny kernels embedded;
m: the number of tiny kenrels embedded in the benchmark.

Comparing to the first method, the contrast execution can reduce cache rate a lot. Ideally, we hope there is no cache before the start of each tiny kernel,

```
//marker kernel;
int P_marker();
//send start monitor request;
int P_startMonitor();
//send stop monitor request;
int P_stopMonitor();
//wait for data
int P_waitData();
//receive data
int P_receveData(int sock_fd, struct record* rec);
//store data into database
int P_storeData(struct record* rec);
// half-half benchmark
//int P_half_test(int time);
//computing intensive benchmark
int P_compute_test(int time);
//memory access intensive benchmark
int P_memory_test(int time);
//return error imformation
int P_getError();
```

Fig. 7 User API

the first execution of the tiny kernel can be repeated. Therefore, the design of the benchmark becomes a key part. Here, we design one benchmark with half computing and half memory access to make the power dissipation average. And the memory access within the benchmark are decentralized which can greately reduce cache hit rate. Besides, we also provide one computing intensive benchmark and one memory access intensive benchmark to let user desgin their own benchmark.

4.5 User APIs

Besides the database access interface, we also provide a set of APIs for users as shown in **Fig. 7**. We provide *P_startMonitor()* and *P_stopMonitor()* functions to control the monitor process in monitor machine. Function *P_waitDate()* is called to wait for the data send request from the monitor machine. Once receiving the request, this function will set up a socket connection between the host machine and monitor machine for data transmission and return the socket handle. With

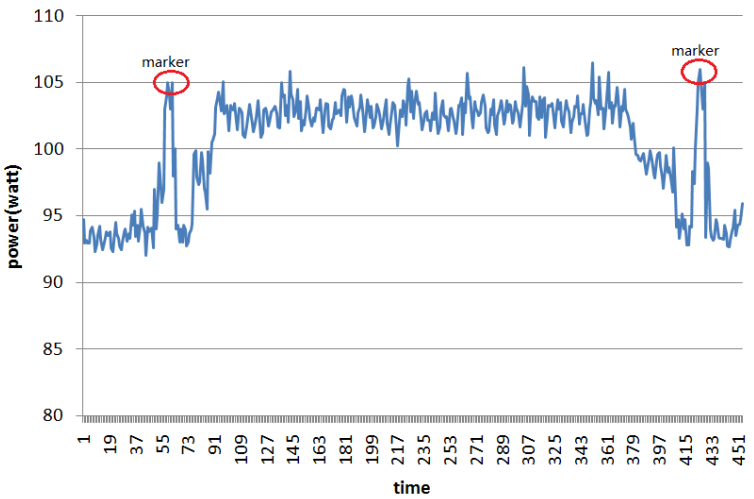

Fig. 8 Power dissipation of Matrix benchmark with 16 warps

the socket handle, function *P_receiveData()* will receive the data. All the data will be store into a struct array. If users want to keep these data, they can call the function *P_storeData()* to store them into database for future usage. To idenfiy the result of measured kernel, users need to call function *P_maker()* before and after launch kernel. For tiny kernel solution, we provide three benchmarks: *P_half_test()*, *P_compute_test()* and *P_memory_test()*. For all functions, we write error messages into a shared buffer. Users can call *P_getError()* to print out the error information.

5. Experiments

To test the performance of our tool, we use matrix multiplication benchmark. We set the size to be 256×256 and run the benchmark with 16 warps. The power

Table 2 The time overhead of each part

warp number	1 warp	4 warp	16 warp
data process(ms)	172	115	83
data transmission(ms)	102	67	41
data size(KB)	213	142	93

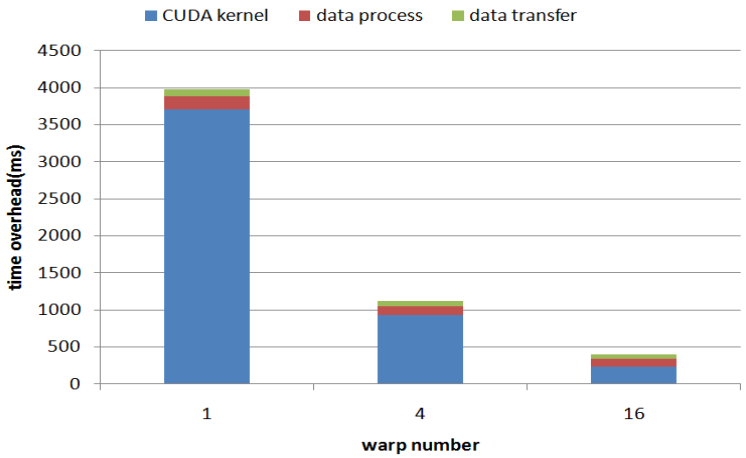

Fig. 9 time overhead of each part

dissipation is shown in **Fig. 8**. As we have inserted one marker benchmark before and after the measured kernel, we can easily find two markers in Fig. 8. Therefore, the data between the two marker is the actual power dissipation of the measured kenerl.

To test the overhead of our tool, we repeatedly run the matrix benchmark with different warps. First of all, we measure the execution time of the CUDA kernel in the host machine. To measure the execution time of the CUDA kernel, we create two events. We start one event(*start*) before launch the kenerl and start another event(*stop*) after the function *cudaThreadSynchronization()* which is used to wait for result from GPU device. By calling the function *cudaEventElapasedTime(stop, start)*, we can get the execution time of the CUDA kernel.

Besides the execution time of CUDA kernel, there are mainly two time overhead in the whole measurement process: the time overhead of data process and the time overhead of data transmission. The data process is responsible to identify the marker from the power dissipation result. We call the function *clock()* to record the start time before the process. Then we call the function *clock()* after the process to record the stop time. Then we can get the ime overhead of data process by calculating the difference of the two time records. To test the time overhead of data transmission, we call the function *clock()* to record the start time before the monitor machine sends data to the host machine. Then the monitor machine sends data to the host machine and waits for an confirmation message from the host machine. After receive the comfirmation message, we call the time function again to record the stop time. With these two time slot, we can get the time overhead of data transmission.

As illustrated in **Fig. 9**, the time overhead of our tool is very small. The details of the time overhead are shown in **Table 2**. We can find that the time overhead of the data process and data transmission decrease along with the increasement of warp number. This is because the size of result data decreases when running with more warps. However, the time overhead of the data process will not always decrease with the increasement of warps. As there is a minimum monitor time for the NI device, so the size of result data from the hardware will can not be smaller than a fixed size.

6. Summary and Future Works

In this paper we have proposed a precise measurement tool for power dissipation of CUDA kernel. Our tool provide an easily API layer to achieve precisely measurement on the power dissipation of their GPU application and good management on the obtained data. With these, users can have a better undertstand in the power dissipation of their application which can help them improve the applicaton.

Power measurement and optimization of CUDA kernels is parts of our research. We are considering further level of power reduction by efficient global memory mangement in GPU and efficient schedule mechanism in multiple GPUs. Especially, latest CUDA veresion provide peer-to-peer communication and unified virtual addressing which enable more improvement in both performance and energy efficient. Therefore, we will analyze the performance of different application on multiple GPUs and try to carry out optimization mechanism. Basing on this, we will design a user-friendly CUDA programming APIs for multiple GPUs programming.

Acknowledgments This work is partially supported by Core Research of Evolutional Science and Technology (CREST) project "ULP-HPC: Ultra Low-

Power, High-Performance Computing via Modeling and Optimization of Next Generation HPC Technologies" of Japan Science and Technology Agency (JST) and Grant-in-Aid for Scienific Research of MEXT Japan.

References

1) A.Nukada, Y.Ogata, T.Endo and S.Matsuoka "Bandwidth Intensive 3-D FFT kernel for GPUs using CUDA", *Proceedings of SC008(electronic),11 pages, 2008.*
2) Reiji Suda, Da Qi Ren. "Accurate Measurements and Precise Modeling of Power Dissipation of CUDA Kernels toward Power Optimized High Performance CPU-GPU Computing",*PDCAT 2009, Proceedings of international conference on Parallel and Distributed Computing, Applications and Technologies*, Higashi Hiroshima, Japan, pp.432-438.
3) NI link,available from ⟨http://www.intel.com/products/processor/corei7ee/⟩ (accessed 2012/1/20).
4) NVIDIA Corporation: *CUDA Programming Guide*, Version4.0.
5) NVIDIA Gefore series GTX 260, available from ⟨http://www.nvidia.com/gefore⟩ (accessed 2012/1/20).
6) NVIDIA Fermi architecture, available from ⟨http://www.nvidia.com/object/fermi_architecture⟩ (accessed 2012/1/20).
7) LabVIEW: System Design Software, available from ⟨http://www.ni.com/labview/⟩(accessed 2012/1/20).
8) D.R.Hipp et al. SQLite, available from ⟨http://www.sqlite.org/⟩ (accessed 2012/1/20).
9) FLUKE i30s AC/DC Current Clamp, available from ⟨http://www.fluke.com/fluke/usen/Accessories/Current-Clamps/i30s.htm?PID=56297⟩(accessed 2012/2/15).
10) YOKOGAWA 700925 15 MHZ DIFFERENTIAL PROBE, available from ⟨http://tmi.yokogawa.com/products/oscilloscopes/voltage-probes/700925-15-mhz-differential-probe/⟩(accessed 2012/2/15).
11) NI USB-6259 BNC, available from ⟨http://sine.ni.com/nips/cds/view/p/lang/en/nid/209150⟩ (accessed 2010/1/20).
12) T. Nishikawa and T. Aoki, personal communication, 2008.