GaudiHive on the landscape of heterogeneous computing

Illya Shapoval, Marco Clemencic
CERN, UNIFE, KIPT, INFN-FE

CERN
9 September 2015
Forum on Concurrent Programming Models and Frameworks
Contents

- Introduction
- Gaudi Hive and offload computations (OC)
- Strategy towards synergetic treatment of OC in GaudiHive
- GaudiHive extension to support OC
  - Benchmarking of composite scheduling
  - Revision of requirements to consider OC as “useful”!
- Few scheduling policies for heterogeneous load in GaudiHive
Maximizing throughput
(highlight of the previous progress)

Framework

N events
To maximize system’s occupancy a dominance must be given to maximization of the intra-event task discovery rate, and NOT to ability to submit an arbitrary number of events. Predictive scheduling helps to realize this strategy, leading to:

- Reduced memory consumption
- Reduced scheduling and framework’s regular overhead
- Increased framework’s throughput!

see my talk @ FCPMF on 2015-02-11
Maximizing throughput

(Next endeavor – in this talk)

Investigate efficient ways of performing heterogeneous computations in Gaudi.
We’re aware of alternative approaches
- The STAR framework, and suchlike
  - Re-implement everything nearly from scratch
  - Have revolutionary visions ourselves...

A more pragmatic approach
- Continue with Gaudi, preserve its heritage
- Develop co-processor oriented Gaudi algorithms (see next slide)
- Augment Gaudi with the necessary infrastructure
  - To execute co-processor oriented algorithms’ efficiently
  - (Possibly) To let them co-exist with the CPU-oriented algorithms
  - Delegate to framework decision on how to maximize its throughput (see slide on “Scheduling policies”)
What do we mean by “co-processor oriented Gaudi algorithms”? 

Algorithms, conforming to Gaudi interfaces, with …

- computations, designed specifically for co-processors
  - several initiatives ongoing in LHCb
- computations, designed for CPUs, but being partially specialized for co-processors (light specialization)
- STL-compatible C++ template libraries, providing parallel algorithms and data structures
  - Thrust (NVidia)
    Back-ends: CUDA, (TBB, OpenMP)
  - Bolt (AMD)
    Back-ends: OpenCL, (C++ AMP)
Nature of “offload” algorithms
(CPU perspective)

Offload computations

Blocking load
Wider spectrum of blocking load (CPU perspective)

Offload computations

Disk & Network I/O
- intentional I/O
- unintentional I/O (swapping-in)

Blocking load

Synchronization mechanisms
- locks, condition variables, etc
CPU-blocking algorithms in Gaudi

Result in suboptimal throughput in all Gaudi incarnations:

- **Serial Gaudi**
  - no efficient treatment of blocking algorithms, by design

- **Gaudi MP**
  - less protruding effect possible (if SMT/TMT is exploited), but no flexibility to organize a throughput boost, by design

- **Gaudi Hive**
  - high, but unrealized, potential for efficient treatment (see next slides)
We use Intel TBB for thread and task management

TBB enforces non-preemptive task scheduling policy
- Intentional - the library was designed for CPU-bound computations
- (AFAIK) not going to change in near future

A blocking task wastes its software thread for the period of blocking
- Occupancy/Throughput degradation
- No scalability by the number of blocking tasks

GaudiHive had no synergy for blocking
Some ways of tolerating latency

- Make blocking calls asynchronous within an algorithm
  - Some potential, limited by depth of “out-of-order” operations

- Extend the use of multi-threading capabilities
  - Great potential, if properly used
Multi-threading for tolerating latency

Tera/Cray MTA architecture
- 128 h/w threads per scalar CPU
- zero-cost h/w context switching (CS)
- interleaved multi-threading (temporal, CS each cycle!)

Efficient built-in tolerance for blocking operations
- Southbridge-based devices
- RAM (no data caches)

(heavily used in real-time systems)

... but modern commodity CPU design is different!
Commodity multi-threading is to increase superscalar occupancy.

**Commodity CPU design**
- 2 h/w threads per superscalar core
- Costly s/w context switching
- Simultaneous multi-threading

**Blocking operations are managed by OS schedulers**
- In Linux (>2.6.22), blocking thread is preempted and displaced from the runqueue until it wakes up
- Another runnable thread is scheduled, if available

The design is efficient enough for our needs, if used properly.
Concrete measures in Gaudi Hive

- Oversubscribe CPU
  - Simulates “Tera/Cray MTA” approach
  - Mostly used in server (farm) scheduling systems
  - Delegates mgmt of blocked threads to Linux scheduler

- Can oversubscribe using TBB in Hive, in 0-approximation (see next slides)
Algorithms
- Precedence graph of real size and topology (LHCb Brunel reconstruction case): ~280 algorithms
- CPUCrunchers as algorithms for better control
  - to simulate blocking: added ability to sleep part of its execution

Workflow setups
- **Non-blocking setup**
  All algorithms: 20ms of CPU crunching
- **Blocking setup**
  All algorithms blocking: 10ms of CPU crunching + 10ms of sleeping
- **Mixed setup** (blocking algorithms randomized across the precedence graph)
  90% of algorithms: 20ms of CPU crunching
  10% of algorithms blocking: 10ms of CPU crunching + 10ms of sleeping

Testbed for benchmarking
- Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
- Single socket: 4 cores w/ HT (but HT disabled)
- L2 256KB, L3 8M B
Blocking vs non-blocking algorithms in oversubscription domain

TBB only, 20ms/algorithm, Intra-event, 1kE

- Non-blocking load demonstrate surprising constancy
Blocking vs non-blocking algorithms in oversubscription domain

- Non-blocking load demonstrate surprising constancy
- Mixed load get the maximum improvement in throughput of 8.25% at 17 threads

TBB only, 20ms/algorithm, Intra-event, 1kE
Blocking vs non-blocking algorithms in oversubscription domain

TBB only, 20ms/algorithm, Intra-event, 1kE

- Non-blocking load demonstrate surprising constancy
- Mixed load get the maximum improvement in throughput of 9.7% at 17 threads
- Blocking load – of up to 119% at 17 threads
Blocking setup, TBB only, 20ms/algorithm, Intra-event, 1kE

The more blocking an algorithm is –the more beneficial it is to oversubscribe with it, as expected
Mixed setup, TBB only, 20ms/algorithm, Intra-event, 1kE

The more blocking an algorithm is – the more beneficial it is to oversubscribe with it, as expected
Can we stay with Intel TBB to oversubscribe?

- If we had all algorithms blocking:).. But most of them won’t be
  - Is it guaranteed that real-life non-blocking algorithms will exhibit the “CPUCruncher” stability in oversubscription domain?
    - to be measured, but in general - CS will most probably kill the stability
    - possible solution (to check): TBB task arenas

- Need to discover peak throughput dynamically by adjusting the oversubscription factor, but TBB thread pool can’t be re-sized at run-time
  - Difficult to predict the factor in advance
  - Number of “active” blocking algorithms may change at run-time (see “Scheduling Policies” slide)
Alternative approach: composite scheduling

- Keep TBB scheduler for regular non-blocking algorithms
- Introduce a new custom one for blocking algorithms only
  - Starts asynchronous threads to execute blocking tasks

Experimental infrastructure has been developed

Throughputs of pure TBB, TBB w/ oversubscription and composite infrastructures

<table>
<thead>
<tr>
<th>Scheduling Approach</th>
<th>Throughput, E/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBB (4T)</td>
<td>0.7</td>
</tr>
<tr>
<td>TBB (15T)</td>
<td>0.7</td>
</tr>
<tr>
<td>Composite</td>
<td>0.7 + 6.3%</td>
</tr>
</tbody>
</table>

Throughput comparison between non-blocking and mixed approaches.
Faster co-processor incarnation is not a must!

- Intuitive: the faster co-processor algorithm is the better throughput we get
- What if it’s not faster than its corresponding CPU-incarnation?
  - All measurements up to now assumed “neutral” case: equal timings
  - Let’s change the “Mixed setup”: blocking algorithms experience degraded timing

**Throughput vs degradation of execution time of co-processor oriented algorithms**

(“Mixed setup” + degraded timing of blocking algorithms)

1. Pure TBB
   - (no oversubscr.)
2. Composite scheduling

### Chart Details:
- **Y-Axis:** Throughput, E/s
- **X-Axis:** Extent of timing degradation (CPU-oriented -> Co-processor-oriented), ms
- **Legend:**
  - Pure TBB (no oversubscr.)
  - Composite scheduling

### CPU-oriented incarnation
- 10

### Co-processor oriented incarnations
- 10
- 10
- 10
- 10
- 10
- 10
- 10
- 10

**Non-blocking part**
- 10

**Blocking part**
- 10

*Illustration of Throughput vs degradation of execution time for both CPU and Co-processor oriented algorithms.*
Two-fold algorithms in the system is a complication, but creative one
  - additional degrees of freedom, that have to be managed

Cooperative approach (modest)
  - Priority: keep CPU fully subscribed w/ CPU-oriented algorithms
  - Monitor the runqueue for the algorithms, which have co-proc. incarnations, and oversubscribe CPU with them
  - Doesn’t require the co-proc. incarnation to be faster than the CPU one.
  - Requires oversubscription factor optimization

Competitive approach (more aggressive)
  - Let the framework learn which incarnations complete faster
  - Schedule only fastest incarnations
  - Requires oversubscription factor optimization
Conclusion

- CPU oversubscription strategies are justified in the context of Gaudi.

- Composite scheduling can efficiently realize such strategies:
  - Efficient treatment of offload/(blocking) computations
  - Improved throughput

- A “winning” co-processor algorithm incarnation is always welcome, but is not a must!
  - Valid up to a limit, defined by a given data processing workflow
  - Opens up a wide window of possibilities in scheduling strategies for peak throughput
Spare slides
Intra-event occupancy in predictive mode

- Non-predictive intra-event concurrency dynamics (8 threads, 263 algorithms):
  - Speedup: 7.45 (93%)

- Non-predictive intra-event concurrency dynamics (20 threads, 263 algorithms):
  - Speedup: 14.7 (74%)

- Predictive (DRE) intra-event concurrency dynamics (8 threads, 263 algorithms):
  - Speedup: 7.91 (99%)

- Predictive (DRE) intra-event concurrency dynamics (20 threads, 263 algorithms):
  - Speedup: 19.2 (95%)