Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

FLORIAN WENDE, THOMAS STEINKE, FRANK CORDES

Multi-threaded Kernel Offloading to GPGPU Using Hyper-Q on Kepler Architecture

ZIB-Report 14-19 (June 2014)

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraße 7
D-14195 Berlin-Dahlem

Telefon: 030-84185-0
Telefax: 030-84185-125

e-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

Multi-threaded Kernel Offloading to GPGPU Using Hyper-Q on Kepler Architecture

Florian Wende[1], Thomas Steinke[1], and Frank Cordes[2]

[1] Zuse Institute Berlin, Takustraße 7, D-14195 Berlin, Germany
{wende,steinke}@zib.de
[2] GETLIG&TAR GbR, Bachstelzenstraße 33A, D-14612 Falkensee, Germany
cordes@getlig.com

Abstract. Small-scale computations usually cannot fully utilize the compute capabilities of modern GPGPUs. With the Fermi GPU architecture Nvidia introduced the concurrent kernel execution feature allowing up to 16 GPU kernels to execute simultaneously on a shared GPU device for a better utilization of the respective resources. Insufficient scheduling capabilities in this respect, however, can significantly reduce the theoretical concurrency level. With the Kepler GPU architecture Nvidia addresses this issue by introducing the Hyper-Q feature with 32 hardware managed work queues for concurrent kernel execution.
We investigate the Hyper-Q feature within heterogeneous workloads with multiple concurrent host threads or processes offloading computations to the GPU each. By means of a synthetic benchmark kernel and a hybrid parallel CPU-GPU real-world application, we evaluate the performance obtained with Hyper-Q on GPU and compare it against a kernel reordering mechanism introduced by the authors for the Fermi architecture.

1 Introduction

General purpose graphics processing units (GPGPU or GPU for short) represent one major technology to implement hybrid computer systems taking advantage of the highly parallel data processing capabilities of modern many-core processor chips. Both the hardware architectures and the programming models – e.g. CUDA, OpenCL, and OpenACC – evolved to match the user's needs, expose architectural features, and provide flexibility for application design and migration.

Throughout various kinds of applications from the different fields of science and economy, the execution of highly parallel code sections on GPU can result in a significant increase in the overall program performance compared to executions on current multi-core CPUs. For SIMD-like regular computations with sufficiently large amount of available parallelism speedups up to one order of magnitude are reported [1, 2].

In contrast, small-scale computations do not rank among the promising candidates to exploit the capabilities of modern GPUs at first sight. In the context of heterogeneous computing, however, sophisticated hybrid codes with multi-threading or multi-processing on the CPU side and small-scale kernel offloads

within a CPU thread (process) may produce concurrent GPU workloads which as a whole can charge the GPU to capacity. The latter point is addressed by the ability of current GPUs to execute multiple independent computations in a concurrent way.

This paper continues our investigations to evaluate and improve concurrent kernel execution (CKE) within multi-level parallel applications [3]. The (CKE) feature introduced with Nvidia's Fermi GPU architecture [4] enables up to 16 independent workloads (kernels) to use a shared GPU device simultaneously. The performance gain for such applications, however, strongly depends on both the possibilities to express the independency of GPU computations within the host program – "implementation challenge" – and the ability of the GPU hardware to actually detect kernel independency at runtime – "architectural challenge."

Within Nvidia's CUDA programming model, independency is expressed by means of CUDA streams, corresponding to logically distinct in-order work queues. Computations associated with different streams do not depend on each other and thus can execute concurrently. On Fermi GPUs all streams are multiplexed into a single hardware work queue, where for two successive kernels to execute concurrently on the GPU it is necessary that they are assigned to different streams. If this is not the case all later kernels have to wait for an individual kernel to complete, resulting in false-serialization, and hence reduced CKE performance. In particular multi-threaded/-process applications are affected due to uncoordinated GPU offloading, possibly with multiple successive kernels on the same stream. With their Kepler GPU architecture [5], Nvidia approaches the false-serialization issue by providing 32 hardware queues streams are mapped to – known as Hyper-Q in the context of CKE.

In the present paper we investigate the Hyper-Q feature on Nvidia Kepler GPUs for both multi-threaded and multi-process applications. Our main contributions in this paper are:

1. Evaluation of the Hyper-Q performance using a synthetic benchmark kernel and a real-world hybrid CPU-GPU application.
2. Performance comparison between Hyper-Q and the kernel reordering mechanism we proposed earlier to compensate for a potential CKE breakdown on Nvidia Fermi GPUs [3].

Section 2 summarizes work already done in the field of small-scale computations on accelerators and concurrent kernel offloading. Section 3 briefly introduces Nvidia's Kepler GPU architecture and summarizes some elements of the CUDA programming model. The focus is on concurrent kernel execution and Hyper-Q. We also recap the basic idea of our kernel reordering mechanism for Fermi GPUs. In Section 4, we compare the CKE performance obtained with Hyper-Q against the one obtained with kernel reordering using a synthetic benchmark kernel. Section 5 applies CKE to a real-word application implementing a simulation of a small molecule solvated within a nanodroplet. Concluding remarks are given in Section 6.

2 Related Work

The strengths and limits of the concurrent kernel execution (CKE) feature have been investigated since the introduction of Nvidia's Fermi GPU architecture in late 2009. The need of a CKE feature was alluded by Guevara et al. [6] for Nvidia GT200 graphics cards. The authors provide an approach which concentrates on merging multiple small kernels into one large kernel (kernel batches) resulting in a concurrent execution of multiple GPU kernels on the same GPU.

Investigations on using the CKE feature for host setups with multiple threads or processes were published by El-Ghazawi et al. [7–9], with the focus on efficiently sharing a single CUDA context between these threads (processes). The authors compare a manual context funneling against the shared CUDA context introduced with the CUDA 4 API. For multi-threaded host setups, the authors found that the shared CUDA context gives about 90% of the performance that can be achieved by manual context funneling.

Wende et al. [3] proposed a GPU kernel reordering mechanism for Nvidia Fermi GPUs to recover concurrency on GPU when used within multi-threaded program setups. Despite of being placed on logically different CUDA streams, on Fermi GPUs all kernels are multiplexed into a single task queue. False-serialization of kernels within that queue can prevent the GPU scheduler from executing independent kernels simultaneously, potentially causing performance degradation of the program. With their kernel reordering mechanism the authors were able to reduce the level of false-serialization in the task queue. They showed their approach allows to almost entirely recover concurrency on GPU for a synthetic GPU kernel. For a real-world molecular simulation program, the concurrency level could be increases by a factor $2 - 3$.

Chen et al. [10, 11] addressed load-balancing issues at a finer granularity level. Their task queue schema enables dynamic load-balancing of application kernels on single and multiple GPU platforms.

Pai et al. [12] proposed the concept of elastic kernels together with multi-program workloads to improve resource utilization. They could overcome the limitations of the CUDA programming model by improving the turnaround time for a two-program workload from the Parboil 2 suite by a factor of 3.7.

Awatramani et al. [13] demonstrated the impact of kernel scheduling from multiple applications and demonstrated an overall throughput increase by 7% on average for the Rodinia benchmark suite.

In the context of migrating the highly parallelized NAMD legacy code, Phillips et al. [14] emphasized the need to schedule work from multiple CUDA streams to overlap and synchronize concurrent work on CPU and GPUs more efficiently.

3 Nvidia Kepler GPU Architecture and the CUDA Programming Model

The Kepler architecture is Nvidia's current unified-shader GPU architecture. It is build around a scalable set of streaming multi-processors, called SMX [5].

Each SMX contains 192 single-precision scalar processors (SPs), 4 scheduling and 8 instruction dispatch units. Kepler-based GPUs contain up to 15 SMX for a total of 2880 SPs and up to 12 GB GDDR5 ECC main memory. In addition to a per-SMX 64 kB configurable shared memory and L1 cache, Kepler doubles the amount of L2 cache to 1.5 MB compared to Fermi. With "Hyper-Q" (see Section 3.1) and "Dynamic Parallelism" indispensable features for more flexible programming found their way into the GPU architecture.

Table 1 summarizes some characteristic architectural properties of the devices used in this document.

Table 1: Characteristics of the processor platforms used in this document. Information are from [5] and from the vendor product homepages.

	Tesla K20c	Xeon E5-2670
Core | SMX Count	13	8 (×2 HT)
Scalar Processors per Core | SMX	192	–
32-Bit SIMD / SIMT Width	32	4 SSE, 8 AVX
Clock Frequency	0.705 GHz	2.6 GHz
Memory Bandwidth	208 GB/s	51.2 GB/s
Single-Precision Peak Perform.	3.52 TFLOP/s	0.33 TFLOP/s
Power Consumption, TDP	225 W	130 W

In the CUDA programming model [15] the GPU acts as a coprocessor to a distinguished host system that executes an adapted Fortran/C/C++ program with GPU kernel calls in it. It is on the programmer to partition the problem at hand into sub-problems that can be mapped onto the GPU's SPs appropriately. The number of CUDA threads that should execute the sub-problems is defined by the programmer using a grid-block hierarchy of threads. Thread blocks are dynamically created at runtime by the GPU scheduler and then are assigned to the SMXs for execution – thread blocks are not migrated between SMXs once their execution has started. The schedulers on the SMXs partition the thread blocks into so-called warps (groups of 32 threads each) which execute on the SPs in SIMD (more precisely: SIMT – Single Instruction Multiple Thread) manner using up to 64-way interleaved multi-threading. In this way, the per-SMX thread schedulers manage a total of up to 2048 concurrent threads, distributed over up to 16 thread blocks. During their execution, threads can access different memory layers ranging from fast on-chip shared memory – used for intra-thread-block communication, synchronization and data sharing – to texture memory with special data fetching modes, constant memory, and finally the global memory locally available on the device.

3.1 Concurrent Kernel Execution and Hyper-Q

With the Fermi GPU architecture Nvidia introduced the concurrent kernel execution (CKE) feature. CKE allows for multiple GPU kernels placed on different

CUDA streams – the means in CUDA to express independency of GPU related operations like data transfers and kernel executions – to potentially overlap during their execution. On Fermi GPUs all streams are ultimately multiplexed into a single hardware work queue, where up to 16 successive kernels in it can be scheduled to the GPU concurrently. For that to happen all 16 kernels need to be placed on different streams, and the resource requirements of all kernels – in terms of shared memory and register usage, etc. – must be compatible with the device's capabilities. States of the hardware work queue where successive kernels `A` and `B` are placed on the same CUDA stream result in false-serialization, that is, `B` and additional kernels in separate streams thereafter wait for `A` to complete before they can execute.

In Fig. 1 we illustrate a setup with multiple threads running on the host's CPU, each of which using a separate CUDA stream multiple kernels are placed on during the thread's execution. In this example host threads do not synchronize their GPU kernel invocations which for multiple kernels placed on the same stream can result in unfortunate states of the hardware work queue.

With the Kepler GPU architecture, 31 additional hardware work queues are introduced – Hyper-Q feature – and the streams `s[0..7]` in Fig. 1 are mapped to 8 different hardware queues. With more than 32 streams, some work queues are oversubscribed so that some streams alias onto the same hardware work queue potentially causing false-serialization.

Kernel Reordering on Fermi GPUs using a Proxy. A simple means to approach the false-serialization problem on Fermi GPUs is to introduce a proxy that is responsible for the actual GPU kernel invocations. Threads (processes)

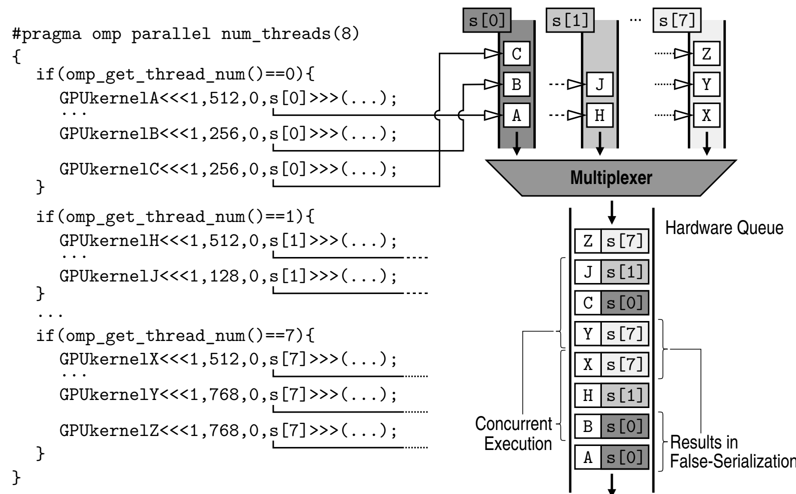

Fig. 1: False-serialization within the hardware work queue of an Nvidia Fermi GPU caused by concurrent GPU kernel invocations from within an OpenMP parallel region.

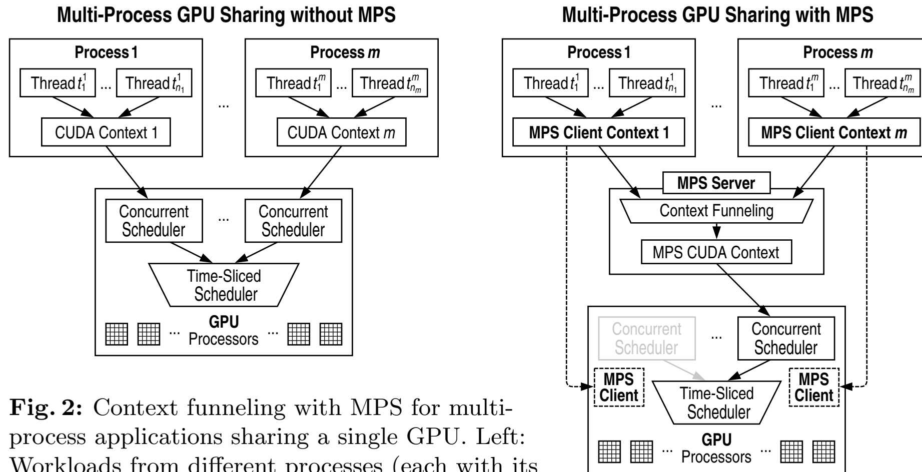

Fig. 2: Context funneling with MPS for multi-process applications sharing a single GPU. Left: Workloads from different processes (each with its own CUDA context) are scheduled to the GPU in a time-sliced manner. Right: Client-server approach with a single CUDA context hold by the server.

insert their GPU kernels into software queues that are associated with different CUDA streams, and signal the proxy to bring the respective work to the GPU for execution. The proxy traverses all software work queues in round-robin fashion and in each cycle schedules one kernel per queue – placed on the queue's CUDA stream – until all queues are empty and the program finishes. The interleaving of work from different work queues (implicit "kernel reordering") helps reduce the false-serialization within the single hardware work queue after the stream multiplexing. On Kepler GPUs the Hyper-Q feature should replace this technique.

A detailed description of the proxy approach as well as performance obtained on an Nvidia Tesla M2090 GPU can be found in [3].

Sharing GPUs across Multiple Processes using MPS. The Nvidia Multi-Process Service (MPS) is a client-server based binary compatible implementation of the CUDA API aiming at enabling multi-process applications to utilize shared GPU devices in a concurrent manner using Hyper-Q [16]. Commonly, each process has its own CUDA context for interactions with the GPU. As offloads to the GPU from within different CUDA contexts are scheduled to the GPU in a time-sliced manner, work from different processes cannot overlap resulting in the GPU is possibly underutilized. With MPS a server process (MPS server) holds and manages a single CUDA context. When a multi-process application is started all processes connect to the server – processes become MPS clients – and interact with the GPU via the MPS server. Since all work from all processes is funneled through the MPS server's CUDA context workloads from different processes can overlap in their execution (Fig. 2).

```
for i in 0 1; do
  mkdir /tmp/mps-$i /tmp/mps-log-$i
  export CUDA_VISIBLE_DEVICES=$i                   # SELECT GPU
  export CUDA_MPS_PIPE_DIRECTORY=/tmp/mps-$i       # NAMED PIPES
  export CUDA_MPS_LOG_DIRECTORY=/tmp/mps-log-$i    # LOGFILES
  nvidia-cuda-mps-control -d                       # START MPS DAEMON
done

# IF A CUDA PROGRAM CONNECTS TO THE MPS DAEMON, THE DAEMON CREATES AN
# MPS PROXY SERVER FOR THE CONNECTING CLIENT IF NOT ALREADY PRESENT.
# THE PROXY SERVER USES THE CORRECT GPU DEVICE.
unset CUDA_VISIBLE_DEVICES
mpirun -x CUDA_MPS_PIPE_DIRECTORY=/tmp/mps-0 -np 4 ./prog.x : \
       -x CUDA_MPS_PIPE_DIRECTORY=/tmp/mps-1 -np 4 ./prog.x

for i in 0 1; do
  export CUDA_MPS_PIPE_DIRECTORY=/tmp/mps-$i       # SELECT MPS DAEMON
  echo quit | nvidia-cuda-mps-control              # STOP MPS DAEMON
  rm -rf /tmp/mps-$i /tmp/mps-log-$i
done
```

Fig. 3: Execution of an OpenMPI application using two GPUs and MPS. In this example directory `/tmp/mps-[0,1]` contains named pipes for communication among the MPS daemons, servers, and the clients. `/tmp/mps-log-[0,1]` is used by the MPS daemons for log messages. All directories must be user-writable and readable.

How to start, use, and quit the MPS is demonstrated in Fig. 3 for a setup using two GPUs with OpenMPI spawning a total of 8 processes.

4 Synthetic Benchmarks

In this section we assess the Hyper-Q feature on Nvidia Kepler GPUs in the context of multi-threaded/-process applications with uncoordinated offloads to the GPU during the host computation. Our aim is to gather information about the impact of the number of concurrent kernel calls and their launch configuration on the program runtime.

Since many HPC applications are limited by the memory bandwidth of the compute system, rather than by its compute performance, for CKE it is important to know how multiple simultaneous executions on the GPU affect each other, especially with respect to sharing the GPUs memory bandwidth. For that purpose we use a synthetic kernel which applies the operation `a[0:size]=CONST+a[0:size]` to all elements of a vector of length `size` – the kernel is obviously memory bound.

Hardware and Software Setup. For the synthetic GPU benchmarks we use a dual-socket Intel Xeon E5-2603 (8 CPU cores) compute node that is equipped

with 32 GB DDR3 ECC main memory and hosts 4 Nvidia Tesla K20c GPUs with 5 GB GDDR5 ECC main memory each. The system runs a 64-bit Debian GNU/Linux 7 and has CUDA 5.0 installed with CUDA driver version 310.19. Codes are compiled with `nvcc` and `g++` version 4.6.3. Multi-process benchmarks use OpenMPI 1.6.5. For all benchmark runs we set the number of Hyper-Q connections to the maximum of 32 via `export CUDA_DEVICE_MAX_CONNECTIONS=32`.

Benchmarking Methodology. The Tesla K20c GPU is equipped with 13 SMX and offers 32 hardware work queues for CKE. On the host side we use $p = 1, \ldots, 32$ threads or $p = 1, \ldots, 16$ processes (restricted by the MPS server[3] can serve at most 16 clients) each of which placing 100 kernels of the aforementioned type on its exclusive CUDA stream one after another. We consider the program execution (a) not making use of CKE – that is, all work is placed on the default CUDA stream –, (b) using Hyper-Q either from within an OpenMP parallel region or from within different MPI processes (via MPS), and (c) using the kernel reordering approach – implicitly also making use of Hyper-Q. We measure the overall time for all kernels to complete using `clock_gettime(CLOCK REALTIME,..)`. The setup for the kernel launch is varied from 1 to 4 thread blocks per kernel invocation, with each thread block made up of 256 threads. We further vary the length of the vector that is processed by the GPU kernel from (A) 8 MB to, (B) 16 MB, to (C) 32 MB, and (D) 64 MB.

In cases where the maximum number of all concurrent thread blocks across all host threads (processes) is at most the number of SMXs (here 13) the overall program execution time should be close to the execution time of the program using a single thread (process) – the speedup over an equivalent execution using a single CUDA stream then would be equal to the number of concurrent threads.

Benchmarking Results. Figure 4 illustrates the speedup of p concurrent kernel executions according to (a), (b), or (c) over the execution of all kernels placed on the same CUDA stream.

Throughout all setups (A) – (D) it can be seen that for the 1-thread-block-per-kernel setups CKE behaves close to optimal when using the kernel reordering approach or Hyper-Q from within multiple processes. The performance of Hyper-Q from within multi-threaded host regions, however, lags significantly behind. By assigning different values to the `CUDA_DEVICE_MAX_CONNECTIONS` environment variable, we found that our benchmarking setup is sensitive to the actual number of Hyper-Q connections. With `CUDA_DEVICE_MAX_CONNECTIONS=2` the overall execution time is half the execution time of the single-CUDA-stream setup. When not explicitly set, the number of Hyper-Q connections is 8. Allowing for 32 connections increases the performance obtained with 8 connections by only 20%. To eliminate any deficiencies in the compute node configuration, we ran the benchmark on another K20c machine with CUDA 5.5 installed. The results remain the same.

[3] Before CUDA 5.5 it was named CUDA proxy server. However, we use MPS server as a synonym for CUDA proxy server hereafter.

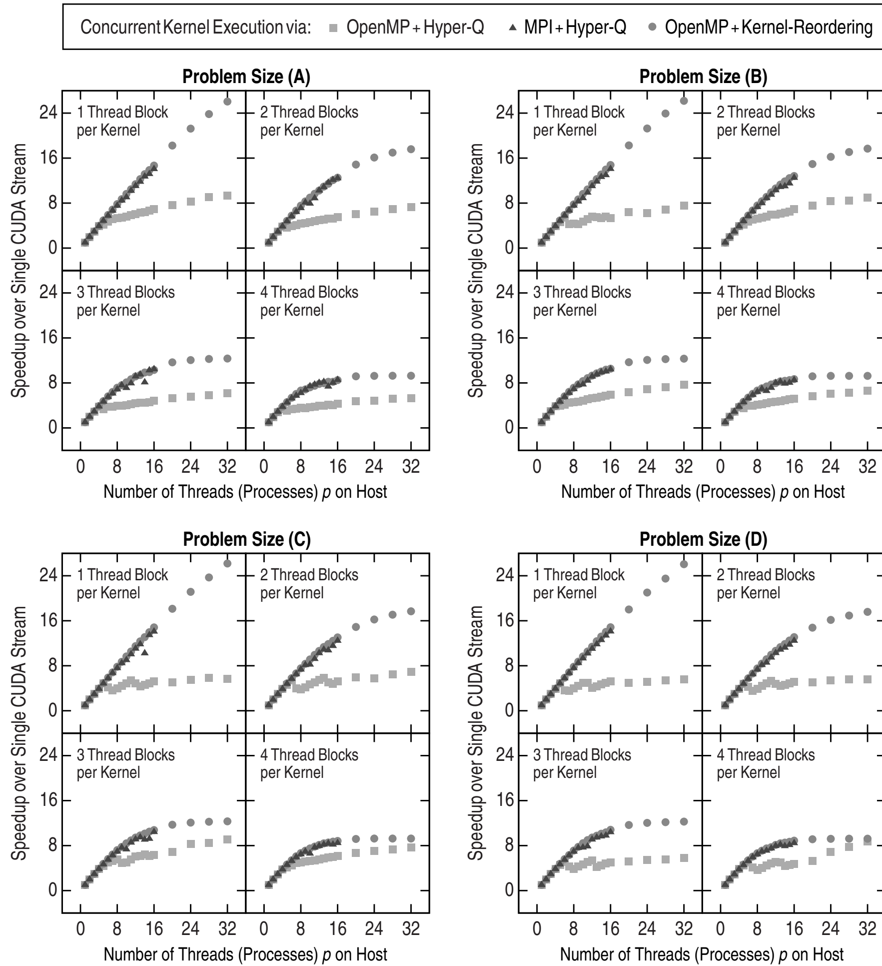

Fig. 4: Benchmarking results for the execution of up to $p \times 100$ streaming-kernels processing vectors of size (A) 8 MB, (B) 16 MB, (C) 32 MB, and (D) 64 MB each. On the host $p = 1, \ldots, 32$ threads or $p = 1, \ldots, 16$ processes each invoke 100 GPU kernels that are placed on different CUDA streams. At every time of the execution at most p kernels can execute concurrently on the GPU. The plots display the speedup over an execution where all work is placed on the same CUDA stream resulting in no concurrency.

Using the Nvidia Visual Profiler (NVVP), for 4 concurrent host threads we found all work placed on the per-thread CUDA streams is executed with almost perfect overlap, as can be seen in the upper profile in Fig. 5. Hyper-Q and kernel reordering perform equal well in this case (upper two profiles). With 8 threads on the host, the Hyper-Q profile shows up bubbles and the overall execution time increases significantly. It seems that some hardware work queues are favored

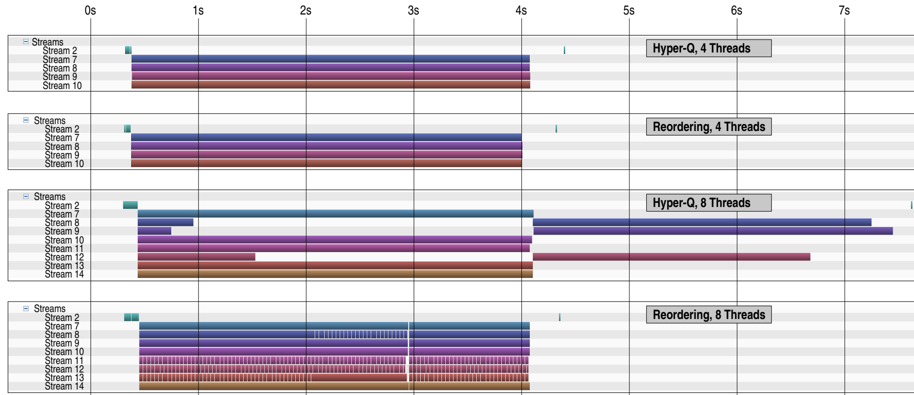

Fig. 5: Timelines showing kernel assignments to CUDA streams for different numbers of host threads. The time lines have been generated with Nvidia's Visual Profiler.

over others. For reasons that are not clear to us, this behavior starts during the execution. Right at the beginning work on all CUDA streams is brought to execution on the GPU. After about one second stream 8, 9 and 12 are suspended. When finished all work placed on the remaining streams, stream 8, 9 and 12 are revived for completion.

Since our host system provides 8 physical CPU cores the 8 host threads should not interfere with each other. We tried different thread pinning schemes using `KMP_AFFINITY=scatter|compact` respectively `sched_setaffinity()`, all without significant effect on the program behavior.

The speedup over executions with no concurrency decreases with number of thread blocks used per offload, since no concurrency means that the entire GPU device is available for the execution of a single GPU kernel. In the concurrent case, increasing the number of thread blocks per kernel launch results in more and more threads share the GPU resources, so that the overall performance starts to saturate already with a few concurrent executions. Consider, for instance, problem size (A) with one thread block per kernel launch. With more than 16 concurrent offloads, the performance does not increase linearly anymore. For the same setup, doubling the number of thread blocks per kernel launch shows that the performance increase is almost linear up to about 8 concurrent offloads. With 4 thread blocks per offload and with more than 4 concurrent kernel launches the performance starts to saturate. We also see that with increasing number of thread blocks per launch the gap between the OpenMP + Hyper-Q setup and the other two setups becomes smaller.

Figure 6 illustrates the speedup over a non-concurrent execution using two Tesla K20c GPUs. In the case of multiple processes, we ran two MPS daemons (server) each associated with one of the two GPU devices (Fig. 3). The performance decrease with the number of thread blocks per kernel launch is delayed by about a factor 2 compared to Fig. 4. Again we can see that using Hyper-Q from

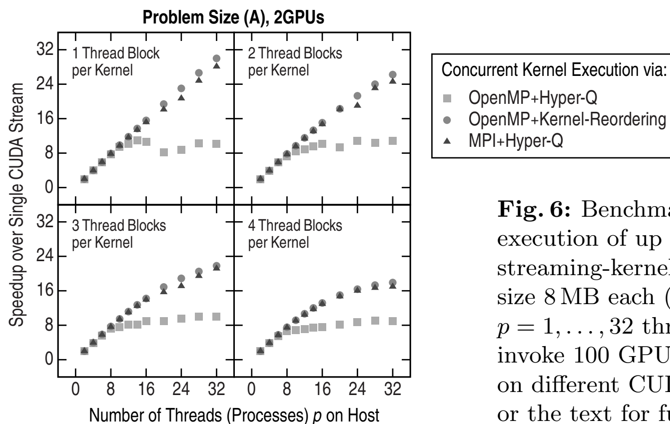

Fig. 6: Benchmarking results for the execution of up to $p \times 100$ streaming-kernels processing vectors of size 8 MB each (setup (A)). On the host $p = 1, \ldots, 32$ threads (processes) each invoke 100 GPU kernels that are placed on different CUDA streams. See Fig. 4 or the text for further explanation.

within different processes or using the kernel reordering scheme gives significant performance gains over using Hyper-Q from within threaded applications.

5 Global Local Adaptive Thermodynamics

The program package GLAT – Global Local Adaptive Thermodynamics – overcomes the problem of critical slowing down of conventional thermodynamical simulations by decomposing the conformational space into metastable subregions, which can be investigated almost independently. A typical question of pharmaceutical or biochemical applications is concerned with the prediction of solvation for a conformational ensemble.

As small drug-like molecules with $10^1 - 10^2$ atoms can exhibit more than 100 metastable states, the sampling of such a small molecule, for instance in a water environment, requires the explicit modeling of a solvation shell which contains at least an order of magnitude more atoms than the “internal molecule.” Strong scaling even for simulations of these small molecules is achieved by GLAT performs almost independent Hybrid Monte Carlo (HMC) samplings of the water solvation for many metastable states in a concurrent manner. HMC is a combination of short term Molecular Dynamics (MD) followed by a Monte Carlo (MC) weighting of the generated conformations with respect to the total energy.

Due to the discrepancy in the computational effort for the internal molecule and the surrounding water, work for calculating the solvation forces is transferred to the GPU. As the detailed conformation of the water itself is not of interest, the data for the water environment can remain on it. Only the forces of the water on the internal molecule, the potential as well as the kinetic energy of the water are required for the HMC step. Therefore, the amount of communication between host and accelerator depends mainly on the size of the internal molecule.

Figure 7 illustrates the workflow of a typical simulation within one metastable state: First the simulation is initialized with coordinates, velocities and forcefield parameters of the internal molecule in a given metastable conformation. Followed

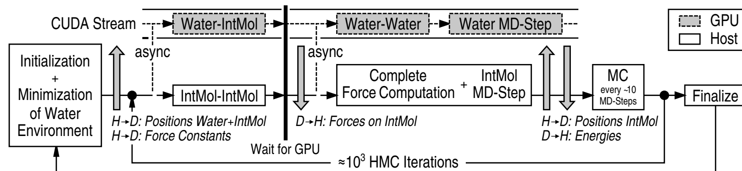

Fig. 7: Workflow of the program package GLAT. For data transfers between host and GPU "D" abbreviates "device" and "H" abbreviates "host."

by an automatic modeling and minimization of the water environment, the result is a water droplet containing the molecule of interest. Then the water data as well as the positions of the internal molecule is transferred to the accelerator where the calculation of the covalent contributions – resulting in vibrational modes of each water molecule – and the forces of the water on the internal molecule is started asynchronously to the host computation. After the invocation of the respective GPU kernel, the force calculation of the internal molecule with itself is performed by the host. At the following barrier the host receives the forces on the internal molecule due to the water and starts a molecular dynamics step, whereas the accelerator calculates all water-water interactions and performs the water MD step thereafter. Meanwhile the host updates the coordinates of the internal molecule and sends them to the GPU.

Since the whole simulation is embedded into a Hybrid Monte Carlo scheme, the host performs some statistical weightings (not time intensive) after a sequence of about 10 MD steps. The HMC sampling is repeated for a given water environment several times, until about 10^3 MD steps are reached. The final convergence check will either finish the simulation or restart it with another randomly created water environment.

Benchmarking Setup. For CPU benchmarks we use a dual-socket compute node that is equipped with two Xeon E5-2670 octa-core CPUs (Hyper-Threading enabled) and 64 GB DDR3 ECC main memory. The system runs a CentOS 6.3 Linux with kernel 2.6.32-279. We use Intel compilers version 14.0.1. For GPU benchmarks we use the compute node described in Sec. 4.

To evaluate the performance of our implementations, we consider three different sized problems: one internal molecule consisting of 27 atoms surrounded by 101 (P1), 302 (P2), and 505 (P3) water molecules. For each setup the total number of MD update steps is 1000. We consider concurrent executions of $p = 1, \ldots, 32$ simulations on the K20c and the CPU (as a reference) – codes for the CPU are tuned using AVX SIMD (Single Instruction Multiple Data) intrinsics. On GPU the compute kernels use up to 4 thread blocks of 128 threads each. On CPU each simulation uses two threads. Concurrency is achieved by means of OpenMP on the host and CUDA streams for GPU offloading. In the case of multi-GPU setups, we use one MPI process per GPU, and up to 16 host threads per MPI rank.

As a performance measure we determine the overall number of particle-particle interactions per second across all concurrent simulations. For that, we measure the time for all simulations to complete using `clock_gettime(CLOCK_REALTIME,..)`.

Benchmarking Results. Figure 8 illustrates the performance achieved for the setups (P1), (P2), and (P3). For setup (P1) the problem size is too small to run on GPU with a performance increase over a CPU-only execution. Throughout the other setups the performance gain over a CPU-only execution is between a factor 2 – 3. On the GPU side the kernel reordering scheme achieves a little more performance than the concurrent kernel execution with Hyper-Q not using reordering. Since for GLAT, all GPU kernel launches use multiple thread blocks each, the performance discrepancy between the two becomes smaller with increasing number of concurrent kernels, as can be also seen in Fig. 4.

An interesting point is the low performance gain when using two accelerators instead of just one. The performance with two GPUs increases rapidly for low numbers of concurrent simulations, whereas for more than 16 simulations it starts to saturate. Even in the case where two MPI processes are used – one process per GPU, with its own MPS server – the performance remains low. We assume

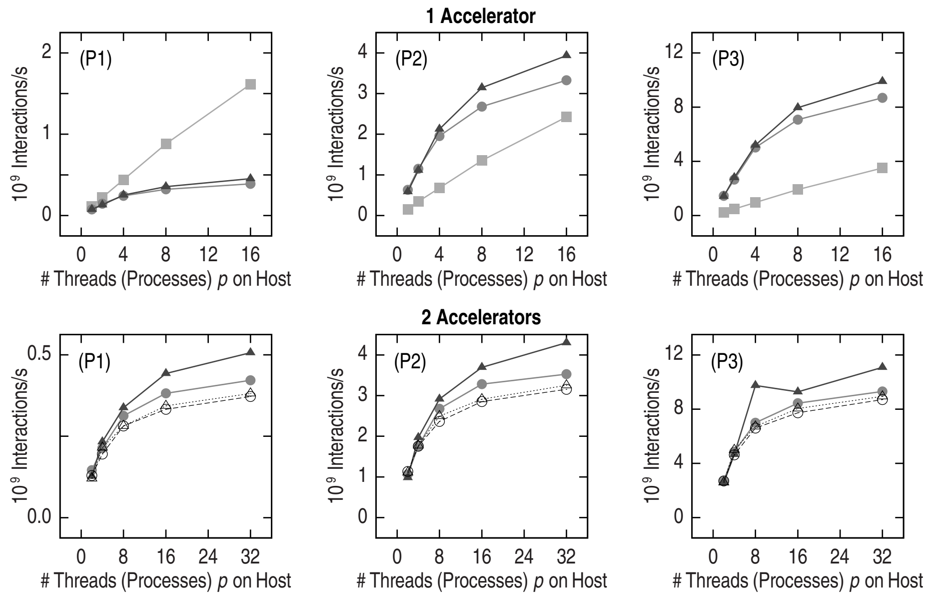

Fig. 8: Giga-interactions per second obtained with different implementations of the GLAT MD update loop. We consider both executions using 1 and 2 accelerator and compare against a CPU-only execution.

that the host system might be the bottleneck in these cases – for 32 concurrent simulations each CPU core is 4-fold oversubscribed. Since a certain portion of the MD update is performed on the host this may have a negative impact on the concurrency on the GPU.

Independently from whether one or two GPUs are used for computations, the performance gap between setups using Hyper-Q and those using kernel reordering is significant for small simulations only. With increasing problem size Hyper-Q is still behind kernel reordering, but with little discrepancy only.

6 Conclusion

Executing single small-scale workloads on GPU is insufficient to fully utilize the GPU's compute performance. A viable approach to take advantage of accelerators, however, is to execute multiple kernels in a concurrent way. For a synthetic benchmarking kernel, we found using Nvidia's Hyper-Q feature – a mechanism for concurrent offload computations on a shared GPU within multi-threaded/-process applications – give significant performance gains over a serial execution of the same amount of work on GPU. However, we encountered some deficiencies when using Hyper-Q from within multi-threaded programs causing the overall program performance be up to a factor 3 behind the expectations. We were able to recover the performance by using a kernel reordering scheme we introduced earlier for Nvidia Fermi GPUs. Using the Nvidia Visual Profiler, we found that seemingly not all hardware queues offered by Kepler GPUs are equally favored, but some of them are processed to completion before the others. Executions using Hyper-Q within multiple processes behave close to optimal with respect to concurrency on GPU. For a real-world application implementing a simulation of a small molecule solvated within a nano-droplet, we achieve about a factor 2 – 3 performance gain over an optimized multi-threaded CPU execution due to Hyper-Q.

Acknowledgments. The authors would like to thank Frank Mueller (North-Carolina State University, USA) and Tobias Kramer (Humboldt Universität zu Berlin, Germany) for giving us access to GPU platforms for benchmarking and development purposes. This work was partially supported by the Deutsche Forschungsgemeinschaft (DFG) in the framework of the Priority Programme "Software for Exascale Computing" (SPP-EXA), DFG-SPP 1648, project FFMK (Fast Fault-tolerant Microkernel), and the ENHANCE project, funded by German ministry for education and science (BMBF), grant No. 01IH11004G.

References

1. Hwu, W.m.W.: GPU Computing Gems Jade Edition. 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2011)
2. Hwu, W.m.W.: GPU Computing Gems Emerald Edition. 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2011)

3. Wende, F., Cordes, F., Steinke, T.: On Improving the Performance of Multi-threaded CUDA Applications with Concurrent Kernel Execution by Kernel Reordering. In: Proceedings of the 2012 Symposium on Application Accelerators in High Performance Computing. SAAHPC '12, Washington, DC, USA, IEEE Computer Society (2012) 74–83
4. Nvidia Corp.: Fermi Compute Architecture Whitepaper, v1.1. (2009)
5. Nvidia Corp.: Kepler GK110 Architecture Whitepaper, v1.0. (2012)
6. Guevara, M., Gregg, C., Hazelwood, K., Skadron, K.: Enabling Task Parallelism in the CUDA Scheduler. In: Workshop on Programming Models for Emerging Architectures. PMEA, Raleigh, NC (September 2009) 69–76
7. Wang, L., Huang, M., El-Ghazawi, T.: Exploiting Concurrent Kernel Execution on Graphic Processing Units. In: Proceedings of The 2011 International Conference on High Performance Computing & Simulation (HPCS 2011), Istanbul, Turkey (July 4-8 2011) 24–32
8. Wang, L., Huang, M., Narayana, V.K., El-Ghazawi, T.: Scaling scientific applications on clusters of hybrid multicore/GPU nodes. In: Proceedings of the 8th ACM International Conference on Computing Frontiers. CF '11, New York, NY, USA, ACM (2011) 6:1–6:10
9. Wang, L., Huang, M., El-Ghazawi, T.: Towards efficient GPU sharing on multicore processors. In: Proceedings of the second international workshop on Performance modeling, benchmarking and simulation of high performance computing systems. PMBS '11, New York, NY, USA, ACM (2011) 23–24
10. Chen, L., Villa, O., Krishnamoorthy, S., Gao, G.: Dynamic load balancing on single-and multi-GPU systems. In: Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on, IEEE (2010) 112
11. Chen, L., Villa, O., Gao, G.R.: Exploring Fine-Grained Task-Based Execution on Multi-GPU Systems. In: Proceedings of the 2011 IEEE International Conference on Cluster Computing. CLUSTER '11, Washington, DC, USA, IEEE Computer Society (2011) 386–394
12. Pai, S., Thazhuthaveetil, M.J., Govindarajan, R.: Improving GPGPU Concurrency with Elastic Kernels. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '13, New York, NY, USA, ACM (2013) 407–418
13. Awatramani, M., Zambreno, J., Rover, D.: Increasing GPU Throughput using Kernel Interleaved Thread Block Scheduling. In: Proceedings of the International Conference on Computer Design (ICCD). (October 2013)
14. Phillips, J.C., Stone, J.E., Schulten, K.: Adapting a message-driven parallel application to GPU-accelerated clusters. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. SC '08, Piscataway, NJ, USA, IEEE Press (2008) 8:1–8:9
15. Nvidia Corp.: CUDA C Programming Guide, v5.5. (July 2013)
16. Nvidia Corp.: Sharing a GPU between MPI processes: Multi-Process Service (MPS) Overview, Technical Brief TB-06737-003. (October 2013)