Accelerator Integration in CMS

Dr Christopher Jones
HOW 2019
19 March 2019
Considerations for Heterogeneous Computing

Scheduling CPU & Accelerator Algorithms

Configuring Heterogeneous Jobs
Scheduling CPU & Accelerator Algorithms
Concurrent CPU/Non-CPU Processing

CMS data processing framework uses a mechanism to interact effectively with non-cpu resources

Non-CPU algorithms are divided into 3 phases

- CPU stage which acquires data and transfers to non-CPU resource
- Non-CPU algorithm is then run
- When finished, a publish step is run on the CPU to move data back to CPU memory

While non-CPU algorithm runs, the CPU is available for other algorithms
Setup

TBB controls running modules

- Can have concurrent processing of multiple events

Have separate helper thread to control GPU

- Waits until enough work has been buffered before running GPU kernel
Acquire

Module acquires method called
Used to pull data from Event

Copies data to buffer
Includes a callback to start next phase of module running
External Work Starts

GPU kernel is run
Data pulled from buffer

Next waiting module can run
External Work Finishes

GPU results are copied to buffer

Callback puts Module back into waiting queue
Produce

Produce method of module is called
Pull data from buffer
Data used to create objects to put into Event
External Work in Practice

GPU

HLT base R&D have been using the facility
See Matti Kortelainen’s talk in this session

Remote FPGA

R&D project which uses remote (on or off site) FPGA for machine learning inference
See Kevin Pedro’s talk Wednesday in ‘Reconstruction and Software Triggers’

Geant V CMS Integration R&D

Control of thread passed to Geant V when new Event is ready
  • Geant V is using TBB task to run its code
  • This allows proper thread pool sharing with framework also using TBB tasks
When Geant V finishes an Event, control returns to framework
  • The finish thread often different than the start thread
See Kevin Pedro’s talk Thursday in ‘Detector Simulation’
External Work and Event Batching

Framework supports running more concurrent events than threads

- Normally not a useful configuration
  - increases memory use
  - does not increase event throughput

External Work modules can process events in batches

- module waits to run external algorithm once it has acquired a certain number of events
- Has been used in framework tests
- Not being used by any R&D projects
  - complicates module book-keeping

Some simple tests showed batching can decrease event throughput

- Have to carefully balance
  - per module speed improvements from batching
  - possible lack of available tasks for CPU to run while accumulating a batch
Configuring Heterogeneous Jobs
Heterogeneous Configuration

Want jobs for a workflow to run at any site

Want same configuration for all jobs in a workflow
  Be agnostic to the kind of hardware being used for a given job
  Hash of configuration already used by framework to segregate data from different workflows

Want to be able to keep CPU and Accelerator algorithms separate
  No need to touch working code
  The different hardware may want to group the work differently
    • e.g. CPU might want to spread over 3 modules while GPU wants them combined to 1
  Not precluding having CPU and Accelerator algorithm in same module either

Use provenance tracking to determine what technology was used
  Framework tracks what data each module uses for each Event
SwitchProducer

SwitchProducer added to configuration

Allows specifying multiple modules associated to same module label
At runtime picks one to use based on available technologies

\[
\text{foo} = \text{SwitchProducer}(
\begin{align*}
\text{cpu} &= \text{Producer}(\text{"FooProducer"}, \\
&\quad \text{input} = \text{"hits"}), \\
\text{gpu} &= \text{Producer}(\text{"FooGPUProducer"}, \\
&\quad \text{input} = \text{"raw"})
\end{align*}
\)

\[
\text{hits} = \text{Producer}(\text{"HitsProducer"}, \\
&\quad \text{input} = \text{"raw"})
\]

\[
\text{bar} = \text{Producer}(\text{"BarProducer"}, \\
&\quad \text{input} = \text{"foo"})
\]
Conclusion

CMS has a mechanism for integrating TBB and accelerators
Can support any conceivable hardware

Possible future changes only after CMS gains more practical experience
Backup Slides
Throughput Scaling Test

Approximate use of non-CPU resource
  Separate helper thread which sleeps for a set amount of time
  All waiting sleep requests handled concurrently
    • thread sleeps only for the longest requested time, not the total requested time
Once sleeping, additional sleep requests will have to wait
Denoted by ‘External Work’

Simple CPU based algorithm for testing
  algorithm sleeps for set amount of time

Require that two algorithms are needed to process each event

Test two different algorithm dependencies
  The two algorithms are independent of each other
  One algorithm depends on the results of the other algorithm
Expectations for Independent Algorithms

- # threads = # concurrent events
  - both CPU algorithms take same time

- # threads = # concurrent events
  - 1 algorithm is faster than other

- # threads = 2 * # concurrent events
  - both CPU algorithms take same time

- # threads = # concurrent events
  - 1 CPU & 1 External Work algorithm
Have two algorithms that can work in parallel on one event

- Algorithms take exactly the same amount of time to process an event
- One algorithm can be written to do external work
  - As fast as using two threads per event

### Independent Algorithm Measurements

<table>
<thead>
<tr>
<th>Event Throughput (ev/s)</th>
<th>Number of Concurrent Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>30</td>
<td>12</td>
</tr>
<tr>
<td>40</td>
<td>16</td>
</tr>
</tbody>
</table>

- External Work
- CPU only 2 threads/event
- CPU only (fast algorithm)
- CPU only
Processing Graph

Stream ID
Denotes an independent event loop

Histogram colors
**Purple**: Work has been passed to the external work controlling thread
- Between *acquire* and *produce*
- Does not mean the work is running

**Green**: a module is running on a CPU
Minimum Number of Events to Process

The external work thread can wait until a set number of events are ready to process.

Constants
- 16 concurrent events
- 16 threads

As minimum number of events approaches number of concurrent events the throughput decreases.

Relative Event Throughput vs. Minimum Number of Events to Process for External Work
Expectations for Dependent Algorithms

# threads = # concurrent events
both CPU algorithms take same time

# threads = # concurrent events
1 algorithm is faster than other

# threads = 2 * # concurrent events
both CPU algorithms take same time
No benefit from extra threads

# threads = # concurrent events
1 CPU & 1 External Work algorithm
Event processing algorithms must run sequentially

Use of external work is faster than algorithms sequentially

not as fast as if second algorithm ran on CPU as fast as it can on external worker
External Work and the CPU module have the same running times

Note the scale change
Cross Event Synchronization

Key
Opaque: Time spent in algorithm/External worker
Semi transparent: amount of time to process data in the External Worker

Can only process 1 work chunk at a time
an event must wait for its turn if it missed the most recent start of a chunk
e.g. See Event 3

External work busy for the longest event time
events with shorter processing time must still wait for the longer time
e.g. see Event 2
Number of Concurrent Events > Number of Threads

Use 16 threads

Require external work to only wait for 1 event before processing

With enough concurrent events, can get same result as if the external work module was not in the job

Event Throughput vs Concurrent Events for External Work with 16 Threads

- Event Throughput (ev/s)
- Number of Concurrent Events
- External Work
- Dropped Module
Conclusion

CMS has a mechanism for integrating TBB and accelerators

Exact event throughput benefits dependent on scheduling work to accelerator
  Waiting for enough events to accumulate can decrease throughput

The more intra-event parallelism improves event throughput
  Can schedule work on CPU and accelerator at the same time

May be able to increase event throughput at the cost of extra memory
  allow number of concurrent events to be greater than the number of CPU threads