Efficiently using CUDA enabled GPUs as shared resource

Hagen Peters, Martin Köper, and Norbert Luttenberger
Research Group for Communication Systems
Department of Computer Science
Christian-Albrechts-University Kiel, Germany

Abstract

General Purpose computation on Graphic Processing Units (GPGPU) is getting more and more important, but when using CUDA-enabled GPUs for GPGPU the special characteristics of NVIDIA’s SIMT architecture have to be considered. Particularly, it is not allowed to run functions concurrently, although NVIDIA’s GPUs consist of many processing units.

Therefore, GPUs are accessed exclusively by a single process or application in general, and for an efficient use of the GPU, it has to be fully utilized by a single function launch, i.e. it is not possible to utilize the GPU by multiple function launches from one or more processes.

In this contribution we present an approach that partially overcomes these restrictions by using the GPU as a shared resource. Therefore, a GPGPU service executes a persistent kernel which consists of a set of device functions. The service controls via asynchronous memory transfers individually for each block of threads which function the block has to execute. Using this novel approach, one or more clients can access and utilize a single GPU by running several different tasks at the same time. Our tests show the great flexibility and the high performance of this approach, by which further areas of applications are enabled to benefit from GPGPU.

1 Introduction

GPGPU is getting more and more important. The computational power of NVIDIA’s GPUs can be easily used for general purpose applications by programming the GPUs with the Compute Unified Device Architecture (CUDA) framework [4].

The development of a kernel (function running on the GPU) is predetermined by the Single Instruction Multiple Threads (SIMT) architecture of NVIDIA’s GPUs: The calling process specifies within the kernel call the number of threads that execute this kernel. Unfortunately, the currently available NVIDIA GPUs do not allow to run kernels concurrently and even future GPUs were advertised to run only a few kernels concurrently. In addition, it is not trivial to efficiently time-share one GPU among threads or processes. By that reasons, CUDA-enabled GPUs are accessed exclusively by the calling host process for the runtime of the called kernel and furthermore, they are generally accessed exclusively for the runtime of the calling process. Thus, it is not possible to share the processing power of the GPU among multiple function launches from one or more processes.

Due to the fact that the GPU is not used as a shared resource, GPGPU is only beneficial if the GPU is fully utilized by a single kernel launch. Due to this reasons, GPGPU may be inadequate when

- computing functions which cannot be expressed efficiently in a massive data-parallel way
- computing functions that do not address enough data to utilize the GPU
- computing functions of unknown priority, start time and runtime.

These restrictions make GPGPU inappropriate for many areas of applications.

As an example lets consider encryptions with DES [?] in CBC mode. Due to the chaining mechanism the encryption cannot be expressed in a massive data-parallel way, although some computations can be processed concurrently. Thus, DES in CBC mode is inadequate to utilize current GPUs that consist of hundreds of processing units. But even if DES is used in ECB mode, the GPU is not fully utilized if not enough data is addressed, i.e. the plain-text is too short (on a GTX280 at least $60 \cdot 128 = 7680$ threads should be used). Furthermore, even when fully utilizing the GPU by encrypting large plain-texts with DES in ECB mode, the encryption may take minutes and during kernel runtime no other client can access the GPU, thus even high priority or real-time tasks have to wait.
In this scenario, conventional GPGPU would be only beneficial in the case that one dedicated process encrypts large plain-texts using DES in ECB mode.

Although these limitations of GPGPU are well known, only a few work concerning this problems is known to the authors. There are some approaches [1, 2] to process many tasks of the same type in one long running kernel, whereby one block processes one task and when it is finished the block fetches the next task from a global pool. These approaches only fit in a scenario where many tasks of the same type, that are all known at kernel launch time, have to be executed by a single process.

In this contribution we present an approach that overcomes partially the limitations of GPGPU. Therefore, a GPGPU service launches a persistent (i.e. infinite running) kernel (executed by a fixed number of blocks of threads) which provides a set of functions. In contrast to previous approaches the GPGPU service controls the kernel through asynchronous memory transfers on block level, i.e. the service manages individually for each block which function it has to execute. The GPGPU service provides interfaces through that many clients can access and utilize the GPU at the same time.

This makes GPGPU beneficial for further areas of applications in which the processing power of the GPU is shared by many functions called by many clients, e.g. web services, terminal server, security gateways etc.

In the following section we introduce the approach of the GPGPU service. Section ?? presents CUDA specific details of the implementation and in section 3 we discuss some characteristics of the design of the persistent kernel. We evaluate the results of the tests with the implementation in section 5 and conclude this contribution in section 6.

2 Approach

As stated above, when a process launches a kernel, the number of blocks of threads that should execute this kernel has to be specified within the kernel call. Threads can access their unique block identifiers as well as their unique identifiers within their block, thus they can address different data or process different instructions depending on their unique identifiers. Although NVIDIA uses SIMT architecture, CUDA-enabled GPUs process different blocks independently, thus different blocks can execute different instructions at the same time without loss of performance.

Because it is not possible to run multiple kernels at the same time, the idea of the approach presented in this contribution is to run one persistent kernel on the GPU that provides a set of different functions. Each block that executes this kernel, reads within a infinite loop its "Function Identifier" from a dedicated memory location and decides depending on the value of the identifier where to branch next, i.e. which function arguments and user data to read and which function to execute. After the function execution the block starts over again. Therefore, the requirements of this approach are:

1. Each block running the kernel eventually gets GPU time.
2. User data (including results) corresponding to the function identifiers has to be synchronized between host and GPU.
3. Function identifiers and arguments have to be passed to each block individually.
4. The state of each block (e.g. "running", "ready" etc.) has to be reported to the host.

While the first requirement can be fulfilled by considering some device specific guidelines (section ??), the other requirements can be fulfilled by carefully using asynchronous memory transfers and zero copy respectively.

3 Implementation

The GPGPU service design closely follows the approach presented in section 4. Figure 2 gives a coarse grained overview of the design of the service that is explained in detail in the following paragraphs.

Active Blocks

When executing a kernel on the GPU, only a small number of blocks are active. Only these blocks, or more precise
threads that belong to these blocks, get GPU time. A GPU has a maximum number of active threads it can handle and all active blocks use their resources (registers, shared memory) at the same time, thus the number of active blocks is limited by the number of active threads the GPU can handle and by the number of registers and the amount of shared memory provided by the GPU. Only if an active block terminates, an inactive block changes its state to active. When running a persistent kernel no active thread ever terminates, i.e., an inactive thread never changes its state to active. Because of that, the number of blocks that run the persistent kernel is strictly limited to a small fixed number \( n \), such that all blocks are active. In figure 2 the blocks are represented by the boxes \( B_1, \ldots, B_n \) on the device side.

**Memory Management**

As stated in the introduction, it is not possible to execute kernels on the GPU concurrently. However, some NVIDIA devices provide asynchronous memory transfer, whereas memory can be transferred from or to the GPU while a kernel is running. Thus, it is possible to exchange function identifiers, function arguments, user data and block states between the host and the GPU during persistent kernel execution.

Memory on the GPU can be allocated using `cudaMalloc()` while the kernel is running. Unfortunately, it is not possible to `cudaFree()` this memory during kernel execution. Thus, when using a persistent kernel and repeatedly allocating device memory, the device memory will be exhausted eventually. Due to this, there is a need for a dedicated host controlled memory management unit (MMU), that is independent of `cudaMalloc()`, `cudaFree()` etc. When starting the GPGPU service our implementation of the MMU allocates user data memory on the GPU and on the host and synchronizes this memory. This is depicted in figure 2 by the boxes for user data on the host and the device, that are connected by the data pipe and controlled by the MMU. Whenever a client requests the GPGPU service, the MMU reserves the memory needed by this client both on the host and GPU side. When the request is processed, the MMU passes the result to the client and frees the memory.

**Function Data and Block Status**

To control the blocks running the persistent kernel, the function data (identifiers and arguments) have to be passed to the GPU and in addition the blocks have to report their states to the host, primarily to report if a block has already finished its task, but also to report failures etc.

As stated above, user data is managed by a dedicated MMU. Certainly, it is possible to handle the function data and the block status within the MMU. On the other hand the functions provided by the persistent kernel and also the number of blocks \( n \) are known at compile time. Thus, we can avoid the additional delay caused by MMU operations by simply pre-allocating for each block the memory required for the function data of each function and the block status.

In figure 2 the function pipe illustrates how the management thread controls function data through memory transactions. The status pipe represents the memory transactions by which the blocks report their states to the host using zero copy [3], i.e., a kind of asynchronous memory transaction triggered by the GPU.

Figure 3 shows how blocks process identifiers, arguments and states. Each block \( B_i, i \in \{1 \ldots n\} \) repeatedly reads the dedicated memory location of its function identifier \( ID_i \). If the value of \( ID_i \) is not valid, block \( B_i \) starts over immediately. If the value matches one of the valid values \( Func_1, \ldots, Func_m \), the block \( B_i \) fetches the arguments \( A_{ID_i} \) of function \( ID_i \) (which contains a pointer to the user data) and executes the function \( Func_{ID_i} \). When the block is finished it writes its state using zero copy to its dedicated memory location \( S_i \) on the host.

**4 Kernel & Device Functions**

As mentioned above, the persistent kernel consists of an infinite loop and a set of device function. Figure 4 shows
When implementing the functions that should be accessible through the GPGPU service, two main aspects have to be considered. On the one hand, kernels were often designed such that the number of blocks that run the kernel fits to the size of the input data, but functions provided by the persistent kernel have to work correctly for a variable input size, although running a constant number of blocks.

On the other hand, previous applications were often designed using kernel launches for synchronization, i.e. to make sure that a task is processed before another one, they were processed in consecutive kernel launches. Obviously, that is not possible when using a persistent kernel, but fortunately CUDA provides inner block synchronization methods (_syncthreads(), __threadfence() etc.) that can replace the "kernel launch synchronization".

For our GPGPU service we implemented two device functions, a matrix multiplication and a sorting function. The matrix multiplication is based on the shared memory implementation of the CUDA Programming Guide 3.0 [4] and the sorting function is based on a normalized variant of the bitonic sort implementation introduced by Peters et al. [5]. We adapted both implementations to be used with a single block by introducing wrapper functions that iterate over virtual block indices. Figure 1 presents the pseudo code for the multiplication wrapper. Thereby, the original code (MatMultKernel) is repeatedly called with virtual block indices virtX and virtY.

While matrix multiplication already uses only one kernel launch, we had to replace multiple kernel launches within bitonic sort by multiple device functions synchronized through _syncthreads().

5 Evaluation

We developed and executed the prototypical implementation of the GPGPU service using Ubuntu 9.04, the g++ 4.3.3 compiler and CUDA 3.0. We made our tests on a ASUS Z8PE-D18 board equipped with two INTEL E5520 processors, 24 GB RAM and both a GTX260 and a GTX280. We configured the persistent kernel such that the number of threads per block is 128 and the number of blocks is twice the number of multiprocessors, i.e. 54 for the GTX260 and 60 for the GTX280.

When testing the GPGPU service we first started the service and then created host threads that independently sent either one sorting request (SR) or one multiplication request (MR) to the GPGPU service, wait until the result returns and quit. Thereby, the input of a sorting request is a sequence of $2^0$ uniformly distributed 32 bit integers, while the input of the multiplication request are two $(2^2 \times 2^3)$-matrices also filled with 32 bit integers. For performance measurement we counted both the number of threads that finished a sorting request and the number that finished a multiplication request within a time interval of length $t = 60$ sec.

Thereby, we mainly varied two parameters:

1. The utilization, i.e. the number of requests currently handled by the GPGPU service
2. The ratio of the number of sorting requests and multiplication requests

We tested 66 different utilizations ranging from 0 to 65 and 26 different ratios (100:0, 96:4, . . . , 4:96, 0:100). To achieve a utilization of $x$ we send $x$ requests at once to the service and when a thread finishes we immediately start a new one. If the utilization is smaller than the number of
blocks running the persistent kernel, all requests that are handled by the services were currently processed by a block and a new request is instantly assigned to a free block. If the utilization is greater than or equal the number of blocks, all blocks were currently processing a request and remaining requests are waiting in a request queue until a block gets free. To achieve a ratio of \( ab \) we simply count the requests already sent and when a thread finishes start the request such that the ratio \( \frac{\#SR}{\#MR} \) is closest to \( ab \).

We varied the parameters of a test run without interrupting the GPGPU service, thus in both tests the kernel ran for more than 28 hours (66.26·60 sec = 28.6h).

Figure 5 presents for both the GTX260 and the GTX280 on the y-axis the sorting rate \((k·2^{20})\) divided by time for sorting \( k \) lists of length \( 2^{20} \) of the GPGPU service when launching only sorting requests. Additionally, the figure shows the maximum sorting rate that can be achieved by the implementation of Peters et al. using the GTX280 when sorting a single sequence of length \( 2^{20} \) without the service. The x-axis depicts the utilization of the GPU.

The figure presents two main results of our tests. First, the sorting rate for both GPUs achieved with the GPGPU service exceeds the sorting rate without the service for a utilization greater than 47. The maximum sorting rate is nearly 190 Mio./sec for the GTX280 and more than 160 Mio./sec for the GTX260. Second, the performance of the GPGPU service scales linearly in the number of used blocks in this scenario, i.e. the per block performance is independent of the utilization. Due to the fact that the per block performance of the GTX280 is not far above the per block performance of the GTX260, in our tests the service is barely able to benefit from the better GPU when the utilization is below 54. Trivially, for both GPUs the maximum performance is achieved when all blocks process a request, i.e. the utilization is greater than 54 (GTX260) and 60 (GTX280).

Figure 6 presents on the y-axis the multiplication rate \( (k·(2^9)^3) \) divided by time for multiplying \( k \) pairs of \((2^9 \times 2^9)\)-matrices) of the GPGPU service when launching only multiplication requests for both the GTX260 and the GTX280 and also the maximum multiplication rate we achieved in our tests without the service using the GTX280.

While the sorting function benefits from avoiding kernel launches for synchronization and performs better with the GPGPU service than without it, the multiplication function performs slightly worse within the service. This may be caused by the fact that matrix multiplication is a massive data-parallel algorithm without read after write dependencies, the need for synchronization etc., thus it cannot benefit from a single block implementation. Additionally, the GPGPU service causes a little management overhead and besides that, the multiplication rate (as well as the sorting rate in figure 5) without the GPGPU service is measured excluding memory transfers, while they are included when using the service.

The curve progression for multiplication request is the same as for sorting requests depicted in figure 5.

In contrast to figures 5 and 6 we used optimal utilization in figure 7 and varied the ratio between sorting and multiplication requests. The figure presents the percentage of sorting performance and multiplication performance relative to the maximum performances depicted in figures 5 and 6. Additionally, it presents the total performance (the sum of the percentages of sorting and multiplying performance).

It is trivial that the sorting performance is at 100% and the multiplication performance is at 0% when using a ratio of 100:0 while the performance is inverted when the ratio is inverted. In contrast the total performance is never below 100%, thus there is no loss in performance when the service has to process different requests.

At a first glance, it may be astonishing that the intersection of the curves is not at ratio 50:50, but one has to keep in mind that we vary the ratio of requests only. Sorting a sequence of \( 2^{20} \) takes more time than multiplying two \((2^9 \times 2^9)\)-matrices, thus a ratio of requests of 50:50 results in a ratio of performance of about 60:40 between sorting...
Figure 7. Performance ratio (GTX260)

and multiplying.

Another interesting effect of mixing up different requests presented in figure 7 is, that the total performance reaches a level slightly higher than 100%. This may be explained by the fact, that different device functions may be primarily limited by different resources, e.g. global memory bandwidth, the FLOPS-performance or shared memory access time etc. Thus, computing several functions limited by different resources at the same time may increase the total performance. In simple words, if one function waits for a resource the other function may execute. In the scenario used in this contribution it may be beneficial not to sort first 1000 sequences and after that to multiply 1000 matrices but to mix up both tasks.

The results of the tests with the GPGPU service clearly show the potential of our approach. The sorting performance of the service is significantly higher than without it and the multiplication performance is only slightly worse compared with the performance achieved with a single kernel launch (without measuring the time for memory transfers). In addition the GPGPU service is able to process different requests of several clients without any loss in performance. A drawback of this approach is, that the service needs a relatively high number of requests to achieve best performance.

6 Conclusion

In this contribution we presented a novel GPGPU service. The service launches a persistent kernel that consists of a set of functions and controls via asynchronous memory transfers for each block that runs the kernel which function it has to execute. The service provides interfaces through that clients can request the execution of one of these functions.

Thus, the service is enabled to share the processing power of the GPU among many requests of many clients at the same time. Our tests show that the performance of the service is competitive to the maximum performance achieved by a single kernel launch, but in contrast to single kernel launches, it can be accessed and utilized by different request of many clients what greatly increases the flexibility of GPGPU.

This makes GPGPU beneficial for new areas of applications that previously could not benefit from GPGPU by the reason they did not fully utilize the GPU or could not be expressed in a massive data-parallel way or should not get exclusive access to the GPU etc.

A drawback of our prototypical implementation is, that the GPU is not fully utilized when the service only processes a few requests. We can overcome this disadvantage in some cases easily by splitting a single request into many, e.g. a matrix multiplication can be expressed in terms of multiple matrix multiplications with smaller size, but in other cases like sorting it is not so easy. In further work we aim on the development of techniques by that a single request is processed by a variable number of blocks that depends on the current utilization of the GPGPU service.

References