CUDA’s Mapped Memory to Support I/O Functions on GPU

Wei Wu
Jiangnan Institute of Computing Technology, Wuxi 214083, China

Fengbin Qi
Jiangnan Institute of Computing Technology, Wuxi 214083, China

Wangquan He
Jiangnan Institute of Computing Technology, Wuxi 214083, China

Shanshan Wang
Jiangnan Institute of Computing Technology, Wuxi 214083, China

Follow this and additional works at: https://tsinghuauniversitypress.researchcommons.org/tsinghua-science-and-technology

Part of the Computer Sciences Commons, and the Electrical and Computer Engineering Commons

Recommended Citation
Wei Wu, Fengbin Qi, Wangquan He et al. CUDA’s Mapped Memory to Support I/O Functions on GPU. Tsinghua Science and Technology 2013, 18(6): 588-598.

This Research Article is brought to you for free and open access by Tsinghua University Press: Journals Publishing. It has been accepted for inclusion in Tsinghua Science and Technology by an authorized editor of Tsinghua University Press: Journals Publishing.
CUDA’s Mapped Memory to Support I/O Functions on GPU

Wei Wu*, Fengbin Qi, Wangquan He, and Shanshan Wang

Abstract: The API interfaces provided by CUDA help programmers to get high performance CUDA applications in GPU, but they cannot support most I/O operations in device codes. The characteristics of CUDA’s mapped memory are used here to create a dynamic polling service model in the host which can satisfy most I/O functions such as read/write file and “printf”. The technique to implement these I/O functions has some influence on the performance of the original applications. These functions quickly respond to the users’ I/O requirements with the “printf” performance better than CUDA’s. An easy and effective real-time method is given for users to debug their programs using the I/O functions. These functions improve productivity of converting legacy C/C++ codes to CUDA and broaden CUDA’s functions.

Key words: CUDA; I/O functions; mapped memory; dynamic polling service model

1 Introduction

Parallel applications for GPU can be easily developed using CUDA with the various API interfaces provided by CUDA providing the powerful tools to manage the GPU and the high memory bandwidth. Besides traditional image processing, CUDA has also enabled analyses at oil reconnaissance, astronomical timing, hydrodynamics, molecular kinetics, biology, audio frequency decoding, and video frequency decoding. The number of applications using the GPU have been accelerated many fold, even a hundred times more than in a CPU[1] with key run-time libraries such as CUBLAS, CUFFT, and CUDPP[2].

However, the API interfaces and libraries in CUDA are self-contained, so most of the I/O functions are not supported in the device (referring to the GPU and its memory) codes, so developing and debugging application is difficult. For example, developers often use “printf” in debugging to access application information. Although CUDA supports “printf” in device codes in version 3.1[2] and higher, users cannot see the results of “printf” in real-time until the kernel function is finished, which is not satisfactory. If the kernel function hangs, the users will not get any of the “printf” information.

Therefore, the I/O functions are not convenient for programmers. Especially, the system will spend extra energy dealing with legacy codes that contain I/O operations. For example, the only method to complete file read/write operations in the device is to add some memory copies between the host (referring to the CPU and the system memory) and the device instead of directly reading/writing files.

Most research on CUDA has been based on existing programming models and compilers that are legacy codes, not CUDA programs. Generally, these studies have changed the programs written in other programming languages to CUDA or to executable target codes on the GPU directly. For example, Lee et al.[3] designed a source-to-source compiler which can change OpenMP program to CUDA and used different program optimization methods in OpenMP and CUDA. HMPP[4] used compiling directives to translate C or FORTRAN programs to CUDA or OpenCL[5] program. PyCUDA[6] allows programmers
to directly use CUDA’s parallel compute APIs in Python\textsuperscript{[7]} codes. CuPP\textsuperscript{[8]} integrates CUDA programs into an existing C++ framework. All these try to make the GPU programming easier and more effective. These tools can quickly transplant legacy codes to the GPU. Therefore, CUDA needs to also support I/O functions and the other common functions in these codes so as to not increase the designers’ workloads and to reduce the compiler’s applicability.

This study uses the characteristics of the mapped memory to support I/O functions, such as reading/writing files and “printf” in device codes. These efficient I/O functions enable application developers to conveniently get data at run time to further enable transplanting legacy codes. At the same time, the I/O agent introduced here can be extended to memory operations, message sending and receiving, and socket operations, even for entire web server.

This paper introduces CUDA and GPU, and the challenges faced when implementing efficient I/O functions in GPUs with descriptions of the implementation ideas and techniques. Examples are given using CUDA SDK 4.0\textsuperscript{[9]}.  

2 Background on GPU and CUDA Architecture

The GPU is built around a scalable array of multithreaded Streaming Multiprocessors (SMs) where each SM consists of a set of Streaming Processors (SP) cores which provide the huge computing ability of the GPU. Execution of a CUDA program uses kernels that execute on the device and a C program that executes on the host. The host program defines the context for the kernels and manages their execution. A kernel can be executed by multiple equally shaped blocks that are organized into one-dimensional, two-dimensional, or three-dimensional grids. A block is formed from a number of CUDA threads and each thread may use I/O functions so that multiple I/O operations in different threads must execute accurately and in order.

Figure 1 shows an overview of the CUDA memory model. The memory hierarchy of a CUDA device has several parts including the global memory, constant memory, shared memory, texture memory, and local memory. Each thread has a private local memory and each thread block has shared memory visible to all threads of the block with the same lifetime as the block. All threads can access the same global memory\textsuperscript{[10]}. The constant memory and texture memory are read-only memory spaces accessible by all threads. Host memory spaces can be divided into pageable memory and page-locked memory (also called pinned memory)\textsuperscript{[11]}. On devices with compute capability greater than 1.0, CUDA has extended the function of pinned memory so that a block of pinned memory spaces can be mapped into the address space of the device, eliminating the need to copy to or from device memory. Such a block is called mapped memory which in general, has two addresses in host memory and in device memory that can be used to access the block for a kernel. The mapped memory is used here for efficient I/O on the GPU. However, the system must ensure the order and coherence of operations to the same mapped memory spaces because of the features of mapped memory, which is a big challenge.

In addition, function arguments in device codes written by users are quite restricted, so they can not satisfy the demands of some I/O functions like the format parameters in “printf”. Thus, future implementations need new methods to deal with this problem. On recent GPUs such as Fermi\textsuperscript{[11]}, the global memory has a data cache that leads to the cache coherence problem, which also complicates I/O function design.

3 Design and Implementation

3.1 Overview

Figure 2 shows the basic framework of implementation of the I/O functions, which includes a preprocessor module, a support library of I/O functions, and a host agent module. The preprocessor inputs are GPU codes
containing I/O functions which require processing and code generation work to provide the arguments for the device functions. The I/O function device library provides the required APIs. These APIs record the I/O function data and parameters, send agent requests, complete interactions with the host, and finally get the return values. The host calls pthread_create to create a process to do the agent job. The process denoted by Agent_tid scans the data structures stored in mapped memory to record information for the agent requests. When the host detects a request, the APIs in the host are called to deal with the agent’s request.

3.2 Dynamic polling service model

This process uses the host’s agent to complete the process using the characteristics of the mapped memory. A dynamic polling service model\textsuperscript{[12]} is used in the host. As shown in Fig. 3, the polling system consists of a single server that polls a number of queues. The component parts of this system are

1. Input process

The client sources, namely, the I/O proxy requests are infinite. The stochastic and substantive processes running at the client occur one request at a time.

2. Queue structure

The model contains a number of queues, which have an infinite length and store I/O request information in mapped memory.

3. Services organization

There is only one server that is one process in the host that deals with all the I/O requests. The service time is stochastic and substantive, and related to the mode (the kind of I/O operation) and the service content (the I/O data).

4. Queue rule

The polling system queue rule includes three factors. The first factor is the order in which the queues are served which is a dynamic process here. The second factor is related to how many customers to serve during each visit to a queue. Many strategies have been proposed such as the exhaustive, gated, and Limited-\(K\) disciplines. This method uses the Limited-\(K\) discipline with \(K\) equal to one. The last factor is the order in which customers within each queue are served, with this system using a simple scheduling policy called the First-Come-First-Served (FCFS) policy.

The Agent_tid process scans each queue at regular intervals, called polling cycles. If a client request exists in a queue, the process provides the appropriate proxy service for the kind of request. After serving this request, the process then scans the next queue. The dynamic characteristics of the polling model are related to the queue scanning order and the determination of the polling cycle time.

The scanning order records the position of the queue containing the last served client with a completed polling process as the start position for the next polling time. The reason for this is that user requests have locality, since multiple sequential I/O requests may be sent out by the same CUDA thread.

The algorithm to determine the polling cycle time is illustrated in Fig. 4. The total execution time for a polling process is used to compute the rate for processing the polling cycles. If the rate changes
slightly within a range, then the number of customer requests is also changing slowly and the polling cycle setting is reasonable. However, if the number of the customer requests is changing rapidly, then the polling cycle time should be changed. If the rate increases, the polling cycle time is decreased to improve the I/O request processing. The polling cycle time is increased to reduce the resource waste when the rate decreases. The model also provides a setting to change the width of the polling window when many rate changes occur to keep the polling cycle time short. This method reduces the negative influence of sudden changes in the number of I/O requests, but also reduces the system sensitivity to make later changes with polling cycle time.

Thus, this model dynamically adapts to the requests to process I/O proxy requests in the device. Moreover, the model is scalable which is important when adding new proxy services. If CUDA had supported an interrupt mechanism, the device could simply send out an interrupt signal to enable the interrupt process.

### 3.3 Determining the number of queues

The simplest way to determine the number of queues is to use the number of total threads as the number of queues and then process each thread in a fixed queue. This method results in no competition between threads, which then avoids misses or data errors in the I/O operations. However, this also results in a huge required storage space since there are many threads.

Another solution is shown in Fig. 5a, which first sets the number of queues equal to the maximum number of threads that run simultaneously on a device. If the number of total threads is less than the maximum, the number of queues is set to the smaller value. For the Tesla C2050 with fourteen SMs where each SM can simultaneously execute a maximum of eight activity blocks, the maximum number of threads is 1024 in each block. Therefore, the number of queues in this GPU is $14 \times 8 \times 1024 = 14336$ at most. The modulus of the number of global threads and the number of queues is used to get the right number of queues. This solution significantly reduces a mount of required space, but has some less obvious difficulties. In CUDA’s execution mode, when one SM implementation is far behind the other SMs, these faster SMs will dynamically receive another patch of blocks. These new blocks would have been mapped into the queue used for the preview SM, which may result in some lost or hanging I/O operations. In most situations, the workload in each SM is balanced, but this assumption is not conservative.

The problem with the solution in Fig. 5a is that it does not address the competition for the same queue resource between threads. This competition will be more severe as the number of queues decreases. This problem can be addressed by a lock operation mechanism among threads based on atomic operations. This mechanism, which provides an exclusionary algorithm for many threads to access the same queue, is shown in Fig. 5b. A thread first gets a queue ID after sending out an I/O service request and then uses atomic operations to access the queue lock in this queue to fetch a service ticket and the current service ID. This thread then enters the stage that waits for the locking. The queue then periodically gets the current service ID of this queue and judges whether the service ID is equal to its own ticket. While the equation is satisfied, the request has been accepted and this thread enters the lock stage,
which exclusively processes the I/O operation. After processing, the unlock operation adds one to the service ID and waits for the next request.

The number of locks can be different than the number of queues, with one actually being enough. However, the competition to get tickets will be more intense with less locks, which will affect the implementation efficiency. This complex mechanism with the queue lock would not be necessary if CUDA could provide a run-time API to get the SM ID for each activity block.

### 3.4 Handshaking

The previous section described the lock mechanism used by the device threads to access the I/O queues. Another handshaking mechanism is then used for interactions between the host and the device.

Efficient interactions are needed to process the agent requests. Handshaking is then used to maintain the order and coherence of operations in each mapped memory space, like a network protocol to complete interactions. A buffer queue is used to deal with each thread’s I/O demands with all the requests of one thread processed in the same buffer. Handshaking is then used to maintain the order of multiple requests.

Figure 6 shows the handshaking process, which is composed of five steps, initialization, certifying the connection, data transport, agent processing, and value return.

1. **Initialization:** This creates the interactive process Agent_tid and initializes the buffer queue. Agent_tid calls the request-scanning function and then the host sends the Host_hello message to the device and waits for the I/O agent requests.

2. **Connection certification:** After sending the I/O function to the device, the APIs in the I/O function support library provide the necessary support. When an agent process starts, the system must certify whether the connection is available. The device sends the Certificate_request message and waits for certification from the host. When the device receives the Certificate message sent by the host, the signal state in the host (host_num) is set equal to the signal state in the device (thread_num). Thus, the handshaking is complete and all the requests in this connection are processed. The host can now send the current request.

3. **Data transport:** The device sends the Data_pack message that contains the thread ID, proxy mode, and parameters. In the next step, the device sends the Agent_request message and waits for the host reply.

4. **Agent processing:** Once the Agent_tid detects an agent request, it runs a pocket distribution process according to the proxy mode field in the data pack and transfers the data pack to the corresponding I/O operation API in the host to process. The host sends an Agent_finished message when the agent process has finished.

5. **Value return:** The device successfully handshakes with the host again by receiving the Agent_finished message and returns the results after getting the return values in the interactive buffer structure.

### 3.5 Preprocessor and argument parsing

Because of the limitation on the available arguments, an I/O operation request may need multiple sentences. Therefore, this algorithm uses a simple preprocessor which scans the input CUDA codes, recognizes the I/O operations needing transformation, and outputs the proper code. An example for “fprintf” is shown in Fig. 7 to describe the particular processes executed by the preprocessor. First, a function call to _fprintf_begin is generated to transport a format argument and a file pointer. Second, a parser parses the format argument and generates multiple evaluation sentences to identify the parameters according to the parsed data type. Finally, a function call to _fprintf_end is generated to send the agent requests, wait for the reply, and return the values. The file pointer is stored as a long type in the device and for transport to the host.
because the device does not support the FILE pointer type. Besides that, the user program can use all normal I/O functions while the preprocessor does transform the codes.

The agent process in the host also needs to parse the format arguments received in the data pack using a similar parsing process to that in the preprocessor. The agent parses the data type one at a time, then gets the data from the corresponding position in the buffer queue and transforms the data according to the parsing type. After that, it calls the host’s I/O functions to deal with this data and organizes the return values.

3.6 Fetch signal states with atomic operations

In early GPUs, the devices used the simple method shown in Fig. 8a to get the state of host_num in the handshaking process. After updating the host_num made by the host, the mapped memory can be used for the implicit copy from host memory to device memory. With the mapped memory, the device can receive the Certificate message or Agent_finished message sent by the host while directly accesses the memory to get the state of host_num. However, in new GPU architectures, such as Fermi, a data cache is used in the device so the original method can not be used. As shown in Fig. 8b, the implicit copy operation of host_num is not in a guaranteed area of the original cache coherence protocol. Sometimes, when the device reads the host_num and gets a cache hit, it reads an old state so the handshaking is not finished. In this situation, the program will wait for the host’s reply message.

A solution to this problem is presented in Fig. 8c. An atomic get operation is used instead of the original fetch operation. The operation uses a CUDA atomic add operation\cite{13} to fetch the signal states, with the actual command being atomicAdd(lamp, 0). Since that atomic operation directly accesses memory and avoids duplicate data in cache, the handshaking process is done successfully. However, the problem shown by the dashed line in Fig. 8c still exists. If the device reads the state of host_num while the implicit copy is not done yet, the fetched data will be for the old state. After the atomic get operation, the data will be stored back to memory and the state of host_num in the host will be set to the old state through the implicit copy from device to host. The host then needs to repeat the I/O request that has already been done before.

This problem can be addressed by the solution shown in Fig. 8d. A new state field is set that is only used in the host called host_num_only. This field represents the correct state of host_num at the current time. Another handshaking process is used for host_num_only to control the agent calls for the I/O operations, with a copy of the state of host_num_only to host_num to make sure that the state of host_num is correct.

This resolves the cache coherence problem. The write updating of host_num in the host can also transfer the information to the data cache in the device. Thus, software is used to maintain the cache coherence protocol in the device.

3.7 Short handshaking

The “fwrite” and “fread” operations need a better handshaking process. Since the number of data bytes to be written or read is not fixed and the storage buffer can not be infinitely large, the operations must be processed in batches. Each batch of data then needs interactions between the host and the device so a short handshaking
process is implemented below the original handshaking process. Figure 9 shows the process with “fwrite” as an example to describe the process.

As with the regular handshaking, the initialization and connection certification are done first. During the data transport stage, Data_pack message will contain information such as the thread ID, proxy mode, file pointer, and two other arguments called “size” and “count” for the “fwrite” operation. A value representing the number of requested short handshaking processes will also be put into the data pack. Need_copy_times is computed from “size”, “count”, and the buffer spaces to write one file at a time. After the device sends the Agent_request message, the device enters the short handshaking process and will iterate the number of times indicated by need_copy_times. Each time does following jobs:

1) Buffer certification: First, the system must certify that the write buffer is available. The device sends the Certi_buf_request message and waits for certification by the host. Once the device receives the Certi_buf message sent by the host which indicates that the short handshaking is completed and the previous write requests in the buffer are processed, the current request is sent.

2) Write buffer: The device sends the Data_pack_buf message that carries the data needed for the write in the current batch. The data is stored in the corresponding position in the interactive queue. Then, the device sends the S_Agent_request message and waits the reply from the host.

3) Small agent processing: Once the Agent_tid detects a small agent request, it fetches the data in the current batch and calls the “fwrite” operation in the host to complete the file write. The host sends an S_AgentFinished message when the small agent process is completed.

4) Evaluating the number of processes: The device has successfully completed the short handshaking when it receives the S_AgentFinished message. Then, the system must decide whether another small agent process should be started according to the value of need_copy_time.

After finishing all the iterations, the host sends the AgentFinished message to tell the device that all the agent processes for this “fwrite” operation are completed. The device can return values after receiving this message.

4 Test Results

4.1 Test environment

The effectiveness of the I/O operations was tested on a NVIDIA Tesla C2050 GPU with an Intel Core2 Duo E7500 CPU with the Red Hat Enterprise Linux Server release 5.3 operating system and nvcc 4.0 CUDA compiler.

Table 1 shows the kernel configurations for representative programs for CUDA SDK 4.0. The maximum number of queues was set to 8192, so that, the total number of queues was the smaller of 8192 and the total number of threads. The user can set this number based on their needs to find the best tradeoff between space and time. The number of queue locks to prevent competition between threads was the same as the number of queues. The tests evaluate the dynamic polling service model and the file and print operations.

4.2 Dynamic polling service mode tests

This section illustrates the performance improvements by the dynamic polling service model. The first tests evaluate the influence of the dynamically determined polling cycle time using a dynamic queue scanning order, a smooth polling window width of one, the error

<table>
<thead>
<tr>
<th>Table 1 Test contents and kernel configurations.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SDK</strong></td>
</tr>
<tr>
<td>BlackScholes</td>
</tr>
<tr>
<td>Dxtc</td>
</tr>
<tr>
<td>convolutionFFT2D</td>
</tr>
<tr>
<td>matrixMul</td>
</tr>
<tr>
<td>Reduction</td>
</tr>
</tbody>
</table>
for updating the polling cycle was 10% and the polling cycle was allowed to change by 10 ms at each time. The test example used a kernel with 256 blocks and 1024 threads within each block. Each thread calls “printf” to print the thread ID and the characters “HELLO WORLD!” ten times.

The test results are shown in Fig. 10 where the x-axis represents the initial polling cycle time and the y-axis is the kernel execution time. The three curves show the execution time without polling, with dynamic polling cycle time and with fixed polling cycle time. The execution time with dynamic polling increases very little, and is consistent with the nonstop method using no polling cycle time. The execution time using fixed polling cycle time increases almost linearly. Thus, the dynamic method is much better with the kernel efficiency improved an average of 3.08 fold over the fixed method. When the initial polling cycle is 100 ms, the efficiency with dynamic polling is improved 4.42 fold. The dynamic method is also sometimes better than the nonstop method because the host scans for requests at times without polling cycles, which will influence the original implementation. The influences are more obvious with heavy workloads, when the dynamic method would be much better.

The polling window width test results are shown in Fig. 11. The dynamic method for determining the polling cycle time was used in the tests with an initial polling cycle of 10 ms. All the other settings were the same as in the previous test except that the tests had different numbers of print threads as indicated in the legend in each block. The tests show that:

1. The polling efficiency is improved by the proper window width with a heavy printing load, especially for T256 and T1024. T1024’s execution time was reduced by only 18% when the width was increased to four from one. The reasons are that loss resources are wasted in the host, and polling cycle time will not be added when the requests are suddenly reduced.

2. The window width increases where the performance deteriorates with more print requests. For example, the T1 execution time with width four is 7.54 times that of width one, while the width where T1024 deteriorates is eight because the number of requests changes slowly in small print tests, which does not result in polling cycle changes. The wider width impairs the system efficiency in this situation, so the efficiency improves as the polling cycle becomes shorter.

3. The changes in execution time are smoother for larger widths and closer to the execution time without polling cycles. For example, the T1 execution time is basically unchanged for widths larger than four and within 2% of the time for no polling.

Thus, users using this model can also decide a proper window width based on the I/O characteristics in their programs.

The final test evaluates the method used to determine the polling queue scan order. The dynamic method was used to determine the polling cycle time with the smooth polling window width set to one. All the other settings were the same as in the previous tests. The results in Fig. 12 show that the dynamic polling order gives a 12% performance improvement relative to the static method. The improvement is 23% when only one thread sends out I/O requests in each block. However, when more threads need to be serviced, the thread localization of the service requests is not obvious so the dynamic method is not as good. For example, the performance improvement is only around 1% in the
tests with T64, T256, and T1024.

The next tests of the file and print operations used the dynamic polling service with the dynamic queue scan order, dynamic polling cycle, an initial polling cycle at 10 ms, a smooth polling window width of one, updating at 10% errors and polling cycle changes of 10 ms.

4.3 File operation efficiency

The file tests for SDK programs included the operations “fprintf”, “fread”, “fwrite”, “fgets”, and “fputs”. Each file test also included one “fopen” and “fclose”. These operations were executed once in each block. The “fprintf” operation printed the block ID and a character string, the “fwrite”/“fread” write/read lengths were 1 KB, the data transfer in each short handshaking was 1 KB, and the “fgets”/“fputs” character string was “HELLO WORLD!”.

Figure 13 shows the implementation results for the file operations with the SDK test program on the x-axis and the execution times for the file operations in addition to the original execution times on the y-axis. The results show that the “fread” or “fwrite” operations gave relatively large increases in the execution times with increases of 17.0% for “fread” and 18.8% for “fwrite”. The average increase for all five file operations was 12.25%. Thus, the tests show that these methods correctly and effectively implement the file functions, which are not supported by CUDA.

4.4 Print operation tests

This section compares the performance of current print method and CUDA’s print operations. The print tests added to the original SDK programs had each thread block print its ID and a string.

The “printf” operation results are shown in Fig. 14 which compares the execution times for the “printf” operation on the y-axis for CUDA and the present implementation. The present “printf” is faster than the CUDA method with an increase on the execution time using the present method of only 7.5%. The tests for the present implementation are on average 6% faster than the CUDA implementation for “printf”.

Figure 15 shows the influence of thread size on the original SDK programs with the two print
implementations. There are no results for large thread sizes because the kernel configurations differ. The efficiency decreases in both methods with more print requests. When huge numbers of threads do many I/O operations, the extra execution time increases greatly. For example, the execution time of the CUDA “printf” rises rapidly for the BlackScholes program with the T128 test, 29.58 times longer than the origin program with the current “printf” method is only 6.73 times longer. Thus, these test results also show that this model is more efficient than CUDA with better results as the number of requests grows increases. In the T128 case for the BlackScholes program, this implementation is 4.39 times faster than CUDA. In the T64 case for the dxtc program, the speed up is 1.67 times.

Therefore, the current implementation is more efficient and effective. Large applications can require hours to get results with CUDA “printf”, while the current implementation can refresh users’ print requests in real-time which is crucial for developing and debugging applications.

5 Related Work

Two types of work are related to this research. The first is for debugging and simulating CUDA programs. The Ocelot\cite{14} framework and the WASTE CUDA\cite{15} simulator have been used on many studies. Ocelot is an open source project that can run CUDA programs on a GPU or a multi-core x86 CPU. The PTX\cite{13} simulator provided by Ocelot can simulate functions and debug with PTX instructions on a CPU, which simplifies developing CUDA applications. WASTE CUDA provides device simulations of CUDA programs in the Windows operating system allow programmers to use new library functions with a dynamic links. For example, WASTE CUDA rewrites the CUDA memory management APIs with a debugging mechanism to detect and output errors. Unlike Ocelot and WASTE CUDA, this research focuses on developing and debugging CUDA programs on the GPU by enabling developers to directly use I/O functions in the device to get interactive data in real time as the programs actually run on the GPU.

The second type of study extends the CUDA functions. For example, CUDASA\cite{16} is a development environment for distributed GPU computing for multi-GPU systems. This system extends CUDA’s parallel programming model for graphics processors to support higher parallelism with PCI bus and network interconnects. Extended APIs are provided for the application developer, including a global memory for all distribution layers and the underlying communication mechanisms. Another example is CheCUDA\cite{17}, which is designed to checkpoint CUDA applications that use GPUs as accelerators. CheCUDA also has new APIs to record status changes during processing.

6 Conclusions

This paper describes I/O functions, such as “printf” and file operations, for a GPU using the mapped memory. This method solves some challenges in the host agent process. A dynamic polling service model is used which has strong adaptability and efficiently implements I/O request processing in the host. Dynamic control of the polling cycles and scan order further improves the execution efficiency farther. A queue lock mechanism solves the competition problem between threads and makes possible use of a fixed number of queues. Handshaking ensures the coherence of interactions between the host and the device, with signal states fetching to solve the cache coherence problem.

Test results show that this implementation correctly and efficiently processes user agent requests, so the I/O operations are efficient with feedback to the user in real-time. This “printf” implementation is 1.06 times faster than CUDA’s implementation on average when one thread sends out print requests in each block, and the advantage is more obvious as the number of requests increases. The influence of the file operations is also small with an average increase execution time of 12.2% relative to the time without file operations.

This research will broaden CUDA applications and support real-time debugging to improve productivity of converting legacy C/C++ codes to CUDA codes. This is very helpful to both application developers and compiler designers. The idea of an I/O agent which uses mapped memory is scalable and can be used in other CUDA functions, such as memory operations, message sending and receiving, socket operations, and even a whole web server.

Acknowledgements

This work was supported in part by the National High-Tech Research and Development (863) Program (No. 2012AA010903).
References


Wei Wu received his MS degree in computer science and technology from the National University of Defense Technology, China in 2009. He joined the Jiangnan Institute of Computing Technology at Wuxi in 2010 and is now an assistant engineer at this institute. His research interests are in compiler design and parallel programming.

Fengbin Qi received his PhD degree from the Jiangnan Institute of Computing Technology and is now a PhD supervisor and a senior engineer at this institute. His research interests are in software design, high performance computing, and computer architecture.

Wangquan He received his MS degree from the Information Engineering University of the People’s Liberation Army, China and is now a senior engineer at the Jiangnan Institute of Computing Technology. His research interests are in compiler design and optimization, parallel algorithm, and high performance computing.

Shanshan Wang received her MS degree from the Information Engineering University of the People’s Liberation Army, China and is now an engineer at the Jiangnan Institute of Computing Technology. Her research interests are in compiler optimization and parallel algorithm.