Heterogeneous computing using HPX for the CMS trigger farm

Jean-Loup Tastet
Supervised by F. Pantaleo & V. Innocente

September 18, 2017

The two-stage CMS trigger system

- Level 1 (hardware) trigger reduces the event rate to 100 kHz.
- (Software-based) High-Level Trigger must further reduce this rate to a manageable level for physics analyses (~ 1 kHz).
- Complexity of reconstruction is combinatorial in the pile-up (PU).
- Current CPU-based farm almost reaches its limits at PU35.
- HL-LHC expected to produce PU200.

Possible solutions

- Improved algorithm: cellular automaton.
- Add accelerators (GPUs) to builder units.
  - Expensive, no load balancing.
- Add dedicated GPU nodes used by all builders.
  - Cost-efficient, maximal hardware utilization.
  - *Heterogeneous computing*
  - Requires a software framework to submit tasks to workers and fetch results over the network.

HPX

- Open-source framework for distributed computing.
- Developped on GitHub, mostly by academics (STE||AR group, ...).
- (Partial) implementation of the ParalleX execution model, designed to enable scaling beyond the Exascale.
  - Avoid explicit thread management and global synchronization.
  - Expose as much parallelism as possible using coroutines.
  - Hide latency by suspending / migrating / resuming tasks on the fly.
  - Allows to perform automatic load-balancing / work-stealing.
  - Active Global Address Space to allow migrating tasks and data in a seamless way.
- Still experimental: version 1.0 released last April.
- For our use case, we will mostly be using the remote execution capabilities.

HPX: API

- Closely follows the development of ISO C++.
- Highly integrated with the STL and Boost.
- Very clean API, based on promises / futures / continuations to manage dependencies.
- Futures represent results which are not available yet.
- Implements the Parallelism and Concurrency TS...

```
hpx::future<int> fut = hpx::async(&my_function, args...);
```

- ... and extend them to the distributed case:

```
hpx::future<int> fut =
    hpx::async(my_action, locality, args...);
```

- Actions generalize functions and allow them to be remotely executed.
- All of their arguments must be made serializable.

Minimal example

```cpp
int sum(std::vector<int> const &v)
{
    return std::accumulate(v.begin(), v.end(), 0);
}
// Defines the necessary boilerplate for calling `sum` remotely
HPX_PLAIN_ACTION(sum, sum_action);

int main()
{
    std::vector<int> v = { 1, 2, 3, 4, 5 };
    // Schedule a `sum` action to run on the current locality
    sum_action act;
    hpx::future<int> fut =
        hpx::async(act, hpx::find_here(), v);
    // Request the result
    // May suspend the current thread if not ready yet
    int res = fut.get();
    assert(res == 15);
    return 0;
}
```

HPX for the High-Level Trigger

- Finding quadruplets can be mapped to parallel architectures using an algorithm based on the cellular automaton.
- A first prototype (Patatrack) has shown that considerable gains are achievable by implementing it on GPUs.
- The goal of this project was to evaluate the suitability of HPX for offloading this step of the reconstruction to remote GPU nodes.
- An HPX-enabled version of Patatrack has been developped.

Overall design

- One CA worker (C++ class) per computing target (GPU, CPU hardware thread, ...).
- Implemented as an HPX component (class acting as a locality, with a unique address in the global address space).
- Custom constructor for each worker...
- ... but common API consisting of a single function:

```
virtual std::vector<Host::Quadruplet>
    CellularAutomaton::run(Host::Event event) = 0;
```

- ... wrapped in a component action.

Data transfer

- Handled transparently by the HPX *parcelport*.
- Data structures and classes only need to be made serializable, e.g.

```
#include <hpx/include/serialization.hpp>

namespace Host {
struct Event
{
    unsigned int eventId;
    std::vector<int> rootLayers;
    std::vector<Host::LayerHits> hitsLayers;
    std::vector<Host::LayerDoublets> doublets;
    // Same interface as Boost::serialization, but faster
    template<typename Archive>
    void serialize(Archive& ar, unsigned int version)
    {
        ar & eventId & rootLayers & hitsLayers & doublets;
    }
};
} // namespace Host
```

GPU workers

- Only GPU workers (hardest part) implemented as part of this project.
- Extending it to CPU workers should be straightforward.
- GPUs programmed and accessed using the Nvidia CUDA kernel language and API.
- Kernels in separate shared library, with standard C++ API.
- HPX-thread-safe: resources are requested by threads by querying the result of a future.

CUDACellularAutomaton::run() structure

```
std::vector<Host::Quadruplet>
CUDACellularAutomaton::run(Host::Event event)
{
    // hpx::lcos::local::channel<unsigned int> resourceQueue;
    auto f_bufferIndex = resourceQueue.get();
    // May suspend if no buffer available
    const unsigned int bufferIndex = f_bufferIndex.get();

    copyEventToPinnedBuffers(event, bufferIndex);

    /* Same thing for streams... */

    // No HPX calls beyond this point, to avoid suspending
    cudaSetDevice(gpuIndex);
    asyncCopyEventToGPU(bufferIndex, streamIndex);

    /* ... */
}
```

CUDACellularAutomaton::run() structure

```
std::vector<Host::Quadruplet>
CUDACellularAutomaton::run(Host::Event event)
{
    /* ... */

    // Define grid and block dimensions
    const std::array<unsigned int, 3> blockSize{256, 1, 1};
    const std::array<unsigned int, 3> numberOfBlocks_create{
        32, h_events[bufferIndex].numberOfLayerPairs, 1};

    // Call kernels through C++ wrappers
    kernel::create(
        numberOfBlocks_create, blockSize,
        0, streams[streamIndex],
        /* device pointers */);

    /* ... */
}
```

CUDACellularAutomaton::run() structure

```
std::vector<Host::Quadruplet>
CUDACellularAutomaton::run(Host::Event event)
{
    /* ... */

    // Fetch results
    asyncCopyResultsToHost(streamIndex, bufferIndex);

    // Reset the CA
    asyncResetCAState(streamIndex);

    // Suboptimal: blocks the HPX thread without suspending it
    // Presumed bottleneck
    cudaStreamSynchronize(streams[streamIndex]);

    /* error handling omitted */

    /* ... */
}
```

CUDACellularAutomaton::run() structure

```cpp
std::vector<Host::Quadruplet>
CUDACellularAutomaton::run(Host::Event event)
{
    /* ... */

    // Create quadruplet vector
    auto quadruplets = makeQuadrupletVector(bufferIndex);

    // Return resources, so other threads can use them
    resourceQueue.set(bufferIndex);
    streamQueue.set(streamIndex);

    // Return result
    return quadruplets;
}
```

CA creation

```
auto const localities = hpx::find_all_localities();
std::vector<hpx::future<hpx::id_type>> f_ca(nGPUs);

for (std::size_t i = 0 ; i < nGPUs ; ++i) {
    f_ca[i] = hpx::new_<CUDACellularAutomaton>(
        localities[i % localities.size()],
        gpuIndices[i],
        /* other args */
    );
}

std::vector<hpx::id_type> cellularAutomatons(nGPUs);
for (std::size_t i = 0 ; i < nGPUs ; ++i) {
    cellularAutomatons[i] = f_ca[i].get();
}
```

Naive attempt: submit all tasks at once

```
std::vector<hpx::future<QuadrupletVector>>
    f_allQuadruplets(nEvents);

// Eagerly send events to CAs in round-robin fashion
for (std::size_t n = 0 ; n < nEvents ; ++n) {
    auto const &ca = cellularAutomatons[n % nGPUs];
    f_allQuadruplets[n] =
        hpx::async(ca_action, ca, events[n]);
}

// Wait for the results
hpx::wait_all(f_allQuadruplets);
```

- Result: the machine goes out of memory fairly quickly...

Better solution: batch processing

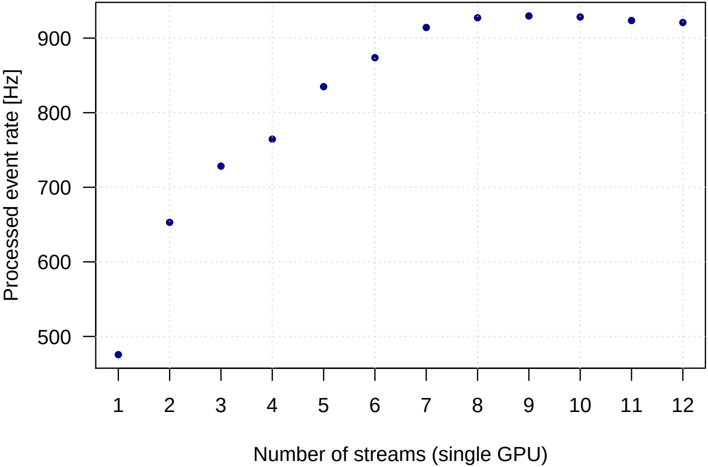
```
while (idx < nEvents)
{
    const std::size_t nextBatchIdx =
        std::min(idx + batchSize, nEvents);
    // Send futures
    for (std::size_t i = idx ; i < nextBatchIdx ; ++i) {
        const auto &ca = cellularAutomatons[i % nGPUs];
        const auto &evt = events[i % nEvents];
        f_allQuadruplets[i] = hpx::async(ca_action, ca, evt);
    }

    // Wait for the results in-order
    for (std::size_t i = idx ; i < nextBatchIdx ; ++i) {
        allQuadruplets[i] = f_allQuadruplets[i].get().size();
    }

    idx = nextBatchIdx;
}
```

Further improvements

- Wait for the results in another HPX thread, launched using `hpx::async` before even starting to send the next batch.
- Parallelize the main loop (one thread per CA worker) using `hpx::parallel::for_loop`. This automatically takes care of load-balancing.
- Send batches of events from several threads to each worker, to keep them constantly busy.

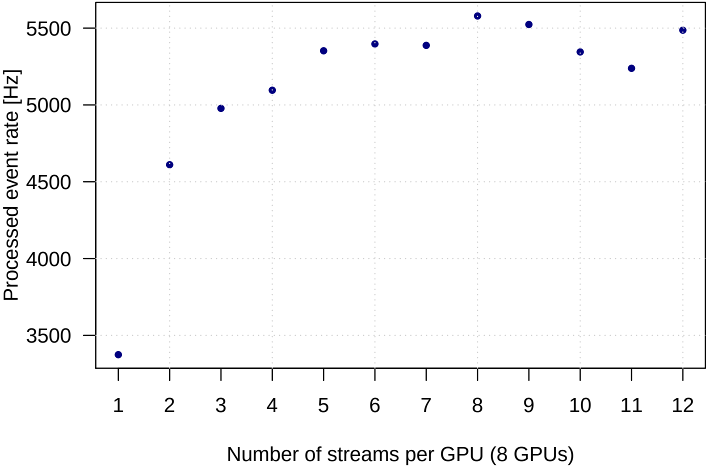
Number of threads / streams per GPU (1 GPU)

- One active HPX thread per CUDA stream.



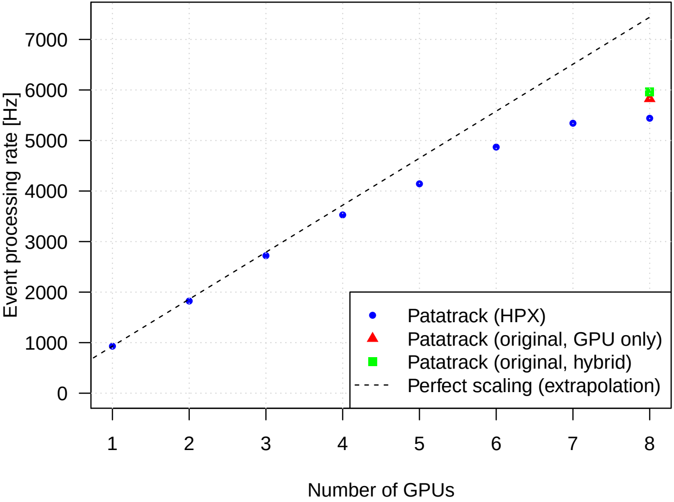
Number of threads / streams per GPU (8 GPUs)

- Max. 48 HPX threads actually executing at the same time.
- `cudaDeviceSynchronize` blocks without suspending.
- Total number of streams running in parallel is below 48.

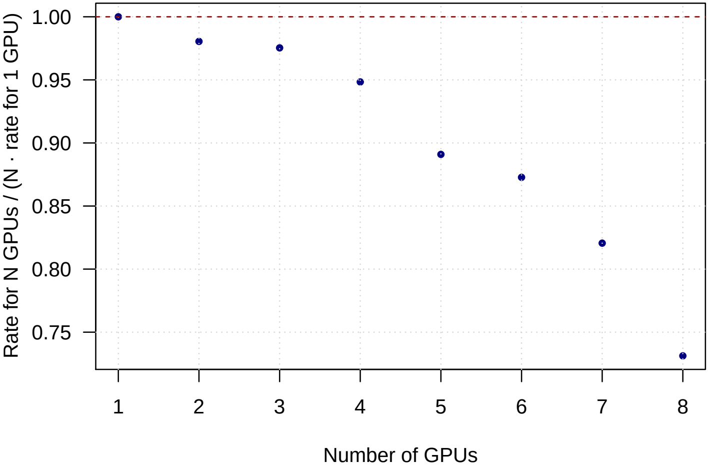


Weak scaling

- Benchmark: process 200000 events per GPU (steady state).



Weak scaling (relative)



HPX issues

- HPX still quite experimental.
- Fast-moving target.
- Documentation is very scarce and can be extremely outdated.
- Only reliable documentation: source code and unit tests.
- Heavy reliance on C macros :-(
- Heavy use of unconstrained templates $\rightarrow$ cryptic error messages.
- Only part of ParalleX is currently implemented.
- Still no standard API to manage nodes / NUMA domains / accelerators.

HPX issues

- All issues have been duly reported.

STEIIAR-GROUP/hpx **Destroying a non-empty channel causes an assertion failure** category: LCOs type: defect
#2890 opened 10 days ago by Element-126 1.1.0 — 5

STEIIAR-GROUP/hpx **Calling a __host__ function from a (__host__) __device__ function breaks HPX build with NVCC 9 in C++14 mode** category: compute compiler: nvcc platform: CUDA type: defect
#2838 opened on Aug 16 by Element-126 1.1.0 — 4

STEIIAR-GROUP/hpx **Unresolved extern variables and Segmentation fault with CUDA Clang++** platform: CUDA type: defect
#2837 opened on Aug 16 by Element-126 1.1.0 — 14

STEIIAR-GROUP/hpx **`constexpr` functions with `void` return type break compilation with CUDA 8.0.** compiler: nvcc tag: duplicate type: compatibility issue type: defect
#2835 by Element-126 was closed 29 days ago 1.1.0 — 5

STEIIAR-GROUP/hpx **`parallel/executors/execution_fwd.hpp` causes compilation failure in C++11 mode.**
#2831 by Element-126 was closed on Aug 14 — 2

STEIIAR-GROUP/hpx **HPX fails to compile with HPX_WITH_CUDA=ON and the new CUDA 9.0 RC.** platform: CUDA type: compatibility issue
#2815 by Element-126 was closed on Aug 11 1.1.0 — 3

CUDA issues

- Very hard to debug (but good tools available).
- Memory management is awful ! (it is basically C).
- NVCC is not fully standard conformant.
  - Cannot compile HPX.
  - All device code must be hidden in a shared library.
- CUDA runtime is *OS*-thread-safe but not *HPX*-thread-safe (relies on per-OS-thread global state for the device selection).

Conclusion

- HPX still quite experimental, but very promising.
- Poor documentation. Can be difficult to get started with.
- Standard-based API is a Good Thing™, could help adoption.
- Needs to mature a bit. C++ concepts would help.
- Most "blocking" issues were related to CUDA interoperability.
- There is ongoing work to address them on the HPX side.
- CPU overhead can be significant...
- ... but would be there anyway in any heterogeneous framework.
- MPI-based ad-hoc code could do the job...
- ... but HPX much more generic and extensible.