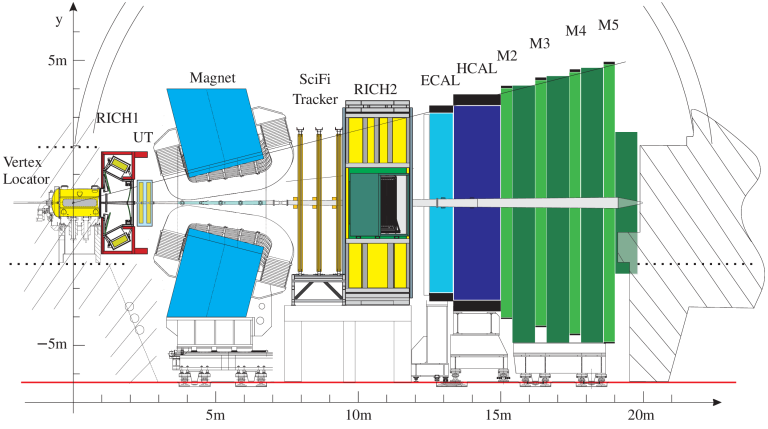
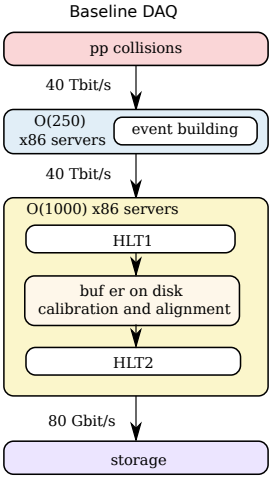
Experience in LHCb

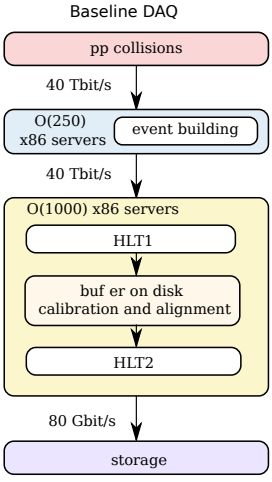
Run 3 upgrade in LHCb

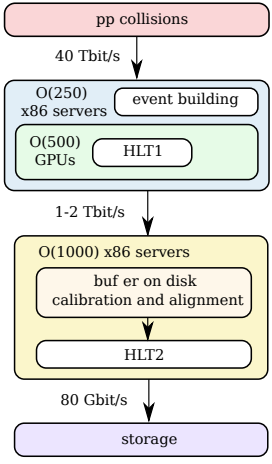


Run 3 DAQ baseline



GPUs in Run 3

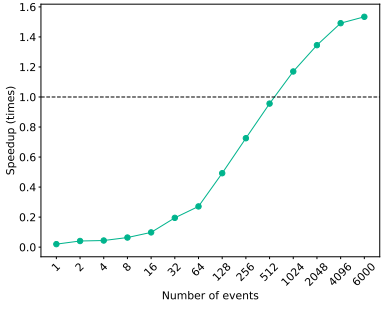

Baseline DAQ
pp collisions
40 Tbit/s
O(250)
x86 servers
event building
40 Tbit/s
O(1000) x86 servers
HLT1
buf er on disk
calibration and alignment
HLT2
80 Gbit/s
storage


pp collisions
40 Tbit/s
O(250)
x86 servers
event building
O(500)
GPUs
HLT1
1-2 Tbit/s
O(1000) x86 servers
buf er on disk
calibration and alignment
HLT2
80 Gbit/s
storage

Running efficiently on GPUs

Previous attempts in LHCb were done using a service to offload part of the work to GPUs. Single events were submitted to an service, which accumulated and offloaded several of them. However, the LHCb event size is too small (60-100 kB), not enough to occupy efficiently the GPU.

The number of events is a key factor in LHCb workload:



LHCb GPU core infrastructure

The Allen framework

The *Allen* framework is a compact, scalable and modular framework, built for running the LHCb HLT1 on GPUs.

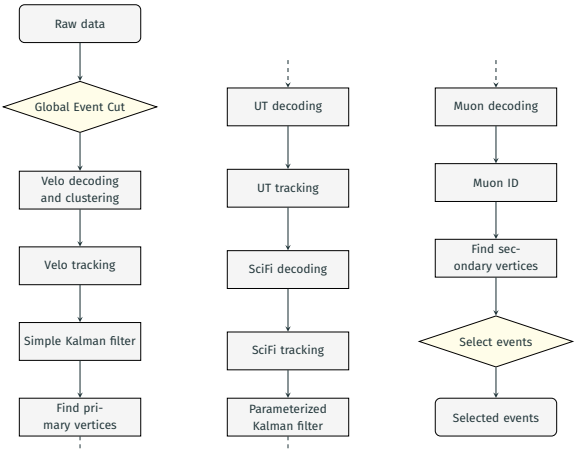
Requirements

- A C++17 compliant compiler, boost, zeromq
- CUDA v10.0 (currently C++14 is supported)

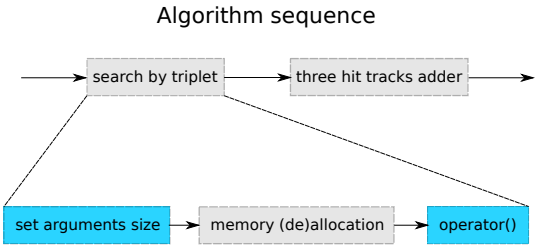
Features

- Multithreaded, multistream framework.
- Configurable static sequences.
- Pipelined stream sequence.
- Custom memory manager, no dynamic allocations, SOA datatypes.
- Built-in validation with Monte Carlo.
- Optional compilation with ROOT for generation of graphs.
- Integration with Gaudi build system.
- Cross-architecture compatibility.

Current Allen HLT1 sequence



Framework modularity



An algorithm definition (1)

```
__global__ void velo_three_hit_tracks_adder(
  uint32_t* dev_velo_cluster_container,
  uint* dev_module_cluster_start,
  Velo::TrackHits* dev_tracks,
  Velo::TrackletHits* dev_three_hit_tracks,
  bool* dev_hit_used,
  uint* dev_atomics_velo);

struct velo_three_hit_tracks_adder_t : public GpuAlgorithm {
  constexpr static auto name {"velo_three_hit_tracks_adder_t"};
  decltype(gpu_function(velo_three_hit_tracks_adder)) function {velo_three_hit_tracks_adder};
  using Arguments = std::tuple<
    dev_velo_cluster_container,
    dev_estimated_input_size,
    dev_tracks,
    dev_three_hit_tracks,
    dev_hit_used,
    dev_atomics_velo>;

  void set_arguments_size(
    ArgumentRefManager<Arguments> arguments,
    const RuntimeOptions& runtime_options,
    const Constants& constants,
    const HostBuffers& host_buffers) const;

  void operator()(
    const ArgumentRefManager<Arguments>& arguments,
    const RuntimeOptions& runtime_options,
    const Constants& constants,
    HostBuffers& host_buffers,
    cudaStream_t& cuda_stream,
    cudaEvent_t& cuda_generic_event) const;
};
```

An algorithm definition (2)

```
void velo_three_hit_tracks_adder_t::operator()(
  const ArgumentRefManager<Arguments>& arguments,
  const RuntimeOptions& runtime_options,
  const Constants& constants,
  HostBuffers& host_buffers,
  cudaStream_t& cuda_stream,
  cudaEvent_t& cuda_generic_event) const
{
  function.invoke(dim3(host_buffers.host_number_of_selected_events[0]), block_dimension(), cuda_stream)(
    arguments.offset<dev_velo_cluster_container>(),
    arguments.offset<dev_estimated_input_size>(),
    arguments.offset<dev_tracks>(),
    arguments.offset<dev_three_hit_tracks>(),
    arguments.offset<dev_hit_used>(),
    arguments.offset<dev_atomics_velo>());
}

__global__ void velo_three_hit_tracks_adder(
  uint32_t* dev_velo_cluster_container,
  uint* dev_module_cluster_start,
  Velo::TrackHits* dev_tracks,
  Velo::TrackletHits* dev_three_hit_tracks,
  bool* dev_hit_used,
  uint* dev_atomics_velo)
{
  ...
}
```

Algorithm design

All algorithms in Allen are by design **SIMD**. Conventional GPU algorithms require parallelism at two levels:

- Grid dimension - Independent groups of work
- Block dimension - Threads sharing a common cache; they can be synchronized

Algorithm design

All algorithms in Allen are by design **SIMD**. Conventional GPU algorithms require parallelism at two levels:

- Grid dimension - Independent groups of work
- Block dimension - Threads sharing a common cache; they can be synchronized

Every event in LHCb is an independent physics event. Within each event, some algorithms exhibit higher parallelizability than others. Most times, algorithms benefit from the following convention:

- Grid dimension - *Events under execution*
- Block dimension - *Intra-event parallelism*

Cross-architecture compatible

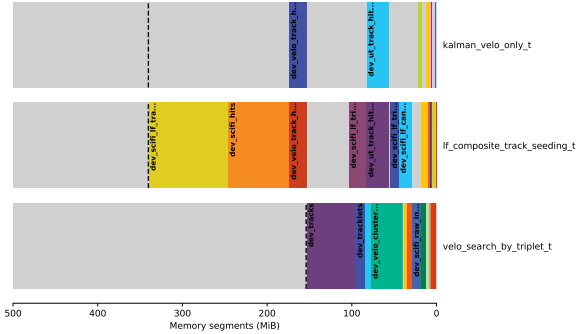
If the code has *block-dimension strided for loops*, and all `if` statements for a single thread refer to threads of index 0, then with some macros and function definitions it is possible to compile the code for CPUs:

```
// Definitions
#define __global__
#define __syncthreads()
struct GridDimensions { uint x, y, z; };
struct BlockIndices { uint x, y, z; };
struct BlockDimensions { uint x=1, y=1, z=1; };
struct ThreadIndices { uint x=0, y=0, z=0; };
thread_local GridDimensions gridDim;
thread_local BlockIndices blockIdx;
thread_local BlockDimensions blockDim;
thread_local ThreadIndices threadIdx;

...

// Kernel call excerpt
gridDim = {num_blocks.x, num_blocks.y, num_blocks.z};
for (unsigned int i = 0; i < num_blocks.x; ++i) {
  for (unsigned int j = 0; j < num_blocks.y; ++j) {
    for (unsigned int k = 0; k < num_blocks.z; ++k) {
      blockIdx = {i, j, k};
      function(std::get<I>(invoke_arguments)...);
    }
  }
}
```

Memory management

We allocate memory at the **startup** of the application. A custom memory manager assigns memory segments on demand. This is essentially the same ALICE is doing. Good practices:

- No dynamic memory allocations (within *operator()*).
- Consolidate memory into compact (AO)SOAs.
- Prefer coalesced accesses where possible.



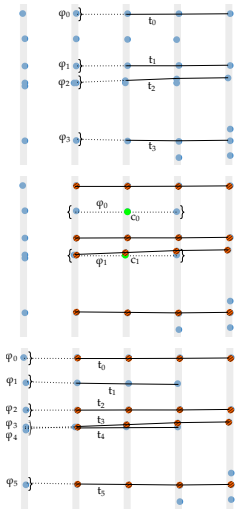
Algorithms

Example on spatial and temporal locality, data access patterns (VELO tracking)

- SIMD architectures benefit from coalesced and contiguous data access patterns
- Cache memory is limited in size
- Locality: Access patterns should restrict to a portion of memory

Search by triplet employs an SOA data structure for the VELO reconstruction, so that every access to memory has an increased probability of returning several required data in a cache line.

Additionally, modules in the VELO subdetector are visited only once, interleaving *seeding* and *forwarding* for all building tracks, maximizing spatial and temporal locality.

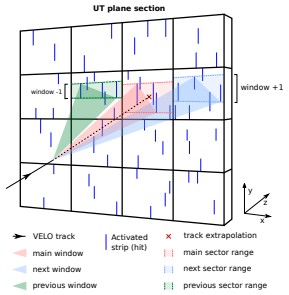


Example on spatial reductions (UT tracking)

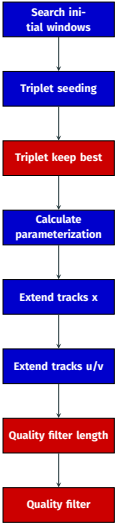
- Track reconstruction typically presents a high multiplicity of hit candidates
- Spatial reductions like KD-tree structures or search windows help reduce the dimensionality of hits under consideration

The UT subdetector is partially decoded into *sector groups*, aka groups of sectors sharing the same starting *x* coordinate. Within each sector group, hits are ordered by their *y* coordinate.

CompassUT determines search windows for each incoming Velo track. A configurable number of windows is determined, and binary searches are performed over *x* and *y*.



Each algorithm with its own grid and block size (Forward tracking)



Generates a window (max. 32 hits) according to track information. Block dimension: {**256**}.

Collects triplets from layers {0, 4, 8} or {3, 7, 11}. Block dimension: {**32, 2**}.

Keeps best 12 triplets according to their χ^2. Block dimension: {**128**}.

Calculates cubic track model. Block dimension: {**128**}.

Extends tracks to other three X layers. Block dimension: {**128**}.

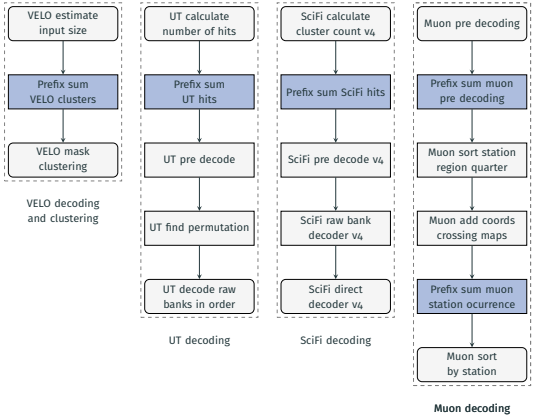
Extends tracks to stereo layers. Block dimension: {**256**}.

Cuts tracks with less than 9 hits. Block dimension: {**256**}.

Keeps best tracks according to χ^2 in X and Y. Block dimension: {**128**}.

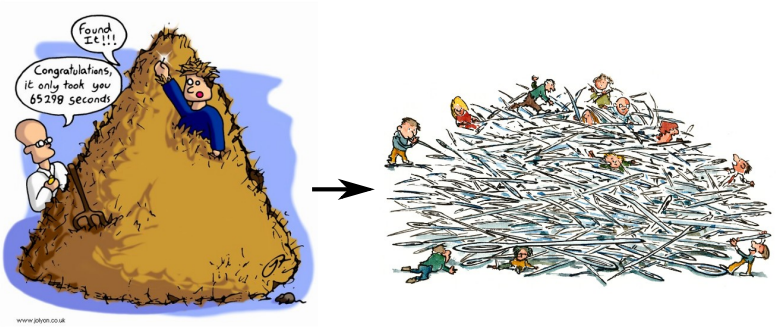
Calculate size, prefix sum (CPU), allocate

The sizes of all buffers are de/allocated with a custom memory manager. Prefix sums (*accumulated sums*) are used to determine offset / size of data buffers per event. We *offload* them to CPU.



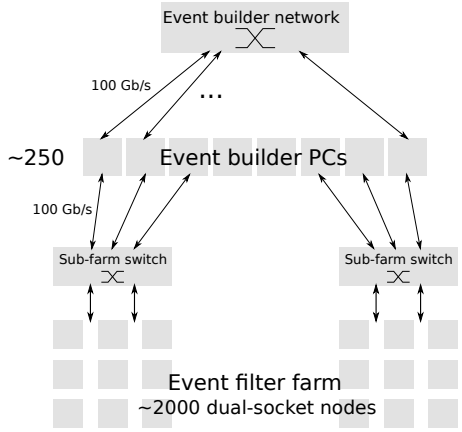
Physics performance

Due to the increase in arithmetic capacity from GPUs, it is possible to tune for better physics while fulfilling the HLT1 throughput requirement of LHCb (more in a moment).



Integration

Baseline scenario without GPUs



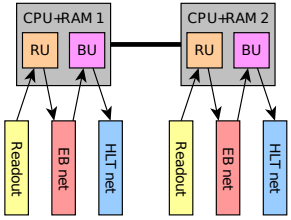
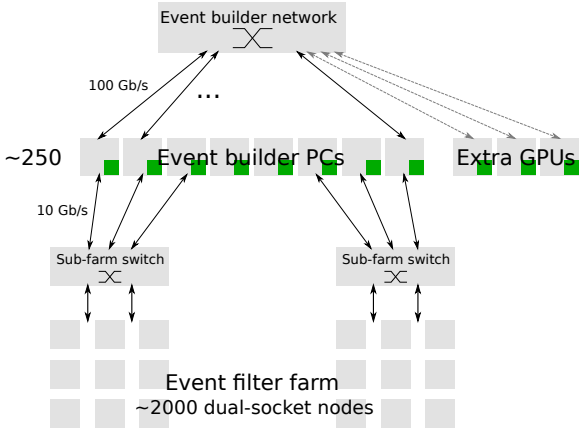


Figure 1: Event builder PC.

Event builders with GPUs



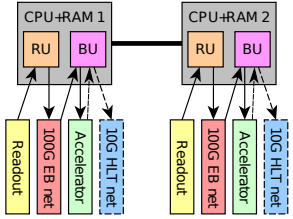
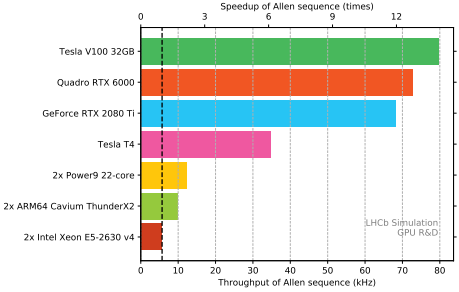


Figure 2: GPU-equipped event builder PC.

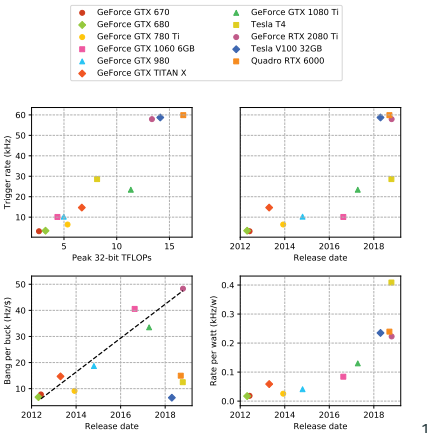
Target processing rate

In order to be able to perform the HLT1 filter inside the event builder with GPUs, the full throughput of collisions must be processed in near-time.



Opportunistic GPU usage: When there is no data-taking, GPUs could be use for something else.

Trigger rate versus peak TFLOPS, cost and power envelope across graphics cards:



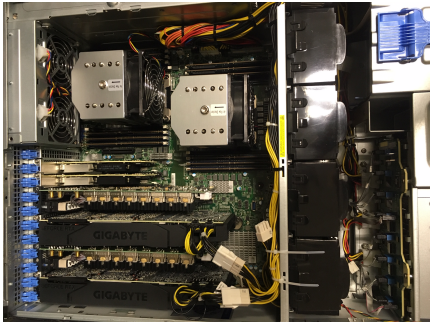
[1]

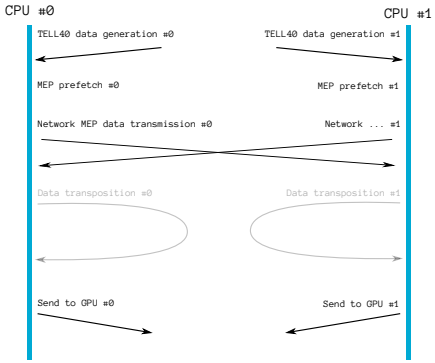
[1] *Optimization of high-throughput real-time processes in physics reconstruction*, PhD thesis, Daniel Hugo Cámpora Pérez.

Integration

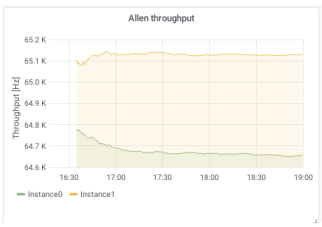
The most attractive realization of a GPU HLT1 in LHCb would be in the Event Builders.

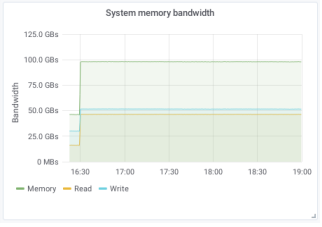
Many aspects need to be demonstrated, such as CPU consumption, memory consumption and throughput, airflow, thermal stability, GPU performance stability, network throughput...

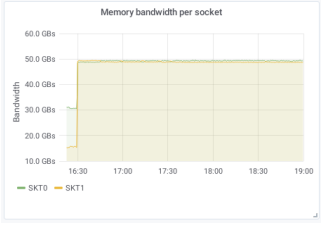


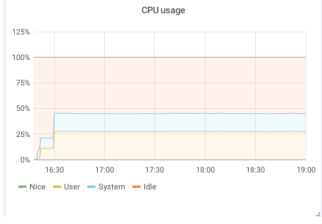


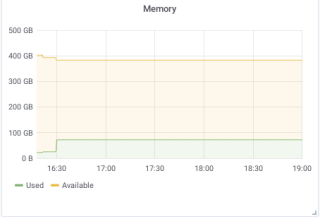
Selected integration test results

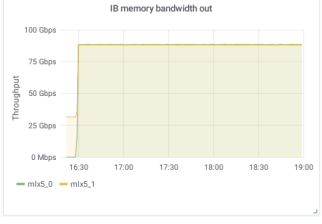












Conclusions

Run 3

- ALICE and LHCb will have **full software** data processing in Run 3.
- Both experiments have developed GPU-based systems that are ready for Run 3.

- ALICE expects **100**$\times$ more data, and do an online compression of events.
- Most of the code exist for GPUs (baseline), and there is room for more.
- Data management incurs overhead, more consecutive components on the GPU would remove this overhead.

- LHCb expects about **40**$\times$ more data, and does a two-stage trigger.
- The entire sequence of the first stage *HLT1* has been developed for GPUs.
- The GPU HLT1 is being studied as an alternative solution to the baseline (CPUs).

Components

- Both ALICE and LHCb use **custom memory managers** for the GPU to avoid the cost of allocating / deallocating.
- Both have **cross-architecture compatibility** by a switch at compile time.

In terms of integration,

- ALICE has more experience with a DAQ system with GPUs.
- Next steps are to have a better GPU usage in synchronous / asynchronous stages.

- LHCb needs proof that we are not breaking the experiment.
- Integration and burnout tests are ongoing.
- In terms of hardware, the system would occupy space in the Event Builders.
- In terms of software, HLT1 is its own application, and needs to be able to speak to certain parts of Gaudi and produce output understandable by HLT2 application.

Algorithms

Both have shown efficient implementations of GPU tracking algorithms and others:

- In ALICE, a *global method* is used to find track seeds, which maps naturally to parallel architectures.
- In LHCb, a *local method* exploits parallelism in modules (VELO), and track seeds are worked on in the following algorithms (PV, UT, Forward tracking, Kalman).
- LHCb requires processing many events in parallel to saturate the GPU, due to the tiny event size compared to ALICE (100 kB versus 50 MB).
- ALICE processes several time frames in parallel to saturate modern GPUs.

Good practices for GPU

In general, good practices of SIMD are useful in GPUs, and GPUs have some additional ones:

- Division of work into parallelizable independent tasks; blocks of threads can communicate (ie. wider vector widths).
- Data structures AOS vs SOA vs AOSOA, temporal and spatial locality.
- Dynamic memory allocation is costly, and more so on GPUs.

- GPUs are good for arithmetic workload with few branches.
- On the other hand, they are not just salvation: too high multiplicities will still run slow, and have to be worked out.
- If possible, it is better to run contiguous reconstruction steps on the GPU.

GPUs in HEP

It has been amply shown that GPUs are efficient architectures for HEP workloads.

- They require hard work, and they provide potential better physics and throughput.
- Heterogeneous computing is becoming more of a reality.

In particular, the experience in ALICE and LHCb show that this effort scales to newer architectures.

- All the lessons learned in modern CPUs (SIMD) transfer to GPUs.
- Learning curve is not steep, many tools to get started.

Looking forward to seeing HEP GPU applications grow in the next few years.

Backup

CPU support: CUDA code

Consider the following CUDA code:

```
1  constexpr int N = 32;
2  __global__ void saxpy_plus(float* x, float* y, const float a) {
3    y[threadIdx.x] = x[threadIdx.x] * a + y[threadIdx.x];
4    __syncthreads();
5    if (threadIdx.x < 10) {
6      y[i] += 1;
7    }
8    if (threadIdx.x == 11) {
9      y[threadIdx.x] += 20;
10   }
11 }
12 ...
13 saxpy_plus<<<1/*blocks*/, M, /*threads*/ N>>>(x, y, a);
```

- The number of threads is set statically to N=32.
- The statement in line 3 makes assumptions of the number of threads.
- The two `if` statements also make assumptions of the number of threads (they require at least 11 threads).

CPU support: Flexible code

In contrast, consider this code:

```
constexpr int N = 32;
__global__ void saxpy_plus(float* x, float* y, const float a) {
  for (int i=threadIdx.x; i<N; i+=blockDim.x) {
    y[i] = x[i] * a + y[i];
  }
  __syncthreads();
  for (int i=threadIdx.x; i<10; i+=blockDim.x) {
    y[i] += 1;
  }
  if (threadIdx.x == 0) {
    y[11] += 20;
  }
}
...
saxpy_plus<<<1/*blocks*/, M, /*threads*/ 1>>>(x, y, a);
```

- A call to `saxpy_plus` with any number of threads will produce the same result.

CPU support: A CPU version

If the code has *block-dimension strided for loops*, and all `if` statements for a single thread refer to threads of index 0, then with some macros and function definitions it is possible to compile the code for CPUs:

```
// Definitions
#define __global__
#define __syncthreads()
struct GridDimensions { uint x, y, z; };
struct BlockIndices { uint x, y, z; };
struct BlockDimensions { uint x=1, y=1, z=1; };
struct ThreadIndices { uint x=0, y=0, z=0; };
thread_local GridDimensions gridDim;
thread_local BlockIndices blockIdx;
thread_local BlockDimensions blockDim;
thread_local ThreadIndices threadIdx;

...

// Kernel call excerpt
gridDim = {num_blocks.x, num_blocks.y, num_blocks.z};
for (unsigned int i = 0; i < num_blocks.x; ++i) {
  for (unsigned int j = 0; j < num_blocks.y; ++j) {
    for (unsigned int k = 0; k < num_blocks.z; ++k) {
      blockIdx = {i, j, k};
      function(std::get<I>(invoke_arguments)...);
    }
  }
}
```

Integration test setup (1)

For our first test, we setup a single server with:

- Supermicro server
- 2 $\times$ Intel Xeon Silver 4114
- 376 GB of memory
- Differences wrt. candidate server: Cascade Lake (better PCIe performance), different chassis (better thermals)

It has three PCIe Gen3 16x slots per socket. Two of those are double width. Configuration on each socket:

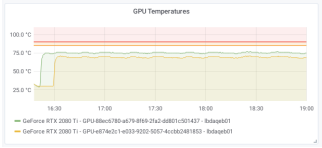
- Infiniband EDR card (100 Gbps)
- TELL40
- Gigabyte GeForce RTX 2080 Ti

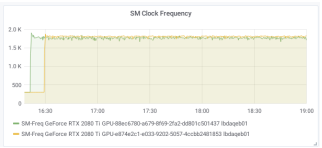
Integration test setup (2)

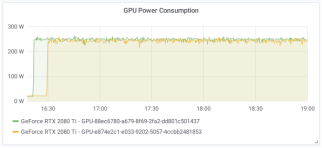
Notes:

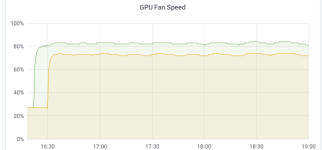
- The TELL40 can generate data into the server memory on each socket.
- Both network cards are connected back to back. A flow can then be simulated as if coming from the event building application.
- Each GPU can process data independently from each other. Two GPU applications are run, each one attached to a different GPU.

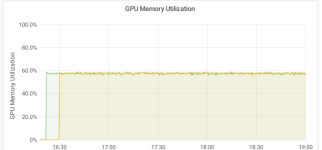
More integration test results (1)

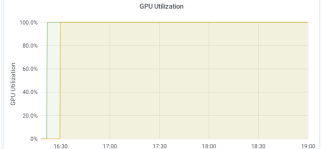












More integration test results (2)