Using High Level GPU Tasks to Explore Memory and Communications Options on Heterogeneous Platforms

Chao Liu
Northeastern University
Boston, MA 02115
liu.chao@husky.neu.edu

Janki Bhimani
Northeastern University
Boston, MA 02115
bhimani@ece.neu.edu

Miriam Leeser
Northeastern University
Boston, MA 02115
mel@coe.neu.edu

ABSTRACT

Heterogeneous computing platforms that use GPUs for acceleration are becoming prevalent. Developing parallel applications for GPU platforms and optimizing GPU related applications for good performance is important. In this work, we develop a set of applications based on a high level task design, which ensures a well defined structure for portability improvement. Together with the GPU task implementation, we utilize a uniform interface to allocate and manage memory blocks that are used by both host and device. In this way we can choose the appropriate types of memory for host/device communication easily and flexibly in GPU tasks. Through asynchronous task execution and CUDA streams, we can explore concurrent GPU kernels for performance improvement when running multiple tasks. We developed a test benchmark set containing nine different kernel applications. Through tests we can learn that pinned memory can improve host/device data transfer for GPU platforms. The performance of unified memory differs a lot on different GPU architectures and is not a good choice if performance is the main focus. The multiple task tests show that applications based on our GPU tasks can effectively make use of the concurrent kernel ability of modern GPUs for better resource utilization.

CCS CONCEPTS

• **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **General and reference** → *Experimentation*; *Performance*; • **Software and its engineering** → *Parallel programming languages*;

KEYWORDS

Heterogeneous platform; GPU; Memory Transfer; Concurrent Kernel

ACM Reference format:
Chao Liu, Janki Bhimani, and Miriam Leeser. 2017. Using High Level GPU Tasks to Explore Memory and Communications Options on Heterogeneous Platforms. In *Proceedings of SEM4HPC'17, Washington , DC, USA, June 26, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3085158.3086160

SEM4HPC'17, June 26, 2017, *Washington , DC, USA*

DOI: http://dx.doi.org/10.1145/3085158.3086160

1 INTRODUCTION

Due to their massively parallel processing ability, heterogeneous platforms that use GPUs for acceleration are more and more popular in high performance and parallel computing [14]. There are various programming languages and tools to help people develop parallel applications on GPU platforms. CUDA [17], which was introduced by NVIDIA in 2005, is now the most widely used programming language to develop programs that can run on NVIDIA GPUs. Unlike CUDA, which only support NVIDIA GPUs, OpenCL [19] is an open language standard that aims to develop parallel applications on manycore architectures which include both GPUs and multicore CPUs. In addition to these novel programming languages, traditional parallel programming methods have started to support accelerators. OpenMP introduced new features that allow users to offload specific code blocks onto accelerators for execution [18].

However, to accelerate applications with GPUs is not straightforward [20] and great efforts are needed for users to develop applications that can make use of GPUs effectively. Usually, a high performing GPU program relies on deep knowledge of the target GPU architecture. The complex GPU memory hierarchy and different GPU/CPU memory transfer options, as well as the use of processing units, are all important factors to consider when designing and developing parallel programs on GPU platform [5, 7, 24]. This brings great challenges to ordinary application developers, especially for those domain application experts who have little knowledge of computer hardware. Further, due to this close association of program performance and hardware features, when running an application on different platforms, lots of effort is needed to tune or reprogram the application. Thus, GPU specific programming methods provide the basics for users to develop high performing parallel programs, but they lack the ability to provide good application portability and productivity.

In our previous work, we proposed a programming framework that allows users to develop parallel applications based on a high level tasks and conduits abstraction [13]. Through this framework, an application's high level program structure and low level computational logic implementation are well separated. When running or porting such an application to different platforms, the application main structure can remain unchanged and only appropriate task implementations need to be adapted, reducing the effort of developing and maintaining parallel applications. Here we develop GPU applications based on this framework; the computational intensive workload of the application is implemented through CUDA programming and wrapped as a GPU task template for use. In this way, a user can tune GPU programs for different platforms without affecting the application main structure and a variety of

task implementations can be organized as libraries for usage and optimization.

To further facilitate developing GPU programs and explore different types of CPU/GPU memory transfer for performance optimization, we provide a uniform and simple interface to allocate space on system main memory (host memory) and GPU memory (device memory) in GPU task implementations. Through the CUDA runtime, there are three different types of memory: pageable memory, pinned memory and unified memory, that can be allocated on system for data transfer between CPU and GPU. The type of memory that is a good choice for an application depends on the characteristics of the application [2]. With the help of this uniform interface, we can flexibly specify the memory that will be used for data transfer. To demonstrate and analyze the effect of different memory for CPU/GPU transfer in real applications, we develop an application benchmark set currently containing nine different kernel applications. We implement each application based on our GPU task design and can flexibly adapt different types of memory transfer for experiments. In addition, through asynchronous execution of multiple GPU tasks and CUDA stream, we are able to explore concurrent kernel execution on a GPU easily.

The contributions of this papers are:

- Developing and running parallel applications on GPU platforms based on high level GPU task abstraction for a better structure and portability.
- Introducing a simple and uniform memory usage interface that enable user to apply different types of memory for CPU/GPU memory transfer with minor programming effort. Exploring concurrent kernel execution through multiple tasks and streams for better resource utilization and performance.
- Developing a benchmark set with nine different kernel applications for test and analysis.
- Analyzing performance of benchmark applications with respect to different host memory schemes on different GPU platforms.

The rest of paper is organized as follow: High level GPU task design is described in Section 2 and we discuss CPU/GPU data transfer and concurrent execution in Section 3. In Section 4, we give some information about benchmark development; detailed tests and analysis are presented in Section 5. In Section 6 we talk about related work, and draw conclusions in Section 7.

2 TASK BASED PARALLEL APPLICATION DESIGN FOR GPU PLATFORMS

Traditional parallel programming methods such as MPI, OpenMP for CPU platforms or CUDA, OpenCL for GPU platforms are all low level programming methods and suitable for implementing data parallelism in an application. But they lack the abstraction of high level task parallelism. To overcome this deficiency, we propose a unified task and conduit framework for producing parallel applications on heterogeneous platforms. Target GPU platforms can make use of this paradigm to define and create GPU tasks which will leverage GPUs of the platform to accelerate critical computation. The basic GPU task based design is shown as Figure 1.

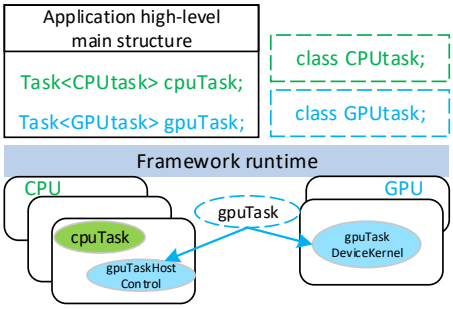


Figure 1: GPU task design

In an application, each sub-workload is implemented as a separate task class that will be used to define and instantiate task object. In our design, a task can be either a sequential program or a multi-threaded parallel program. Because of the unique features of current GPU programs, a GPU task implementation is usually comprised of two parts: host control and device kernel. The host control part is similar to a normal CPU task, except that it is now associated with a GPU device in the system, from which it is able to launch device kernels for accelerated computation. The GPU bind and other necessary initialization work are all completed by the runtime during task creation. The task is not constrained to run on a single node, we support running tasks on many nodes to make use of available computing resources when developing applications on a cluster platform. Overall, by developing parallel applications with this task mechanism, we can leverage various computing resources of the system as well as keep a well defined application structure, which makes it easy for users to port and tune the application for a target platform.

3 DATA TRANSFER EXPLORATION AND CONCURRENT EXECUTION ON GPU

3.1 Uniform data allocation for data transfer

A GPU is used as an accelerator (device) and usually connected to the system (host) by PCI-e. It has its own memory for accessing data while running programs. So any GPU application has two primary phases (which may overlap in time) shown in Figure 2: transfer required input data and results between system main memory (host memory) and GPU memory (device memory); execute GPU kernels with GPU processing units. Targeting NVIDIA GPUs and with the help of the CUDA runtime system, users can allocate three different types of memory in a system for communication between host and device through PCI-e: pageable memory, pinned memory and unified memory.

Pageable memory allows users to allocate memory blocks in virtual address space, and the allocated memory size can be larger than the available physical RAM size. The OS manages user allocated memory through memory page mapping mechanism: when a program accesses a virtual address that is not mapped to physical RAM, a memory page will be mapped and transfer the data to physical RAM (page-in); when a memory page on physical RAM is not accessed for a while, it will be unmapped from physical RAM and the data cached on disk (page-out). So pageable memory is

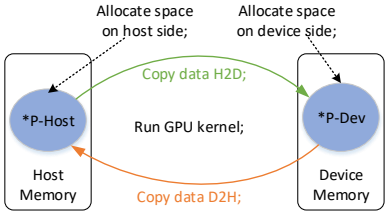


Figure 2: GPU program procedure

also called non-locked memory, which means it may not constantly reside in physical RAM. The drawback of using pageable memory for host/device transfer is that the communication procedure needs CPU involvement and possible page-in/page-out operations introduce extra overhead. So the bandwidth of the PCI-e bus that connects GPU to host is not fully exploited.

Pinned memory which can be allocated by CUDA utilities *cudaMallocHost* or *cudaHostAlloc* is a locked memory, which means it cannot be swapped out (page-out) from physical RAM once it is allocated. Because it resides in physical RAM permanently, data transfer between host and device can be carried out by DMA (Direct Memory Access) operations. This can enhance the bandwidth utilization of PCI-e and speed up the data transfer procedure. However, physical RAM is a limited resource. Allocating too much pinned memory could consume too much available physical memory and affect other programs or the entire system performance negatively.

In addition to pageable and pinned memory, starting from CUDA 6.0, unified memory has been introduced. Unified memory is memory blocks allocated using *cudaMallocManaged* function. In previous two types of memory, host and device have different address pointers and explicit data transfer operations are needed. However, the unified memory system creates and manages a pool of memory blocks that are shared between CPU and GPU. There is only one address pointer exposed to the user and both programs on host and device can use this pointer to access data. Data transfer between host and device happens implicitly in the underlying CUDA runtime. Unified memory frees a user from tedious host/device memory creation and management, easing GPU programming. But unified memory management is not always as efficient as explicit memory transfer assigned by a user manually.

As described above, different kinds of memory can be used in a GPU program, no one is the best choice all the time. Pinned memory provides better bandwidth usage, but we should avoid using it too much. Unified memory eases programming, but may not deliver the best performance. Also, using different types of memory requires different operations. The allocation and management of more and more different types of memory blocks is error prone and burdensome. Therefore, to facilitate GPU task implementation and enable a user to explore different types of memory freely and conveniently, we provide a simple and uniform interface to be used in GPU task implementations. In general, we defined following classes:

```
enum class MemType_t{
    pageable,
    pinned,
    unified}
```

```
template <typename T>
class GpuMem{
    ...
    MemType_t   m_memType;
    T           *m_hostPtr;
    T           *m_devPtr;
    ... }
```

When we need to allocate memory for a data set that is used on a GPU, we can create a *GpuMem* object with the desired memory type (pageable, pinned or unified). The object contains two pointers. When the memory type is pageable or pinned, the two corresponding pointers have different values and refer to host memory and device memory separately. When memory type is unified, these two pointers have the same value. Along with the object, we have different methods to get proper pointers and complete host/device data synchronization. Through these helper classes and methods, we can create a GPU task with appropriate memory type arguments to make use of different memory schemes easily and effectively.

3.2 Concurrent Execution on GPU

In early generations of NVIDIA GPUs, only one kernel program can be executed at a time. Multiple kernels are queued and executed by a GPU sequentially. However, from the Fermi GPU, NVIDIA supports running multiple kernels at the same time on one GPU if the hardware resources are available. With the enhancement of single GPU's computing resources (the GP100 GPU has 3840 CUDA cores), it is common that one kernel cannot exhaust all the resources at a given time. So running multiple kernels at the same time can make use of GPU resources more efficiently.

In CUDA programming, when initializing a GPU, a CUDA context is created. The following GPU related operations are associated with this context. There may be several CUDA contexts for the same GPU at the same time, but only one context is active at one time. A GPU will switch between these contexts frequently to process each context's operations. However, in one CUDA context, a user can have multiple CUDA streams. A CUDA stream comprises a series of GPU operations, such as memory copy and kernel execution, and they are executed one by one in a stream. Different streams of the same context can be processed in parallel. So different GPU operations of different CUDA streams can be executed simultaneously, as shown in Figure 3.

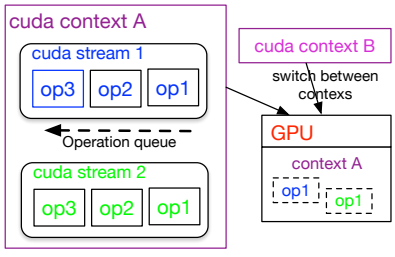


Figure 3: GPU stream for parallel execution

In our GPU task design, each task runs with single or multiple threads asynchronously and simultaneously. Task threads serve as

Table 1: Benchmark applications

Application	Domain	Workload(Small)
Image Rotation(Rotate)	Image Processing	320*200 pixels
Color Conversion(YUV)	Image Processing	320*200 pixels
Matrix Multiply(MM)	Linear Algebra	256*256 matrix
2D Heat Conduction(HC)	Linear Algebra	128*128 matrix
MD5 Calculation(MD5)	Cryptography	2K buffers with 4K bytes per buffer
K-means Clustering(Kmeans)	Data Mining	8K objects with 16 features per object
N-body simulation(Nbody)	Astrophysics Simulation	1k objects
Ray tracing(Raytrace)	Computer Graphics	128*128 pixels
Breadth First Search(BFS)	Graph Algorithm	64K nodes with 160K edges

the host control part for a GPU program and are associated with a GPU device for later GPU kernel execution. When different task threads are bound to different GPUs, GPU kernels of different task threads run in parallel. Furthermore, when they are bound to the same GPU, we can leverage CUDA streams to explore concurrent kernel execution. To make use of CUDA streams, for each GPU task we define a class *GpuContext* to record useful information related to a GPU device as shown below. When a GPU task is created, a `GpuContext` object is also setup.

```
class GpuContext{
    ...
    unsigned int    gpuId;
    cuContext_t     cudaCtx;
    CudaStream_t    streamId;
    ...
    }
```

In this object, GPU id indicates which GPU in the system a host thread is bound with. Multiple host threads may bind with the same GPU. The CUDA context handle (*cudaCtx*) tells the CUDA runtime environment the proper CUDA context to activate. The stream handle refers to the unique CUDA stream for the related GPU task thread. In the GPU task implementation, a user will use this stream as the argument for GPU operations. To target a single GPU, there are two mapping methods between a *GpuContxt* and CUDA context:

(1) One to one mapping: each *GpuContxt* refers to unique CUDA context. In this way, each GPU task will use a different CUDA context and they share a single GPU by context switching.
(2) One to more mapping: multiple *GpuContxt* refers to the same CUDA context on a single GPU. However, different *GpuContxt* still use different CUDA streams. So different GPU tasks can execute on a single GPU concurrently with the help of multiple streams. This is also the default configuration.

Therefore, based on the task asynchronous execution and multiple streams binding, we can easily run GPU kernels in parallel either on a single GPU or multiple GPUs.

4 BENCHMARK APPLICATIONS DEVELOPMENT

To demonstrate the use of GPU task based applications and analyze the performance of different types of host memory, we developed a benchmark set that currently includes the nine applications shown in Table 1. [1]

These applications involve a variety of domains. Many of them are well-known applications frequently used to demonstrate parallel processing, such as Matrix Multiply or N-body simulation. We have adapted them from other benchmarks using Tasks and Conduits. We implement the sequential version of each application as the base implementation and then implement the parallel version with GPU tasks to make use of GPU for acceleration. For each application, we prepare three different sizes of workload in our experiments, which are referred to as Small, Medium and Large (S, M, L). A brief information about small workload is also shown in Table 1; medium and large workload are generally one to two orders of magnitude larger than small workload.

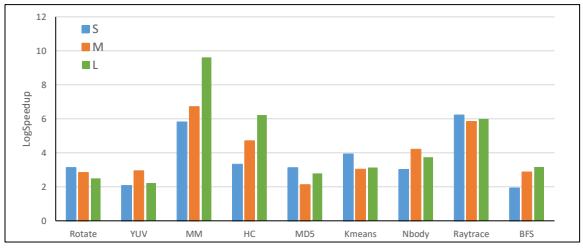
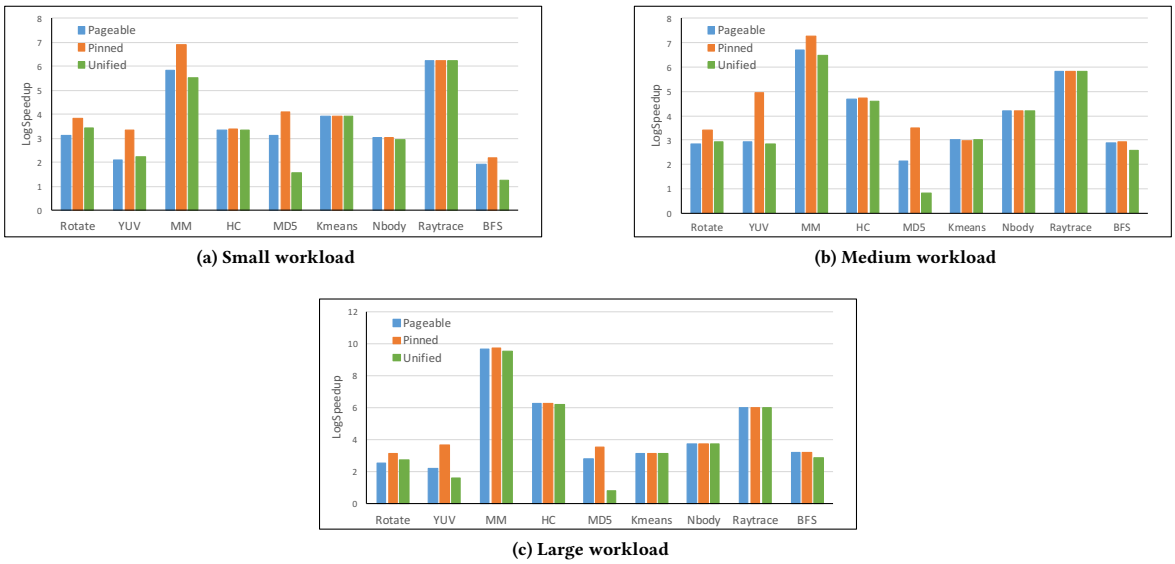


Figure 4: Applications speedup of different workloads on Tesla K20m

We run sequential implementations on an Intel Xeon E5-2650 CPU, recording the runtime as the baseline. All the speedup results in this work are computed with regard to this baseline execution time and we normalize the speedup value by $\log_2$ for convenience. Figure 4 shows the general speedup of each application on a NVIDIA Tesla K20m GPU using pageable memory. Both time costs of data transfer between host/device and kernel execution are counted to calculate speedup. We can see that by using GPUs for acceleration

[1] We plan to make benchmarks available on github.



(a) Small workload

(b) Medium workload

(c) Large workload

Figure 5: Applications speedup of using different types of memory on Tesla K20m

we are able to get speedup, but outcomes of different applications vary a lot, ranging from 4 times faster for BFS to as large as 600 times for MM. Also, for each application, the speedup trend for different workload size is not consistent. There are many factors leading to this variation, such as GPU hardware characteristics, parallel algorithm design, device memory access pattern, host/device memory transfer and so on. Therefore, great effort is required to develop a high performing implementation on a GPU platform. By designing and developing applications based on our high level task paradigm, we can keep a better program structure for porting and tuning applications on different platforms.

5 EXPERIMENTS AND ANALYSIS

In this section, we describe more detailed tests with the benchmark applications. Our test platform includes three different GPUs: NVIDIA Tesla C2070, NVIDIA Tesla K20m, and NVIDIA Tesla K40m. Tesla C2070 is NVIDIA Fermi architecture GPU and Tesla K20m and K40m belong to NVIDIA Kepler architecture GPU. Detailed GPU hardware features can be found in [15, 16].

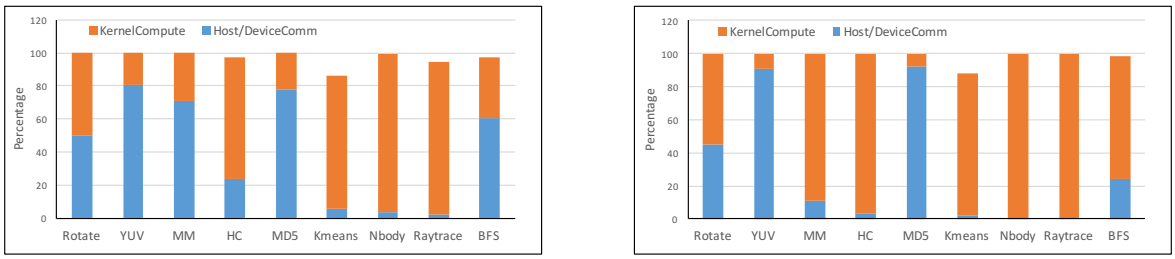
5.1 Pageable/Pinned/Unified Memory Usage

As discussed in Section 3, there are three types of memory available for a user to transfer data between host and device. Here we try each kind of memory with our benchmark applications. With the assistant of uniform memory using interfaces, we only need to pass different memory type arguments to adjust and experience various types of memory for the application. Figure 5 shows the speedup results of kernel applications running on a Tesla K20m GPU.

From the results we can see that, for Rotate, YUV and MD5, pinned memory performs better than pageable or unified memory. And the larger workload is more advantageous for pinned memory. For MM and BFS, using pinned memory is also preferable. However, for large workloads, the performance improvement by using pinned memory is not as good as small and medium workload. So considering the scarcity of pinned memory, pageable or unified may be a wise choice for applications with a large amount of data to process. For the rest of the applications, performance of the different types of memory are very close under all three workloads. If we want to save pinned memory usage we can use pageable memory for these applications, or use unified memory for program simplicity if you are not using our framework.

To further understand the character of these kernel applications, we record time costs of both host/device communication and GPU computation. Figure 6 shows the this information as a percentage of total time using pageable memory for each application. We can see that host/device communication time of HC, K-Means, Nbody and Raytrace are relatively much smaller compared to other applications. This is also why these four applications change less when adopting different types of memory. For MM and BFS, from small workload to large workload, we can see the time percentage of host/device communication reduced. So for large workload, using pinned memory for them cannot get much improvement. Figure 7 shows the host/device communication improvement when changing pageable memory to pinned memory for some of the benchmark applications. The pinned memory improvement in BFS reduces a lot for medium and large workloads mainly because the application implementation has an iterative structure, and during every iteration a flag value is transmitted to and from GPU device memory, but the amount of data transmitted is very small. This procedure does not dominate the communication time for small workloads, but for large workloads it is the primary part. Due to the small size of transferred data, even with pinned memory, the bandwidth use ratio is still too low to get much advantage.

From these tests, we conclude that pinned memory can improve host/device communication performance significantly. For applications where memory transfer takes up a substantial amount of total



(a) Small workload

(b) Large workload

Figure 6: Computation/Communication time cost percentage using pageable memory

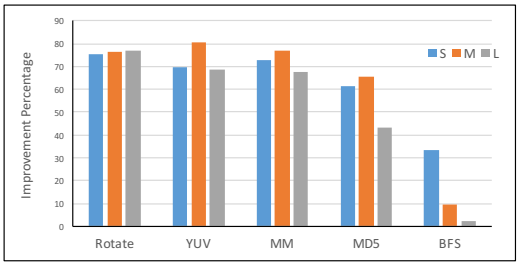


Figure 7: Host/Device communication improvement from pageable to pinned memory

runtime, using pinned memory can improve overall performance a lot. But memory transfer percentage may change with respect to the workload, like MM in our tests, so we should choose the memory type accordingly. Also, we cannot use pinned memory randomly. In some applications, choosing different memory types carefully depending on data size is a wise approach to improving overall performance. For choosing between different types of memory, with our interface it saves the user extra programming efforts, thus making the exploration of different design choices easier.

5.2 Unified Memory on Different GPUs

Unified memory provides a single memory pointer for both host side and device side to access memory. There is no explicit memory transfer operation compared to using pageable and pinned memory. From previous test results, we can learn that the performance of unified memory is usually similar or worse than using pageable memory. Here, we choose three applications from the benchmark to test on different GPU architectures to check its performance.

Figure 8 shows the test results of each application for different sizes of workload. We can find that on Tesla C2070 GPU, the performance of Rotate and MM using unified memory is much lower than pageable memory. While on Tesla K40m GPU this performance degradation improves a lot. For Raytrace, because there is little host/device transfer happened, using unified memory has little affection on application performance. Generally, NVIDIA Kepler architecture GPU has better support than Fermi architecture GPU. So unified memory eases the programming procedure, but the implicit data transfer managed by the underlying unified memory system and hardware support are not always efficient. using our approach, there is no difference in programming complexity between unified and other types of memory.

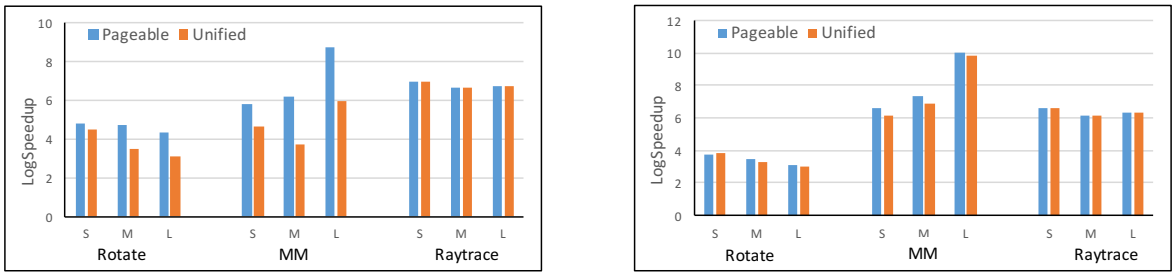
5.3 Concurrent Kernel Execution

In this section, we show the effect of concurrent GPU kernel execution. In our GPU task based application, each task is run by independent threads asynchronously and bound to a unique CUDA stream. Multiple tasks that share the same GPU are configured to be in one CUDA context. So GPU kernels of multiple tasks can be potentially executed in parallel on a single GPU. Here we test Nbody simulation on a Tesla K40m GPU. In Nbody simulation, when the total number of bodies is small, one GPU kernel can not consume all the computing resources of one GPU and multiple kernels are possible to execute simultaneously. With the existed task implementation of Nbody application, there is little changes to Nbody simulation to run multiple GPU kernels: only to instantiate more GPU task objects in the main program and invoke them to run.

Figure 9 shows the program run time results. We create different numbers of GPU tasks and run, keeping each task with the same number of bodies for computing. So when the number of GPU tasks doubles, the total computation workload of the application also double. As 4K bodies are 4 times larger than 1K bodies, the GPU kernel under 4K bodies consume more GPU resources and fewer GPU tasks are able to execute in parallel. Our test results just reflect this relationship: for 1K Bodies, the run time starts increasing noticeably after 8 GPU tasks; While for 4K bodies the run time starts growing after about 2 tasks.

6 RELATED WORK

Due to the high processing throughput and massive parallel processing ability of GPUs, heterogeneous platforms with GPUs are more and more prevalent. There is lots of research interest about developing and optimizing parallel applications on GPU platforms. To develop programs for GPU, CUDA and OpenCL are the two primary and basic programming methods. They expose detailed GPU characters to developers and need great effort to produce high performance applications. To ease programming difficulty, new programming language features and compilers are introduced to help user generate GPU programs automatically. OpenAcc [23], OpenMP and X10 to CUDA [4] allow users to insert compiler directives to let a compiler generate GPU program automatically. But it is hard to guarantee the quality of generated code [6]. In order



(a) Tesla C2070 test

(b) Tesla K40m test

Figure 8: Unified memory test on different GPUs

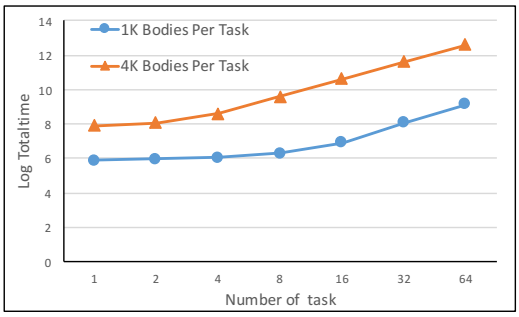


Figure 9: Nbody concurrent GPU kernel execution

to generate high performance code, a user may still need to get familiar with the hardware and insert complex directives to assist compiler. Along with compiler approaches, library or algorithm template based approaches also reduce programming difficulty and can provide high performance, for example cuFFT and the Thrust library [1]. Similar to Thrust, SYCL [9] introduces specifications of a C++ template library and SYCL device compiler, which enable users to run C++ programs on OpenCL supported heterogeneous platforms. In this work, we implement parallel applications based on a high level tasks and conduits model, which ensures a well defined program structure to improve program portability. By implementing and wrapping GPU kernels as tasks, we can easily modify and tune GPU kernel code without affecting the program main structure. As GPU tasks are still implemented in CUDA, we are able to make use of existing CUDA supported libraries for good performance and program productivity. Furthermore, by running tasks asynchronously and with appropriate task/stream mapping, we can easily explore concurrent kernel execution on GPUs for better GPU utilization.

Besides programming tools and methods for GPU platforms, other research work is to optimizing GPU applications. Dymaxion [3] provides means to reform the data placement on GPU memory and GPU memory access patterns to improve kernel execution. M. Wahib et al. [21] introduce a method to fuse small GPU kernels together to reducing data traffic to off-chip memory for better performance. [8, 11, 22] are all research about optimizing host/device memory transfer in GPU applications. They all provide approaches to manage host/device transfer, keeping data coherency between host and device and trying to avoid redundant communication. CUDA provides different types of host memory to carry out host/device transfer and which choice is good for an application depends on the characteristics of the application [2, 10, 12]. In our work, we also provide a simple uniform interface to allocate and manage data sets that will be used on both host/device sides. Together with GPU tasks design, we can freely choose the desired type of memory during runtime to explore the performance differences for an application.

7 CONCLUSIONS

In this work, we show the development of parallel applications on heterogeneous GPU platforms through a high level task design. Together with GPU tasks, we provide a simple and effective interface for allocating and transferring data that enables us to flexibly choose pageable memory, pinned memory or unified memory for data communication between host main memory and GPU memory. We developed a benchmark set including nine applications for test. By running these applications using different types of memory for data transfer, we notice that pinned memory can improve the data transfer due to its physical persistence feature. But for applications where data transfer is not critical, it is not worth it to use pinned memory. Unified memory provides a single address pointer for both host and device, but its performance is the drawback. On older generations of GPUs, such as Tesla C2070, unified memory performs worse than pageable memory. Also, by binding different GPU task threads with unique CUDA streams, we can easily explore concurrent GPU kernels on a single GPU through our asynchronous task execution. Through multiple task execution tests, we can see the effect of concurrent kernels on a single GPU, which proves our GPU task based parallel applications can benefit from concurrent kernels for good performance. In future work, we will further improve our task/conduit based framework on heterogeneous platforms and enrich the test benchmarks with more comprehensive and representative applications. Currently, these kernel applications make use of a single GPU for execution; we plan to extend them to multiple GPUs and muti-node cluster environments.

REFERENCES

[1] Nathan Bell and Jared Hoberock. 2011. Thrust: A productivity-oriented library for CUDA. *GPU computing gems Jade edition* 2 (2011), 359–371.

[2] J. Bhimani, M. Leeser, and N. Mi. 2016. Design space exploration of GPU Accelerated cluster systems for optimal data transfer using PCIe bus. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[3] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. 2011. Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. Article 13, 11 pages.

[4] Dave Cunningham, Rajesh Bordawekar, and Vijay Saraswat. 2011. GPU Programming in a High Level Language: Compiling X10 to CUDA. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop (X10 '11)*. Article 8, 10 pages.

[5] Tianyi David Han and Tarek S Abdelrahman. 2011. Reducing branch divergence in GPU programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 3.

[6] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and Ryoji Takaki. 2013. CUDA vs OpenACC: Performance case studies with kernel benchmarks and a memory-bound CFD application. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 136–143.

[7] Thomas B Jablin, Prakash Prabhu, James A Jablin, Nick P Johnson, Stephen R Beard, and David I August. 2011. Automatic CPU-GPU communication management and optimization. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 142–151.

[8] Jens Kehne, Jonathan Metter, and Frank Bellosa. 2015. GPUswap: Enabling Oversubscription of GPU Memory Through Transparent Swapping. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '15)*. 65–77.

[9] Ronan Keryell, Ruyman Reyes, and Lee Howes. 2015. Khronos SYCL for OpenCL: A Tutorial. In *Proceedings of the 3rd International Workshop on OpenCL (IWOCL '15)*. Article 24, 1 pages.

[10] R. Landaverde, Tiansheng Zhang, A. K. Coskun, and M. Herbordt. 2014. An investigation of Unified Memory Access performance in CUDA. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6.

[11] Lu Li and Christoph Kessler. 2017. VectorPU: A Generic and Efficient Data-container and Component Model for Transparent Data Transfer on GPU-based Heterogeneous Systems. In *Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM '17)*. 7–12.

[12] W. Li, G. Jin, X. Cui, and S. See. 2015. An Evaluation of Unified Memory Technology on NVIDIA GPUs. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 1092–1098.

[13] Chao Liu and Miriam Leeser. 2017. A Framework for Developing Parallel Applications with High Level Tasks on Heterogeneous Platforms. In *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'17)*. ACM, New York, NY, USA, 74–79. DOI: https://doi.org/10.1145/3026937.3026946

[14] J. Nickolls and W. J. Dally. 2010. The GPU Computing Era. *IEEE Micro* 30, 2 (2010), 56–69.

[15] NVIDIA. 2013. TESLA K20 GPU ACCELERATOR. (2013). http://www.nvidia.com/content/pdf/kepler/tesla-k20-passive-bd-06455-001-v07.pdf

[16] NVIDIA. 2013. TESLA K40 GPU ACTIVE ACCELERATOR. (2013). https://www.nvidia.com/content/PDF/kepler/Tesla-K40-Active-Board-Spec-BD-06949-001_v03.pdf

[17] NVIDIA. 2015. CUDA C Programming Guid. (2015). http://docs.nvidia.com/cuda/pdf/

[18] OpenMP Architecture Review Board 2013. *OpenMP Application Program Interface, Version 4.0*. OpenMP Architecture Review Board.

[19] John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12, 3 (2010), 66–73.

[20] Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure. 2010. On the Limits of GPU Acceleration. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism (HotPar'10)*. 13–13.

[21] Mohamed Wahib and Naoya Maruyama. 2014. Scalable Kernel Fusion for Memory-bound GPU Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. 191–202.

[22] Kaibo Wang, Xiaoning Ding, Rubao Lee, Shinpei Kato, and Xiaodong Zhang. 2014. GDM: Device Memory Management for Gpgpu Computing. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '14)*. 533–545.

[23] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. 2012. OpenACC: First Experiences with Real-world Applications. In *Proceedings of the 18th International Conference on Parallel Processing (Euro-Par'12)*. 859–870.

[24] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu. 2010. Adaptive Optimization for Petascale Heterogeneous CPU/GPU Computing. In *2010 IEEE International Conference on Cluster Computing*. 19–28.