On Improving the Performance of Multi-threaded CUDA Applications with Concurrent Kernel Execution by Kernel Reordering

Florian Wende*, Frank Cordes† and Thomas Steinke*
*Zuse Institute Berlin
Takustr. 7, D–14195 Berlin-Dahlem, Germany
Email: {wende,steinke}@zib.de
†GETLIG&TAR
Bachstelzenstr. 33A, D–14612 Falkensee, Germany
Email: cordes@getlig.com

***Abstract*—General-purpose graphics processing units (GPUs) have been found to be viable solutions for large-scale numerical computations with an inherent potential for massive parallelism. In contrast, only few is known about using GPUs for small-scale computations. To have the GPU not be under-utilized for small problem sizes, a meaningful approach is to perform as many small-scale computations as possible in a concurrent manner. On NVIDIA Fermi GPUs, the concept of *Concurrent Kernel Execution* (CKE) allows for the execution of up to 16 GPU kernels on a single device. While using CKE in single-threaded CUDA programs is straightforward, for multi-threaded programs it might become a challenge to manage multiple host threads interacting with the GPU device, and in addition to have the CKE concept work properly. It can be observed that CKE performance breaks down when multiple host threads each invoke multiple GPU kernels in succession without synchronizing their actions. Since in real-world applications it is common that multiple host threads process their data independently, a mechanism is needed that helps avoiding CKE breakdown. We propose a *producer-consumer principle* approach to manage GPU kernel invocations from within parallel host regions by reordering the respective GPU kernels before actually invoking them. We are able to demonstrate significant performance improvements with this technique in a strong scaling simulation of a small molecule solvated within a nanodroplet.**

***Keywords*-GP-GPU; CUDA; concurrent kernel execution; multi-threaded applications**

I. Introduction

For certain application classes general-purpose graphics processing units (GPU) are viable compute devices for fast and energy-efficient simulations due to their massively-parallel processor design. Prominent examples of large production codes with GPU support exist in various application domains, e.g. in astrophysics [1]–[3], genomics [4]–[6], computational chemistry [7]–[11], and computational fluid dynamics [12]. The NVIDIA Fermi GPU architecture [13] introduced several new features relevant for production level simulation environments, namely a balanced double precision floating point performance and ECC for memory and data paths. With these new features the acceptance of GPUs in HPC and enterprise computing was given a further boost.

For computations with large problem sizes, a high GPU utilization can be achieved and performance improvements as expected are obtained. On the other hand, the GPU performance of computations with small problem sizes suffers not only from data transfer and management overhead but in addition from under-utilization of GPU resources.

With the Fermi architecture NVIDIA has introduced support for *Concurrent Kernel Execution* (CKE) [14], a feature that opens the door for small-scale simulations to benefit from general purpose computing on GPUs too, by executing many of them concurrently.

By means of CKE, up to 16 GPU kernels can execute concurrently on a single NVIDIA Fermi GPU device depending on the currently available local/shared/texture memory. The required condition for CKE is independence of GPU kernels which is expressed by using different CUDA streams (hereafter also simply referred to as streams) for kernel execution. GPU kernels placed on different CUDA streams are potential candidates for being executed concurrently to each other. As a consequence, for GPU kernels placed on different CUDA streams there is no guarantee for in-order kernel execution on the device, whereas GPU kernels placed on the same stream are guaranteed to be executed in the order they are invoked in the host program (for pre-Fermi GPUs, kernels are always processed in the order they are invoked in the host program).

This work is motivated by our efforts to extend the design of a multi-level parallelized application (see Sec. V) with GPU support. Multi-threaded host applications with coarse-grained parallel execution models tend to be a natural program class having an appropriate number of independent, overlapping GPU kernels which can benefit from CKE with respect to the overall throughput. Sharing the same GPU device for CKE across multiple threads of a host application was challenging prior the CUDA 4 API [15] (in CUDA 3 each host thread has its own CUDA context which is in conflict with the CKE pre-requisite of one CUDA context per device). With the CUDA 4 API, multi-threaded host programs share the same CUDA context per default.

CFP1225P-ART/12 $26.00 © 2012 IEEE
DOI 10.1109/SAAHPC.2012.12

In this study, we focus on the management of multiple host threads in order to exploit concurrent kernel execution as much as possible and so to improve the overall throughput for many small-sized GPU kernels. We are not aware of investigations on using CKE in combination with multi-threading on the CPU for setups beyond a single-GPU-kernel-per-host-thread scenario. Therefore, we are interested in the multiple-kernels-per-host-thread case in the scope of CKE. In particular, we draw on the shared CUDA context and provide a mechanism which allows for multiple host threads using the same GPU without explicit coordination of their GPU interactions. The latter point is non-trivial, as CKE with multiple independent host threads can make the whole thing break down if host threads run multiple GPU kernels on the same CUDA stream each.

Our main contributions within this study are:

- the investigation of concurrent kernel execution (CKE) within multi-threaded CUDA programs on the NVIDIA Fermi GPU architecture,
- the development of an execution model of CKE,
- the design and validation of a producer–consumer based kernel reordering mechanism to resolve the previously mentioned limitations of GPU kernel scheduling, and
- the demonstration of our approach within a real-world application.

Our paper is organized as follows: In Section II we discuss recent efforts in the area of concurrent kernel execution on GPUs. In Section III we then recall the concept of CKE together with a GPU kernel execution model, and discuss in detail the circumstance which leads to a breakdown of CKE. In Section IV we introduce our concept of *Kernel Reordering* as thin software layer to support CKE more efficiently in multi-threaded applications. The advantage of kernel reordering is validated by a synthetic demonstrator in subsection IV-C. In Section V we present the integration of our software layer into a real-world application package and demonstrate significant performance improvements when using our approach. The paper is closed with a conclusion and outlook in Section VI.

II. Related Work

We are aware of a few published studies on exploring concurrent kernel execution (CKE) in combination with multi-threaded host applications, only.

The need of a CKE feature for such situations was alluded by Guevara et al. [16] for NVIDIA GT200 graphics cards. The authors provide an approach which concentrates on merging kernels into one large kernel resulting in the execution of multiple GPU kernels on the same GPU in a concurrent manner (kernel batches).

With the introduction of the CKE feature in the NVIDIA Fermi architecture in 2010, the graphics hardware itself now is capable of scheduling multiple kernels for CKE. Investigations on using this feature for host setups with multiple threads/processes were published by El–Ghazawi et al. [15], [17], [18] 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.

Finally, in the context of migrating the highly parallelized NAMD legacy code, Phillips et al. [19] emphasized the need to schedule work from multiple CUDA streams to overlap and to synchronize concurrent work on CPU cores and GPUs more efficiently.

III. Concurrent Kernel Execution

The concurrent kernel execution (CKE) concept allows for the execution of up to 16 GPU kernels concurrently on a single NVIDIA Fermi GPU device, with independence of kernels expressed by means of streams in the CUDA framework (for further details see [14], [15]).

Basically, the degree of achievable concurrency is restricted by two factors: i) the amount of currently available local/shared/texture memory, and ii) the number of streaming multi-processors available on the GPU. For CUDA programs executing multiple GPU kernels, with some of them being dependent of each other, a further (and maybe non-obvious) criterion for achieving maximum concurrency is the order in which GPU kernels are invoked on the host side.

For in-depth understanding and further discussion, we introduce a model of how the GPU's thread scheduler [13], [20] puts GPU kernels into execution. Suppose we have a set of $N = m \times n$ GPU kernels placed onto n different streams; n sequences of m dependent GPU kernels each, for instance. When invoked on the host side, kernel $k_{i \in [0,N-1]}$ is placed into an internally managed kernel queue per device. Each time sufficient GPU resources become available the GPU's thread scheduler tries dequeuing the next kernel. Kernels are dequeued successfully if they consume at most the amount of resources currently available, and if the stream they should run onto is not in use at that point in time. Importantly for NVIDIA Fermi GPU devices is the fact that there is actually just one queue for managing GPU kernels. These then are tagged with their particular launch configuration and the CUDA stream they should execute on.

For the GPU's thread scheduler, dequeuing kernels basically works as in the following pseudo-code:

```
while(true) {
 if(!queue.empty) {
  if(streamCurrentlyNotInUse(queue.front.stream) &&
     canLaunch(queue.front))
   queue.dequeueAndLaunch()
 }
}
```

If the queue contains successive kernels which are placed onto the same stream, the GPU's thread scheduler stops

dequeuing kernels after having brought the first of them onto the GPU for execution. This also applies for queue fillings with only two successive GPU kernels placed onto the same stream followed by multiple kernels placed onto different streams each. The scheduler continues dequeuing kernels only if the currently executing GPU kernel which caused the thread scheduler to stop dequeuing kernels completes its execution.

In the following subsections we analyze this behavior using two simple test scenarios. Hereby, we restrict ourselves to single-threaded CUDA programs.

A. Ideal Concurrent Kernel Execution

With respect to the GPU thread scheduler's working method outlined before, maximum concurrency can be achieved if the kernel queue is filled up from the host program in such a way that successive kernels are placed onto different CUDA streams. The thread scheduler then should be able to detect potential for CKE each time a kernel is dequeued.

In the following pseudo-code snippet we use 16 streams only (the current CKE limit per device), and invoke a total of up to 128 single-block GPU kernels with successive kernels placed on different streams (modulo 16):

```
#define N (1024*1024)
...
__global__ void kernel(float *dA) {
 for(int i=threadIdx.x; i<N; i+=blockDim.x)
  dA[i] = sqrtf(dA[i]);
}
...
int main() {
 ...
 float *dA[128],*hA[128];
 // allocate memory for dA[],dH[],
 // assign random values to hA[],
 // and copy hA[] to the GPU (dA[])
 ...
 float time;
 cudaStream_t st[16];
 cudaEvent_t start,stop;
 // create streams and events
 ...
 for(int n=1; n<128; n++) {
  cudaEventRecord(start,0);
  for(int i=0; i<n; i++)
   kernel<<<1,512,0,st[i%16]>>>(dA[i]);
  cudaDeviceSynchronize();
  cudaEventRecord(stop,0);
  cudaEventElapsedTime(&time,start,stop);
  ...
 }
 ...
}
```

Single-block GPU kernel means that the grid of thread blocks associated with the respective kernel consists of just one thread block. In this way it is guaranteed that for the execution of the GPU kernel at most the compute resources of a single streaming multi-processor are consumed, allowing to have as many GPU kernels be executed concurrently as there are streaming multi-processors available on the device.

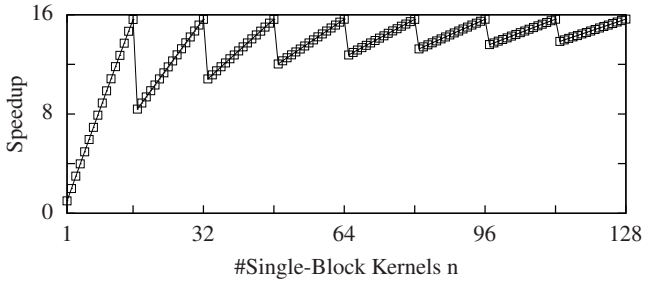

Figure 1. Ideal concurrent kernel execution within a single-threaded CUDA program. Speedups are deduced from directly comparing against sequential kernel execution with all GPU kernels placed onto the same CUDA stream. Speedups are for a NVIDIA Tesla M2090.

On a NVIDIA Tesla M2090 GPU, the speedup over a sequential execution of the respective GPU kernels converges to a factor of 16 (see Fig. 1), which corresponds to the number of streaming multi-processors of the Tesla M2090.

B. Demonstration of Concurrency Breakdown

While in the previous scenario kernels are invoked such that successive kernels are placed onto different CUDA streams (modulo 16), we now arrange kernel invocations into blocks with size $b \geq 1$. That is, groups of b GPU kernels are placed onto the same stream and are invoked back-to-back (the following code draws on the previous one):

```
int main() {
 ...
 for(int b=1; b<8; b++) {
  cudaEventRecord(start,0);
  for(int i=0; i<128; i++)
   kernel<<<1,512,0,st[(i/b)%16]>>>(dA[i]);
  cudaDeviceSynchronize();
  cudaEventRecord(stop,0);
  cudaEventElapsedTime(&time,start,stop);
  ...
 }
 ...
}
```

From the GPU thread scheduler model it can be derived that CKE will work only for GPU kernels which are the last and the first of successive blocks. For this scenario, the maximum speedup over sequential kernel execution is restricted to be at most 2 if $b > 1$.[1] The speedup is then given by the following inequality:

$$\text{Speedup} \leq \min\left\{\text{\#SM}, \frac{\text{\#Kernel}}{\text{\#Kernel} - \text{\#Blocks} + 1}\right\}, \qquad (1)$$

where #SM is the number of streaming multi-processors available on the GPU device, and $\text{\#Blocks} = \lceil \text{\#Kernel}/b \rceil$. Values for equally sized blocks of GPU kernels with b ranging from 1 to 8 are in very good agreement with speedups depicted in Fig. 2.

[1]For $b > 1$, we derive $\text{\#Kernel}/(\text{\#Kernel} - \text{\#Blocks} + 1) \leq 1/(1 - 1/b) = b/(b-1) \leq 2$. Equ. (1) then yields: Speedup $\leq \min\{\text{\#SM}, 2\}$ (≤ 2 on the Tesla M2090).

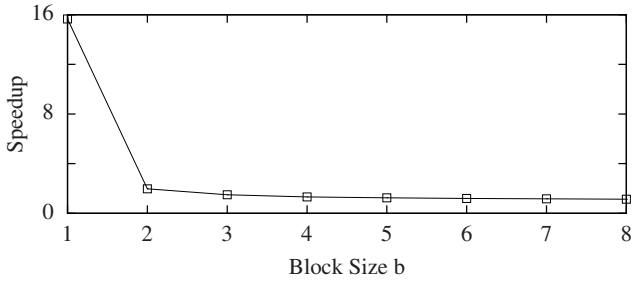

Figure 2. Concurrency breakdown within a single-threaded CUDA program. Speedups are deduced from directly comparing against sequential kernel execution with GPU kernel invocations arranged into blocks running on different CUDA streams. Speedups are for a NVIDIA Tesla M2090.

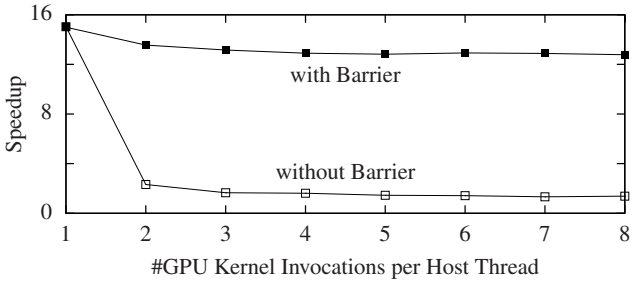

Figure 3. Speedups over sequential kernel execution for a multi-threaded CUDA program making use of concurrent kernel execution. We use 64 host-threads, each with its own CUDA stream for GPU kernel invocations. Speedups are for a NVIDIA Tesla M2090.

IV. The Kernel Reordering Mechanism

A. Multi-Threaded CUDA Programs

An obvious application scenario demanding for concurrent kernel execution (CKE) exists if multiple host threads within a multi-threaded CUDA program perform computations on both the CPU and the GPU. With the CUDA 4 API such setups are straightforward to implement since host threads which belong to the same host process share the same CUDA context and thus can use the same GPU device in a concurrent manner.

For illustration, a simple CUDA program using an OpenMP parallel region with GPU kernel invocations inside that region may look as follows (in pseudo-code style):

```
#include <omp.h>
...
#define N (1024*1024)
#define OPENMP_THREADS (16)
...
__global__ void kernel(float *dA) {
 for(int i=threadIdx.x; i<N; i+=blockDim.x)
  dA[i] = sqrtf(dA[i]);
}
...
int main(){
 ...
 float *dA[128],*hA[128];
 // allocate memory for dA[],dH[],
 // assign random values to hA[],
 // and copy hA[] to the GPU (dA[])
 ...
 #pragma omp parallel num_threads(OPENMP_THREADS)
 {
  int myId = omp_get_thread_num();
  cudaStream_t st;
  cudaStreamCreate(&st);
  for(int i=0; i<8; i++){
   kernel<<<1,512,0,st>>>(dA[myId*8+i]);
   // optional:
   //  #pragma omp barrier
  }
  cudaStreamSynchronize(st);
  cudaStreamDestroy(st);
 }
 ...
}
```

Speedups over sequential kernel execution for a setup with 64 host-threads, each using its own CUDA stream for GPU kernel invocations and no CPU computations between kernel invocations, are depicted in Fig. 3. For runtime measurements we used the OpenMP timer `omp_get_wtime()` and a `cudaDeviceSynchronize()` immediately after the parallel region instead of a `cudaStreamSynchronize()` within the parallel region.

For the case that no explicit barrier for host thread synchronization is used, we encounter "concurrency breakdown". Since GPU kernel invocations are non-blocking, multiple host threads invoke their kernels immediately in succession. The kernel queue then contains blocks of GPU kernels placed onto the same stream, and CKE can be put into effect only for a small fraction of the kernels in the queue. If instead kernels are invoked in such a manner that between successive invocations host threads are synchronized, almost all blocks within the queue have size $b = 1$, and the number of blocks with size $b = 2$ is less or equal to the number of barriers used for synchronization. Figure 3 also depicts speedups for the latter case where explicit synchronization is used.

Furthermore, Fig. 3 displays that, even in the case of explicit host thread synchronization, the maximum speedup over sequential kernel execution is below 16. The context sharing mechanism in CUDA 4 works at about 90% of the performance that can be achieved with manual context sharing [15]. Thus, we expect a speedup with the CUDA 4 context sharing on a Tesla M2090 of about 14. The speedups shown in Fig. 3 confirm our expectation.

B. Kernel Reordering

Making use of CKE from within a thread-parallel region in the host code may result in a reduced parallel throughput on the GPU when the host threads' GPU interactions are not synchronized to each other and each host thread invokes multiple GPU kernels placed onto the same CUDA stream (as demonstrated in the previous subsection). Although with

explicit host thread synchronization we have a mechanism at hand to overcome concurrency breakdown, in some cases it is not practicable to steadily synchronize all host threads with each other. If the amount of computations to be performed on the CPU during GPU kernel executions varies significantly from one host thread to another (imbalanced load), synchronizing host threads would make those with small computational amount wait for others with large computational amount.

We propose a solution for this observed concurrency breakdown by introducing a kernel reordering mechanism. Here, host threads become producers, which communicate their intention to execute certain computations on the GPU to a consumer which then actually invokes the respective GPU kernels.

A simple approach for that to work is using thread-safe queues as data structures which hold 2-tuples consisting of i) a function pointer, and ii) a unique producer identification number each. Producers are statically assigned to queues and continuously enqueue these 2-tuples. Each time any of the queues are filled up, the consumer becomes active and consecutively inspects the queues in round-robin fashion dequeuing one 2-tuple per queue in a cycle. After having dequeued a 2-tuple, the consumer invokes the respective GPU kernel and then gives a notification to the associated producer to inform that its GPU kernel was invoked. The notification is important since at the time at which the producer placed the 2-tuple into its queue it looses control over when its kernel is actually invoked. On the other hand, synchronizing to the CUDA stream the GPU kernel should be placed on requires the GPU kernel be already invoked. In order to avoid synchronizing to the CUDA stream before kernel invocation, the consumer thus gives a notification to the producer which then can use `cudaStreamSynchronize()` to wait for its kernel to finish. Figure 4 illustrates the process of enqueuing kernels by the producers, and dequeuing them by the consumer.

Aside from introducing an additional software layer between host threads (our producers) and the GPU device, the advantage of this approach is maintaining asynchronous execution while enabling a better CKE. Host threads thus can continue their computations on the CPU after having enqueued their GPU kernels. The consumer automatically reorders the GPU kernels due to dequeuing one kernel per queue per round-robin cycle. If queues are filled up, kernel invocations are such that successive kernels are placed onto different CUDA streams which then translates into the ideal CKE case described in Sec. III.

Basically, our kernel reordering mechanism can be translated into the following pseudo-code:

```
// VARIABLES:
// ==========
// newJob: shared semaphore for all
//         consumers and the producer
// notification[]: shared semaphore,
//         one per producer-consumer pair
// queue[]: array of thread-safe queues
// st[]:   array of CUDA streams, each
//         used by just one producer

int main(){
 ...
 // initialize semaphores, set up queues,
 // create CUDA streams,...
 startConsumer(); // set 'runStatus=run'
 ...
 #pragma parallel omp
 {
  // PRODUCER X:
  // ===========
  // assign producer to queue[X]
  ...
  queue[X].enqueue(kernel_1,id_X);
  sem_post(newJob);
  // do CPU computations
  sem_wait(notification[id_X]);
  cudaStreamSynchronize(st[id_X]);
  queue[X].enqueue(kernel_2,id_X);
  sem_post(newJob);
  // do CPU computations
  sem_wait(notification[id_X]);
  cudaStreamSynchronize(st[id_X]);
  ...
 }
 ...
 stopConsumer(); // set 'runStatus=stop'
                 // and 'sem_post(newJob)'
 ...
}

// CONSUMER:
// =========
// separate thread started from within the main
while(true){
 sem_wait(newJob);
 if(runStatus!=run)
  break;
 jobsDequeued=0;
 for(int i=0;i<numQueues;i++){
  if((job=queue[i].dequeue())!=NULL){
   jobsDequeued++;
   job.invokeKernel();
   sem_post(notification[job.id]);
  }
 }
 for(int i=1;i<jobsDequeued;i++)
  sem_wait(newJob);
}
// dequeue all remaining jobs and
// join the main program
```

The consumer functionality is implemented as a single separate thread which runs in an infinite loop until explicit termination by the application. Within the loop the consumer waits for events triggered by the producer threads enqueuing kernel calls ("new jobs"). The event is signaled by a semaphore (`newJob`) since waiting for a semaphore to become available does not consume any CPU cycles. During each round-robin cycle, the consumer thread counts the number of dequeued jobs in order to afterwards adjust the semaphore variable `newJob` appropriately. If signalized to leave the loop, the consumer thread dequeues all remaining

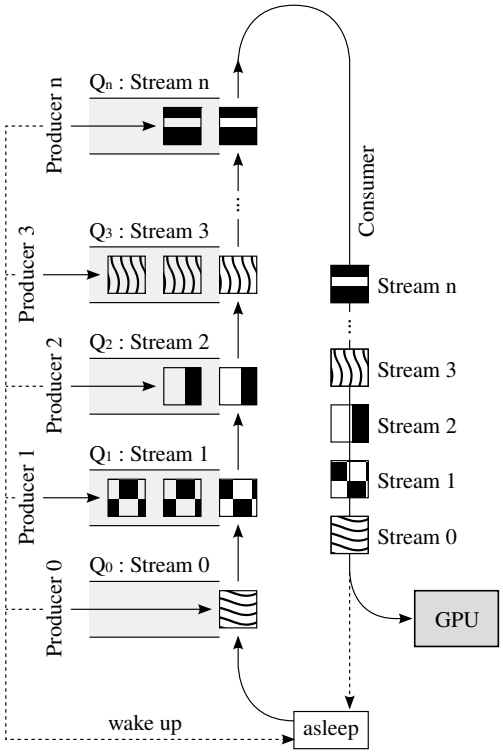

Figure 4. Illustration of a producer-consumer mechanism allowing multiple host threads (here referred to as producers) have their kernels execute on the same GPU without direct GPU interaction. GPU kernels are placed into queues associated with different CUDA streams, and then are dequeued by a consumer responsible for kernel invocations only. The way the consumer dequeues GPU kernels results in a kernel reordering with respect to the CUDA streams used.

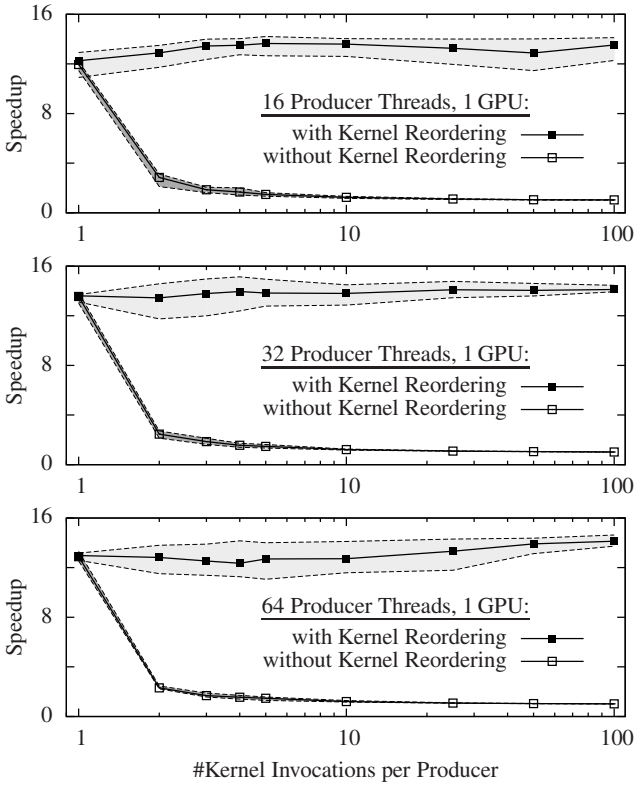

Figure 5. Speedups of the demonstrator with kernel reordering over sequential kernel execution with varying number of kernel calls on a NVIDIA Tesla M2090 GPU. Speedups are averaged over 10 independent measurements. Envelopes around measured points refer to minimum and maximum speedup values achieved for the respective setup.

kernels and then joins back into the main program.

On the producer side, producer threads place their kernels and their identification number into the queue they are assigned to. Here, the `newJob` semaphore is also used as a counter for the total number of waiting kernels in all queues. The producer can asynchronously continue its computations on the CPU. For synchronization purposes afterwards, producers each wait for the notification about kernel invocation by the consumer, and then synchronize to the CUDA stream used (by means of `cudaStreamSynchronize()`).

C. Evaluation of the Kernel Reordering Mechanism

For the evaluation of our kernel reordering mechanism we implemented a simple demonstrator. We extended our previous implementation using multiple host threads (producers) such that GPU kernels are invoked i) either directly by host threads or ii) by using our reordering mechanism. Communication between the consumer and the producers is realized by means of semaphores, which further act as barrier for each producer. Each producer thread invokes a certain number of single block GPU kernels and leaves the parallel region if and only if all of its GPU kernels were actually brought to execution by the consumer thread. Runtimes can be measured by taking a time stamp before the parallel region, and after a `cudaDeviceSynchronize()` directly behind the parallel region. In this way, we make sure that all GPU kernels were invoked and all kernels on all CUDA streams finished their execution on the device.

The demonstrator code was compiled using GCC version 4.6.2 and the CUDA 4.1 SDK. Benchmarks were performed on a Supermicro server with an X8DTG-QF+ motherboard, two Intel EP E5620 processors (quad-core Westmere, 2.4 GHz), 48 GB DDR3 RAM, and four NVIDIA Tesla M2090 GPU modules in PCIe x16 slots each.

Speedups over sequential kernel execution are shown in Fig. 5. We use up to 64 producers, each of which invoking up to 100 GPU kernels in succession. In this demonstrator the producer threads do not perform computations on the host CPU during kernel invocations.

We expect that the performance of the reordering mechanism strongly depends on the processor time slices given to the producer threads and the consumer thread. While the number of producer threads can be exceedingly large, we have just one consumer thread responsible for all GPU

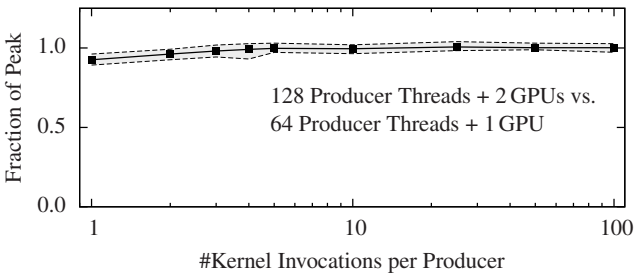

Figure 6. Ratio of the total execution times for a setup with 128 producer threads using two GPUs, and a setup of 64 producer threads using one GPU (weak scaling test). Values are for two NVIDIA Tesla M2090 GPUs connected via one PCIe hub chip to the same CPU socket the host threads were pinned to.

kernels. We therefore assigned to the consumer thread a higher priority compared to the producer threads. To ensure reproducible measurement conditions, all host threads were pinned to the CPU socket that is directly connected to PCIe hub chip of the GPU used.

As can be seen in Fig. 5, our measurements show non-negligible variations around average values which are indicated by envelopes built by minimum and maximum values of the speedups for a given setup. With our reordering mechanism, a speedup of almost a factor 14 over the sequential kernel execution can be achieved which fits well with the maximum achievable speedup for CKE within a shared CUDA context [15]. With the kernel reordering demonstrator we are able to recover the ideal concurrent kernel execution setup for GPU kernel invocations from within multi-threaded host regions (as described in Sec. III) in cases of independent producer threads.

In a next step, we extended our reordering mechanism to support multi-GPU setups. To drive multiple GPUs, the consumer thread needs to explicitly set the respective CUDA context before any kernel invocation. Figure 6 depicts the ratio of the total execution times for a setup with 128 producer threads using two GPUs, and a setup of 64 producer threads using one GPU (weak scaling test). Since this value converge very fast to the theoretical peak value of 1.0, context switching in CUDA 4 seems to work well. Our measurements reveal further that the overhead associated with the kernel reordering mechanism is negligible and is already compensated with five consecutive GPU kernel calls per producer.

V. Kernel Reordering in a Real-World Application

Our work is motivated by improving the overall throughput of multi-threaded CUDA applications using concurrent kernel execution. Therefore, we attached our developed kernel reordering software layer with a real-world application, and measured the performance improvements by means of

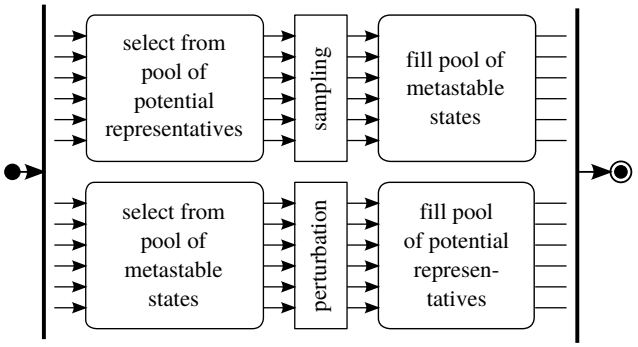

Figure 7. Parallelization model of GLAT: Each thermodynamical sampling starts with a candidate from the pool of potential conformers generated by a perturbation. Each sampling is restricted to the metastable area the starting conformer belongs to. The simulation converges if no new potential conformer can be generated by the perturbation part.

benchmarks. We briefly describe the underlying simulation method, in particular the implemented parallelization schemes which cause the need of kernel reordering in this application.

A. Parallel Thermodynamic Sampling of Molecules

To overcome the critical slowing down of conventional molecular dynamics simulations, the overall sampling in GLAT (*Global Local Adaptive Thermodynamics*) is decomposed into local samplings within metastable regions of the conformational space, which can be handled in parallel (Fig. 7). Each local distribution is generated through Hybrid Monte Carlo samplings with nearly independent Markov chains (MC). Starting with initial coordinates chosen from a pool of potential start conformers, and velocities chosen randomly from the Maxwell–Boltzmann distribution, new conformations are generated by propagation of the molecule along short periods of time. Each local sampling is restricted to the metastable region to which the starting conformation belongs to and fills up the pool of metastable states [21]. A perturbation approach selects candidates from this pool and fills the pool of start conformers until no new conformer is found with respect to energetic criteria and the coverage of the sampled space. The global overall thermodynamical distribution of molecular conformers can be approximated by the weighted sum of the local distributions.

The benefit of parallelization of the GLAT protocol correlates with the number of metastable states. Even for small drug-like molecules with 50 atoms or less the conformational space can often be decomposed into 100 and more metastable states. At the lowest level, the CPU time is consumed by computational kernels which calculate the interactions between the atoms of a molecular system. GLAT uses the MMFF94 force field [22] – an analytical model for covalent (bonded) as well as electrostatic and dispersion (non-bonded) forces. The computational expense scales linearly with the number of atoms for bonded interactions,

and quadratically for non-bonded interactions (without a cut-off distance), respectively. Traditionally, the non-bonded interactions are the objectives of parallelization strategies (e.g. [7], [9], [19], [23]). Unfortunately, these strategies are not strong scaling, i.e. a substantial speedup on many parallel units requires adequate large problem sizes, namely molecules with 1000 or much more atoms. This is not feasible for drug-like molecules being of interest in our scientific research.

Parallelization of independent samplings of metastable states in GLAT benefits from the negligible communication between these tasks. Moreover, each sampling of a metastable state is performed by more than one MC to provide accurate convergence criteria. For instance a molecule with 100 metastable states will be sampled by 400 MCs if a starting conformation for each state is known.

Typical showcases in pharmaceutical or material sciences focus on molecular libraries with millions of rather small molecules which decompose on average into hundreds of metastable states. Therefore, an overall high computational throughput for this class of application is ultimately needed.

B. GLAT Implementation

The program package GLAT comprises 50,000 lines of code written in Fortran2003. The nested parallelization strategy on metastable states in the outer loop and Markov chains in the inner loop is realized with OpenMP. CUDA routines are invoked via Fortran's ISO C binding interface to a C-wrapper.

On the GPU, the non-bonded interactions between the water molecules, the covalent contributions of each water molecule, and the interaction of the water molecules with the inner drug-like molecule are calculated. Since no cut-off distance is used for the modeled nanodroplet the number of interactions for N_{H2O} water molecules and N_{atoms} of the inner molecule is

$$\text{interactions} = 3N_{\mathrm{H2O}} \times ((3N_{\mathrm{H2O}} - 3) + 2 + N_{\mathrm{atoms}}) \,. \quad (2)$$

For each MC, the GPU has to be initialized with the initial coordinates and velocities of the water molecules, the initial coordinates of the "inner molecule", the force constants which parametrize the water-water interaction, and the water molecule interaction (Fig. 8). For each time step of the molecular dynamics propagation, the GPU receives the coordinates of the "inner molecule", and after calculating the forces it sends the contribution of the water to the forces on the "inner molecule" to the host thread. Subsequently, the coordinates and velocities of the water are propagated according to a given time step (not shown in Fig. 8). The implementation design minimizes the amount of communication between GPU and CPU for small drug-like "inner molecules".

The GLAT executable is generated with GCC version 4.6.2 and the CUDA 4.1 SDK.

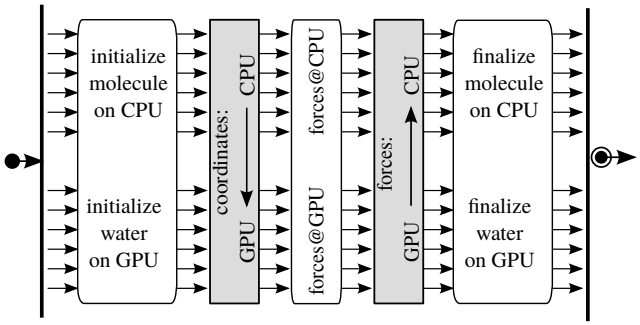

Figure 8. In GLAT for each of the Markov chains (horizontal arrows) the time consuming force calculation is subdivided into force calculations i) of the small molecule on the CPU and ii) of the water environment on the GPU, respectively. Data transfer between CPU and GPU concerns the coordinates of the small molecule and the force onto the small molecule. After initialization, the force calculation can be repeated within the MD- and Hybrid Monte Carlo framework until the finalizing checkpoint is reached.

C. Benchmark Setup for Kernel Reordering Evaluation

The benchmark suite includes simulations with 4, 8, 16, and 32 initial conformers of a drug-like molecule. For each conformer a sampling with 4 independent MCs over 1000 molecular dynamics steps is started, which results in collections of 16, 32, 64 and 128 producers, because every MC is represented by one producer. The drug-like "inner molecule" consists of 12 atoms and 4 conformational degrees of freedom. The initial conformers have been drawn from a simulation of the molecule in vacuum. For the benchmarks the initial conformers have been surrounded with 356 explicitly modeled water molecules each. A preconditional simulation for conformation analysis in vacuum is necessary to calculate the solvation energy of the "inner molecule" within a nanodroplet in a second simulation.

D. GLAT Performance Results

All performance measurements were executed on the system described in Sect. IV-C. GLAT benchmarks ran on CPUs only ($2\times4=8$ physical cores), and with one and two additional GPUs with and without kernel reordering, respectively. Speedups are derived from achieved performance measured as the number of computed pairwise atomic interactions (according Equ. 2) per time. For the GPU vs. CPU-only comparison, the maximum speedup for setups using 1 GPU and kernel reordering are about 11.1 for 128 producers, whereas maximum speedups for the same setup are about 4.2 when kernel reordering is not used (see Fig. 9). When 2 GPUs are used for computations, maximum speedups for the GPU vs. CPU-only comparison are about 13.1 and 6.5, respectively – again for the 128 producers setup with and without kernel reordering (see Fig. 9).

From Fig. 9 it can be derived, that speedups for comparing setups using kernel reordering with those not using kernel reordering range from about 1.4 to 2.6 for the 1-GPU case, and

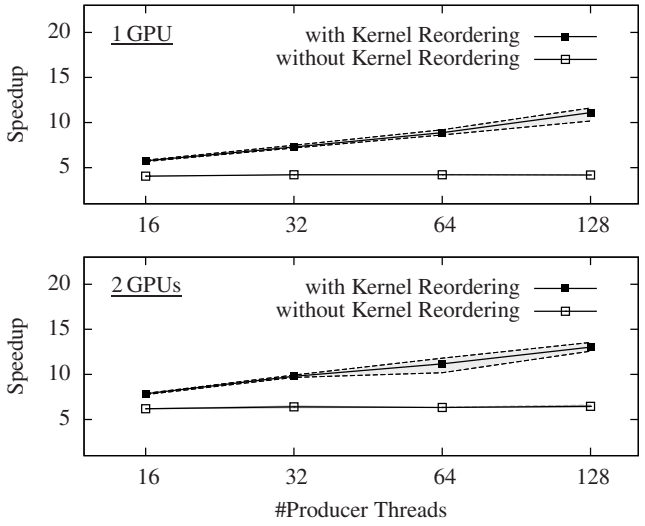

Figure 9. Speedups of a GLAT droplet simulation over a CPU-only implementation (8 cores) i) using one GPU (top) and ii) two GPUs (bottom), with and without kernel reordering, respectively. Grayed areas depict variation of results.

1.3 to 2.0 for the 2-GPU case. Both values are significantly below the theoretical peak value 14. There are several reason why this is going to happen. The first one is that for both setups with and without kernel reordering, respectively, we use two CUDA-streams for GPU computations per producer, as GPU kernels for evaluating pairwise atomic interactions split up into those including water only, and those including water and the inner-molecule. This lowers the maximum speedup by about a factor 2. Another reason which has significant impact onto the overall performance with respect to CKE is that GLAT GPU kernels are multi-block kernels. In particular, we allow for up to 8 thread blocks per kernel. Since all kernels consume lots of registers (up to 42 registers are granted per CUDA thread) it is most likely that the GPU scheduler never brings 16 or more kernels to execution at the same time.

VI. Conclusion and Outlook

For multi-threaded applications which on the one side implements parallelism at different levels, and on the other side benefit from processing many but rather small-sized workloads on the GPU, concurrent kernel execution is a key enabler to improve the overall computational throughput. We designed a kernel reordering mechanism as a thin software layer which resolves concurrency breakdowns appearing in certain multi-threaded application scenarios on GPUs. We developed a functional model of the NVIDIA Thread Scheduler of GPUs of the Fermi architectures and could validate the model by means of performance measurements for two extreme scenarios of a multi-threaded host application.

Our encouraging results demonstrate that kernel reordering has a strong effect on GPU performance if the simulation problem can be decomposed in moderate independent tasks and the problem size is relatively small.

By integrating our kernel reordering mechanism into the real-world GLAT program package we were able to demonstrate the applicability and advantages of our design. Nevertheless, transfer of the proposed GPU implementation towards a high-throughput simulation tool for a large number of molecules needs further experiments to analyze the problem dependent scheduling relations between the concurrent computations on CPU and GPU, respectively. More precisely, the impact of the size of the "inner molecule" as well as the number of solvent molecules on scheduling needs to be further investigated.

The recently disclosed NVIDIA Kepler GK110 architecture [20] includes the features *Hyper-Q* and *Dynamic Parallelism*. Hyper-Q addresses the lack of there is just one kernel queue on present Fermi GPU devices which is shared by all host threads using the same device concurrently (this is what the present paper focuses on). Other than for Fermi GPUs, with the Kepler GK110 GPU architecture CUDA streams are mapped onto up to 32 hardware managed kernel queues. For up to 32 producers this could make our approach obsolete on Kepler GPUs. But, for a larger number of producer threads (cf. scaling of GLAT in Fig. 9) we have to extend the consumer part of our middleware to support 32 hardware queues on the GPU. Note, that this would require no change on the application side. To take a possible advantage of the Dynamic Parallelism feature on Kepler, a larger part of the calling tree of the GLAT application down to the current GPU kernels, e. g. the time integrator, needs to be migrated onto the GPU device. It thus will be of great interest for us to see how concurrent kernel execution will work on the Kepler architecture, and what is the performance compared against our kernel reordering mechanism.

Acknowledgment

We would like to thank Sebastian Dressler for valuable discussions and his support. Further, the authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. This work is funded by the German BMBF project ENHANCE, grant no. 01IH11004A-G.

References

[1] R. Spurzem, P. Berczik, I. Berentzen, K. Nitadori, T. Hamada, G. M. Martinez, A. Kugel, R. Männer, J. Fiestas, R. Banerjee, and R. Klessen, "Astrophysical particle simulations with large custom GPU clusters on three continents," *Computer Science - R&D*, vol. 26, no. 3-4, pp. 145–151, 2011.

[2] R. Spurzem, P. Berczik, K. Nitadori, G. M. Martinez, A. Kugel, R. Männer, I. Berentzen, R. Klessen, and R. Banerjee, "Astrophysical Particle Simulations with Custom GPU Clusters," in *CIT*. IEEE Computer Society, 2010, pp. 1189–1195.

[3] C. J. Fluke, D. G. Barnes, B. R. Barsdell, and A. H. Hassan, "Astrophysical Supercomputing with GPUs: Critical Decisions for Early Adopters," *Publ.Astron.Soc.Austral.*, vol. 28, pp. 15–27, 2011.

[4] Y. Liu, B. Schmidt, and D. L. Maskell, "CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions," *BMC Research Notes*, vol. 3, p. 93, 2010.

[5] W. Liu, B. Schmidt, Y. Liu, G. Voss, and W. Müller-Wittig, "Mapping of BLASTP Algorithm onto GPU Clusters," in *ICPADS*. IEEE, 2011, pp. 236–243.

[6] W. Liu, B. Schmidt, and W. Müller-Wittig, "CUDA-BLASTP: Accelerating BLASTP on CUDA-Enabled Graphics Hardware," *IEEE/ACM Trans. Comput. Biology Bioinform.*, vol. 8, no. 6, pp. 1678–1684, 2011.

[7] J. E. Stone, D. J. Hardy, B. Isralewitz, and K. Schulten, "GPU algorithms for molecular modeling," in *Scientific Computing with Multicore and Accelerators*, J. Dongarra, D. A. Bader, and J. Kurzak, Eds. Chapman & Hall/CRC Press, 2011, ch. 16, pp. 351–371.

[8] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," *Journal of Computational Chemistry*, vol. 28, pp. 2618–264, 2007.

[9] P. Eastman and V. Pande, "OpenMM: A Hardware-Independent Framework for Molecular Simulations," *Computing in Science and Engg.*, vol. 12, no. 4, pp. 34–39, Jul. 2010. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2010.27

[10] N. Luehr, I. S. Ufimtsev, and T. J. Martinez, "Dynamic Precision for Electron Repulsion Integral Evaluation on Graphical Processing Units (GPUs)," *Journal of Chemical Theory and Computation*, vol. 7, pp. 949–954, 2011.

[11] I. S. Ufimtsev and T. J. Martinez, "Quantum Chemistry on Graphical Processing Units. 3. Analytical Energy Gradients and First Principles Molecular Dynamics," *Journal of Chemical Theory and Computation*, vol. 5, pp. 2619–2628, 2009.

[12] J. Beisheim, "Speed Up Simulations with a GPU," ANSYS, Inc., Tech. Rep., 2010.

[13] NVIDIA, *NVIDIA's next generation CUDA Compute Architecture: Fermi (White paper v1.1)*, 2009.

[14] ——, *NVIDIA CUDA C programming guide, v4.0*, 2011.

[15] L. Wang, M. Huang, and T. El-Ghazawi, "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, pp. 24–32.

[16] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron, "Enabling Task Parallelism in the CUDA Scheduler," in *Workshop on Programming Models for Emerging Architectures*, ser. PMEA, Raleigh, NC, September 2009, pp. 69–76.

[17] L. Wang, M. Huang, V. K. Narayana, and T. El-Ghazawi, "Scaling scientific applications on clusters of hybrid multicore/GPU nodes," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, ser. CF '11. New York, NY, USA: ACM, 2011, pp. 6:1–6:10. [Online]. Available: http://doi.acm.org/10.1145/2016604.2016612

[18] L. Wang, M. Huang, and T. El-Ghazawi, "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*, ser. PMBS '11. New York, NY, USA: ACM, 2011, pp. 23–24. [Online]. Available: http://doi.acm.org/10.1145/2088457.2088473

[19] J. C. Phillips, J. E. Stone, and K. Schulten, "Adapting a message-driven parallel application to GPU-accelerated clusters," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 8:1–8:9. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413379

[20] NVIDIA, *NVIDIA's next generation CUDA Compute Architecture: Kepler GK110 (White paper v1.0)*, 2012.

[21] F. Haack and S. Röblitz and O. Scharkoi and B. Schmidt and M. Weber, "Adaptive Spectral Clustering for Conformation Analysis," in *AIP Conference Proceedings*, vol. 1281, no. 1, 2010, pp. 1585–1588, http://link.aip.org/link/doi/10.1063/1.3498116.

[22] T. A. Halgren, "Merck molecular force field. I-V. Basis, form, scope, parameterization, and performance of MMFF94," *J. of Comp. Chem.*, vol. 17, no. 5-6, pp. 490–641, 1996.

[23] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl, "GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation," *Journal of Chemical Theory and Computation*, vol. 4, no. 3, pp. 435–447, 2008. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/ct700301q