High-Throughput Concurrent Kernel Execution by Kernel Reordering

Florian Wende
Zuse Institute Berlin
April 20, 2012
Outline

- Motivation
- Concurrent Kernel Execution
  - Ideal Concurrency
  - Concurrency Breakdown
- Kernel Reordering
- A Case Study: Molecular Thermodynamic Sampling
- Current State & Outlook
Motivation: From Large- to Small-Scale

- GPU accepted as viable accelerator for large-scale simulations in the HPC domain: $O(10^1)$ speedup over multicore CPUs for selected kernels
Motivation: From Large- to Small-Scale

- GPU accepted as viable accelerator for large-scale simulations in the HPC domain: $O(10^1)$ speedup over multicore CPUs for selected kernels

- What about (multiple) small-scale simulations?
  - GPGPU less suitable due to underutilization of the GPU
  - Fermi: up to 16 kernels may run concurrently on the same GPU
Motivation: From Large- to Small-Scale

- GPU accepted as viable accelerator for large-scale simulations in the HPC domain: $O(10^1)$ speedup over multicore CPUs for selected kernels
- What about (multiple) small-scale simulations?
  - GPGPU less suitable due to underutilization of the GPU
  - Fermi: up to 16 kernels may run concurrently on the same GPU
- Scenarios of using concurrent kernel execution
  - single host thread runs multiple kernels
  - multiple host threads run a single kernel, each
  - **multiple host threads run multiple kernels**, each (high-throughput computing on a single GPU)
Concurrent Kernel Execution

- Introduced by NVIDIA with their Fermi architecture
- Up to 16 kernels may run concurrently on the same GPU
  - Streams are used to express independency between kernels
  - Register & shared memory usage restrict concurrency
**Concurrent Kernel Execution**

- Introduced by NVIDIA with their Fermi architecture
- Up to 16 kernels may run concurrently on the same GPU
  - Streams are used to express independency between kernels
  - Register & shared memory usage restrict concurrency

**Serial kernel execution vs. concurrent kernel execution**

![Diagram showing serial vs. concurrent kernel execution]

High-Throughput Concurrent Kernel Execution by Kernel Reordering:
Ideal Concurrent Kernel Execution

Successive kernels run on different streams 'mod 16':

```c
.cudaStream_t stream[16];
...
for (i=0; i<32; i++)
    kernel<<<1,1024,0,stream[i%16]>>>(data[i]);
```

Stream 0

Stream 1

Stream 15

Max. speedup 16x on Tesla M2090

"the way it's meant to be done" (NVIDIA CUDA SDK samples)
Ideal Concurrent Kernel Execution

Successive kernels run on different streams 'mod 16':

```c
cudaStream_t stream[16];
...
for(i=0;i<32;i++)
    kernel<<<1,1024,0,stream[i%16]>>>(data[i]);
```

- 'the way it's meant to be done' (NVIDIA CUDA SDK samples)
- Max. speedup **16x** on Tesla M2090
Ideal Concurrent Kernel Execution

Successive kernels run on different streams 'mod 16':

```c
cudaStream_t stream[16];
...
for(i=0;i<128;i++)
    kernel<<<1,1024,0,stream[i%16]>>>(data[i]);
```

Runtime measurements: * up to 128 kernels on 16 streams

---

*T. El-Ghazawi et. al.—Exploiting Concurrent Kernel Execution on Graphic Processing Units*
Concurrency Breakdown

≥16 streams are used, but blocks of successive kernels run on the same stream, e.g. 32 kernels in blocks of size 2:

cudaStream_t stream[16];
...
for (i=0; i<32; i++)
    kernel<<<1,1024,0,stream[(i/2)%16]>>>(data[i]);
Concurrency Breakdown

≥16 streams are used, but blocks of successive kernels run on the same stream, e.g. 32 kernels in blocks of size 2:

```c
cudaStream_t stream[16];
...
for(i=0; i<32; i++)
    kernel<<<1,1024,0,stream[(i/2)%16]>>>(data[i]);
```

Max. speedup over serial kernel execution:

$$S \leq \min \left\{ \#SM, \frac{\#kernels}{\#kernels - \#blocks + 1} \right\}$$
Concurrency Breakdown

cudaStream_t stream[16];

...  
for (i=0; i<128; i++)
    kernel<<<1,1024,0,stream[(i/blockSize)%16]>>>(data[i]);

**Runtime measurements:** 128 kernels in blocks of size 1...8
Concurrency Breakdown

Where do we meet such kernel blockings?

- Multiple host threads, each of which runs...
- ...a single kernel multiple times in succession

```c
#pragma omp parallel // OpenMP
{
    ompId = omp_get_thread_num();
    for(i=0;i<numIterations;i++)
        kernel<<<,,stream[ompId]>>>(data[ompId][i]);
}
```
Concurrency Breakdown

Where do we meet such kernel blockings?

- Multiple host threads, each of which runs...
  - ...a single kernel multiple times in succession
  - ...multiple kernels, that are not independent of each other

```c
#pragma omp parallel // OpenMP
{
    ompId = omp_get_thread_num();
    kernel_1<<<,,stream[ompId]>>>(a[ompId], b[ompId]);
    kernel_2<<<,,stream[ompId]>>>(b[ompId], c[ompId]);
}
```
Concurrency Breakdown

Where do we meet such kernel blockings?

- Multiple host threads, each of which runs...
  - ...a single kernel multiple times in succession
  - ...multiple kernels, that are not independent of each other

```c
#pragma omp parallel // OpenMP
{
    ompId = omp_get_thread_num();
    kernel_1<<<,,stream[ompId]>>>(a[ompId],b[ompId]);
    kernel_2<<<,,stream[ompId]>>>(b[ompId],c[ompId]);
}
```

Synchronization of host threads counteracts concurrency breakdown, but is possibly *unpracticable/unwanted*
Proposed Solution: Kernel Reordering

Concurrency breakdown due to multiple host threads send their kernels to the GPU immediately in succession

Ways out:

- **Explicit synchronization** of host threads, but impractical if...
  - ...host threads are assigned to independent problems
  - ...host threads also perform computations between kernel calls
Proposed Solution: Kernel Reordering

Concurrence breakdown due to multiple host threads send their kernels to the GPU immediately in succession

Ways out:

- **Explicit synchronization** of host threads, but impractical if...
  - ...host threads are assigned to independent problems
  - ...host threads also perform computations between kernel calls
- Host threads **do not invoke GPU kernels directly** but send them to a 'third person' which brings them to the GPU
  - Requires a control thread for GPU communication
    - ...allows for **reordering** of kernels with respect to streams used
  - No need for explicit synchronization of host threads
    - ...host threads may handle computations of different complexity
Proposed Solution: Kernel Reordering

Producer-Consumer Principle:

- Producer: insert kernels into $Q_s$ assoc. with different streams
- Consumer: take kernels from $Q_s$ round-robin (one per $Q$ per cycle) and invoke them → implicit kernel reordering

High-Throughput Concurrent Kernel Execution by Kernel Reordering:
A Case Study: Mol. Thermodyn. Sampling

- Small molecule surrounded by $O(10^1..10^3)$ water molecules
- Leap-frog: LOOP( position update | force comp. | velocity update )

- Kernels depend on each other (implicit barriers via streams)
- Host threads: each run their own molecular simulation
A Case Study: Mol. Thermodyn. Sampling

Setup:
- 576 water molecules +
- 57 atoms per main molecule

**Speedup 2x-3x through kernel reordering** compared to non-reordered kernels

Hardware:
Intel Xeon E5620
Nvidia Tesla M2090
A Case Study: Mol. Thermodyn. Sampling

Setup:
- 32 . . 576 water molecules +
- 10 . . 57 atoms per main molecule

**Speedup 2x-3x through kernel reordering** compared to non-reordered kernels

Hardware:
Intel Xeon E5620
Nvidia Tesla M2090
Current State & Outlook

- Focus is on running multiple but small molecular simulations each using both CPU & GPU in a cooperative manner
- Kernels are already proven to give correct results
- Simulations can be created/deleted dynamically
Current State & Outlook

- Focus is on running multiple but small molecular simulations each using both CPU & GPU in a cooperative manner
  - Kernels are already proven to give correct results
  - Simulations can be created/deleted dynamically
- CPU & GPU parts will be extended by further computations
Current State & Outlook

- Focus is on running multiple but small molecular simulations each using both CPU & GPU in a cooperative manner
  - Kernels are already proven to give correct results
  - Simulations can be created/deleted dynamically
- CPU & GPU parts will be extended by further computations
- Improve reordering procedure by means of heuristics?
Current State & Outlook

- Focus is on running multiple but small molecular simulations each using both CPU & GPU in a cooperative manner
  - Kernels are already proven to give correct results
  - Simulations can be created/deleted dynamically
- CPU & GPU parts will be extended by further computations
- Improve reordering procedure by means of heuristics?
- Detailed analysis of concurrency in the interplay

  **GPU kernels**  **vs.**  **GPU ↔ CPU mem copies**  **vs.**  **CPU kernels**
Acknowledgment

Work partially funded by

Frank Cordes
(GETLIG&TAR)

Sebastian Dressler
Thomas Steinke
(ZIB)

BMBF Grant No. 01IH11004G