GPU hackathon report

Scott Snyder

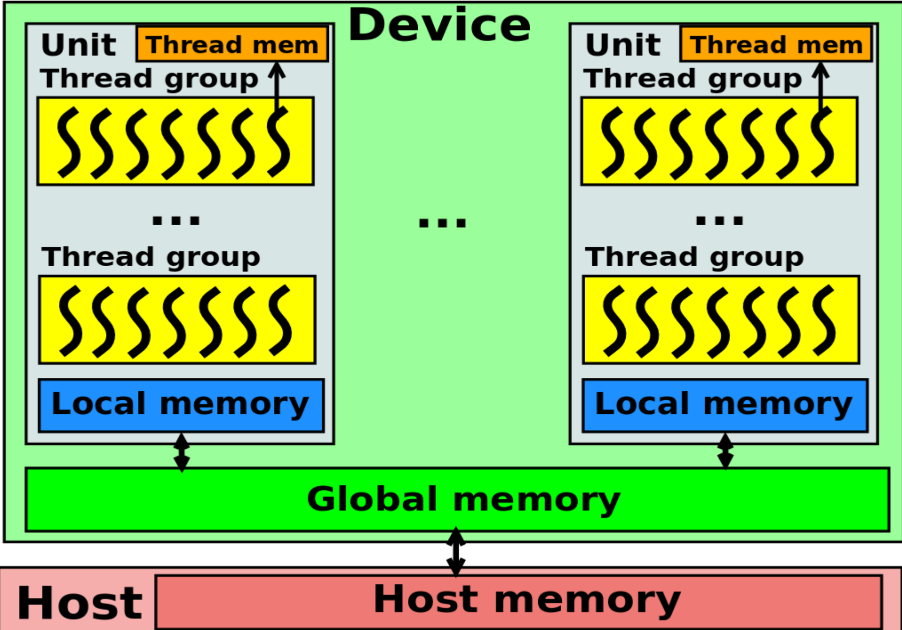
Brookhaven National Laboratory, Upton, NY, USA

Oct 25, 2018

BNL GPU hackathon

- Sep. 17–21.
- Invite groups to come to work on opimizing an existing application for GPUs.
- ATLAS is far from being able to do this.
- I attended as a guest.
- Goals: Start learning about practicalities of GPU programming. Prototype sharing an xAOD-style object with a GPU.

GPU architecture



GPU programming

- Two APIs generally available:
  - CUDA: Nvidia proprietary, closed-source. Useful only with nvidia hardware. More mature.
  - OpenCL: Attempts to be a vendor- and device-neutral platform. Originated with Apple, now seems to be pushed mostly by AMD. Open-source implementations exist to run on CPU resources. Less mature.
- Both support writing GPU code in C (with some limitations) and C++ (with more limitations), but OpenCL does C++ only with the new 2.2 version.
  - Would have liked to use OpenCL for this exercise, but the nvidia drivers available on the BNL cluster didn't support C++ with OpenCL.
- Source:
  - CUDA: Host/device source mixed in single file, tagged with compiler directives. Process with CUDA compiler which produces object with embedded CUDA binary code.
  - OpenCL: Device code in separate source file. Compiled on-the-fly at runtime; or alternately, compile in advance to binary intermediate code.

High-level toolkits: SYCL

- Single-source; modern C++; standardized.
- Same source can be compiled for CPU and device.
- CPU-only open-source version; one commercial implementation.
- Looks interesting, but wasn't able to try it.

```
// wrap our data variable in a buffer
buffer<int, 1> resultBuf(data, range<1>(1024));

// create a command_group to issue commands to the queue
myQueue.submit([&](execution_handle<opencl22>& cgh) {
   // request access to the buffer
   auto res = resultBuf.get_access<access::mode::write>(cgh);

   // enqueue a prallel_for task
   cgh.parallel_for<class simple_test>(range<1>(1024),
      [=](id<1> idx) { res[idx[0]] = idx[0]; });
}); // end of our commands for
```

High-level toolkits: Kokkos

- Similar in intent to SYCL; didn't look at it very closely.

What is "Kokkos" ?

- **κόκκος (Greek)**
  - **Translation: "granule" or "grain" or "speck"**
  - **Like grains of salt or sand on a beach**
- **Programming Model Abstractions**
  - **Identify / encapsulate grains of data and parallelizable operations**
  - **Aggregate these grains with data structure and parallel patterns**
  - **Map aggregated grains onto memory and cores / threads**
- **An Implementation of the Kokkos Programming Model**
  - **Sandia National Laboratories' open source C++ library**

1

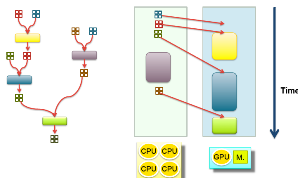
High-level toolkits: StarPU

- Framework for executing task graphs.
- Tasks can execute on one a number of devices, chosen dynamically by scheduler.
- Tasks declare inputs and outputs. Scheduler respects dependencies.
- Framework is responsible for copying data to/from devices as needed.

StarPU Execution Model: Task Scheduling

Mapping the graph of tasks (DAG) on the hardware

- Allocating computing resources
- Enforcing dependency constraints
- Handling data transfers



Inria Team Storm – Olivier Aumage – Introducing StarPU – 2. StarPU Programming/Execution Models 15

StarPU

- Developed by a small group from Bordeaux.
- Task graph orientation matches well what Athena is doing. ☺
- Package and interface are plain-C. User interfaces are untyped. Code quality not great. ☹
- Basic concepts:
  - Device: Coprocessing unit attached to the host.
  - Handle: Opaque reference to a data object managed by StarPU.
  - Task: Unit of processing. May run on either CPU or on a device, as chosen by the scheduler. Has lists of handles for input, output, and update dependencies.
  - Codelet: Piece of code executed by a task. May have different versions for CPU and each type of device.

Basic StarPU example

```
starpu_init (NULL);

// Data object.
static const int NX = 2048;
float v[NX] = {1, 0};
starpu_data_handle_t vector_handle;
starpu_vector_data_register (&vector_handle,
                             STARPU_MAIN_RAM,
                             reinterpret_cast<uintptr_t> (v),
                             NX, sizeof (v[0]));

starpu_codelet cl;               starpu_codelet_init (&cl);
cl.cpu_funcs[0] = cpu_func;      cl.cuda_funcs[0] = cuda_func;
cl.nbuffers = 1;                 cl.modes[0] = STARPU_RW;

float factor = 2.7;
starpu_task* task = starpu_task_create();   task->cl = &cl;
task->handles[0] = vector_handle;
task->cl_arg = &factor;  task->cl_arg_size = sizeof (factor);
```

Basic StarPU example, cont.

```
  starpu_task_submit (task);
  starpu_task_wait_for_all();
  starpu_data_unregister (vector_handle);
  starpu_shutdown();
```

CPU function:

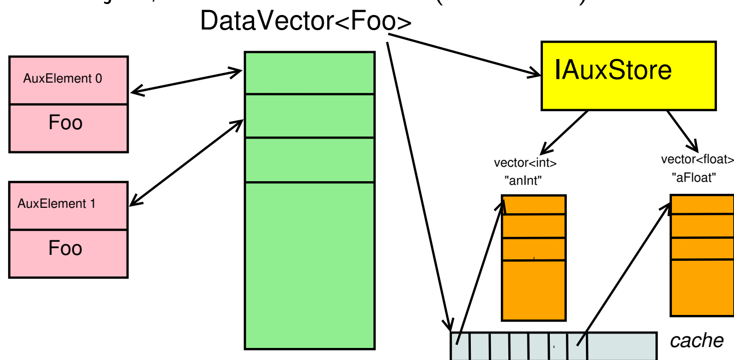
```
void cpu_func (void *buffers[], void* cl_arg)
{
  float factor = *reinterpret_cast<float*> (cl_arg);
  unsigned n = STARPU_VECTOR_GET_NX (buffers[0]);
  float* val = reinterpret_cast<float*>
    (STARPU_VECTOR_GET_PTR (buffers[0]));
  for (unsigned i=0; i < n; i++) {
    val[i] *= factor;
  }
}
```

Basic StarPU example, cont.: CUDA function

```
static __global__ void vector_mult_cuda
   (unsigned int n, float* val, float factor)
{
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n)
    val[i] *= factor;
}

void cuda_func (void* buffers[], void* args)
{
  float factor = *(float*)args;
  unsigned n = STARPU_VECTOR_GET_NX (buffers[0]);
  float* val = (float*) (STARPU_VECTOR_GET_PTR (buffers[0]));
  const unsigned int threads = 64;
  const unsigned int nblocks = (n + threads-1) / threads;
  vector_mult_cuda<<<nblocks, threads, 0,
                     starpu_cuda_get_local_stream()>>>
    (n, val, factor);
  cudaStreamSynchronize (starpu_cuda_get_local_stream());
}
```

xAOD data

Reminder: xAOD stores object data as individual vectors in a separate store object, via abstract interface (but cached).



- Accessing data via element pointers would spoil vectorization.
- DataVector itself is anyway too complicated for GPU code (virtual functions, exceptions, dynamic memory, STL).
- Restrict functionality for this excercise: No dynamic variables; only deal with entire objects; GPU can't change container size.

DataVectorAccessor

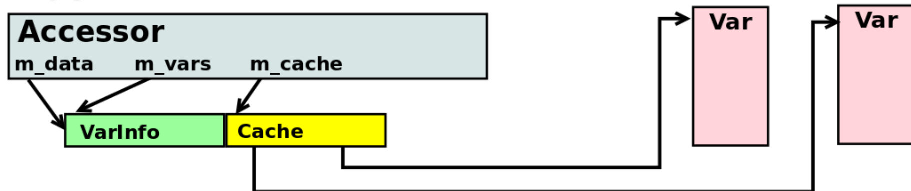
Define 'Accessor' object for use by GPU-style code.

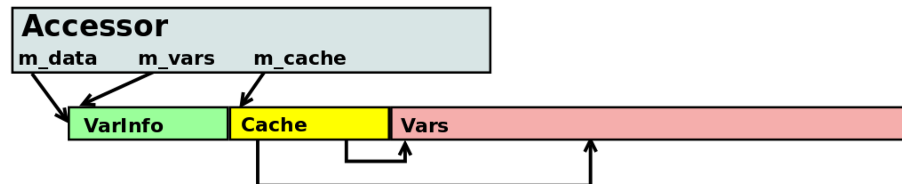
```
class DataVectorAccessor {  public:
 DataVectorAccessor (const SG::AuxVectorData& ccc,
                     size_t sz, size_t accSize);
 ...
protected:
  template <class T>
  ATH_DEVHOST
  T* arr (SG::auxid_t id) { return (T*)m_cache[id]; }
  ...
private:
  struct VarDesc
  {
    SG::auxid_t m_id;
    size_t m_eltsize;         size_t m_offs;
  };
  size_t m_size;              size_t m_nvar;
  VarDesc* m_vars;            void** m_cache;
  ...
```

Derived Accessor class

```
class CaloCellAccessor : public DataVectorAccessor {
public:
  // Host only
  // Inits base class; fills Id members.
  CaloCellAccessor (const xAOD::CaloCellContainer& ccc);

  ATH_DEVHOST float* eArray()
  { return arr<float>(m_eId); }

  ATH_DEVHOST float e (size_t ndx) const
  { return arr<float>(m_eId)[ndx]; }

  ...
private:
  SG::auxid_t m_eId;
  ...
```

Accessor layouts





StarPU Interface

- StarPU allows adding custom types.
- Also supports the notion of running a conversion between CPU and device representations.
  - Conversion can run on device.
  - Possibly may have been simpler, but not used here.
- Data object on a given node (CPU+devices) described by an *interface* structure.
  - Plain C: must be able to copy with memcpy(); no destructor.
  - One instance for each node managed by StarPU.
- Define operations on interface (via struct of function pointers):
  - Allocate on node.
  - Free on node.
  - Copy betewen nodes.
- A bit tricky to be able to get copying to work properly, given distinction between host/device memory. Eventually ended up maintaining serveral copies of an Accessor object from an interface.
  - Distinguishing between CPU pointers and device pointers is error-prone; could the C++ type system help? Probably would imply distinct types for CPU and device.

Final example

```
AthStarPU::Init init;
xAOD::CaloCellContainer ccc1, ccc2;
... Fill ccc1 with N elts; ccc2 with N empty elts ...
{
  CaloCellAccessor acc1 (ccc1);
  AthStarPU::Handle<CaloCellAccessor> handle1 (acc1);
  CaloCellAccessor acc2 (ccc2);
  AthStarPU::Handle<CaloCellAccessor> handle2 (acc2);
  MyTask mytask;
  mytask.submit (handle1, handle2);
  starpu_task_wait_for_all();
}
// Read results from ccc2.
```

Final example, cont.

```
class MyTask { public:
  MyTask() {
    starpu_codelet_init (&m_cl);
    m_cl.cpu_funcs[0] = cpu_func;   m_cl.cuda_funcs[0] = cuda_func;
    m_cl.nbuffers = 2;
    m_cl.modes[0] = STARPU_R;       m_cl.modes[1] = STARPU_RW;
  }
  void submit (const AthStarPU::Handle<CaloCellAccessor>& h1,
               AthStarPU::Handle<CaloCellAccessor>& h2) {
    starpu_task_submit (task (h1, h2));
  }
  starpu_task* task (const AthStarPU::Handle<CaloCellAccessor>& h1,
                     AthStarPU::Handle<CaloCellAccessor>& h2) {
    starpu_task* task = starpu_task_create();
    task->cl = &m_cl;
    task->handles[0] = h1;   task->handles[1] = h2;
    return task;
  }
  starpu_codelet m_cl;
```

Final example, cont.

```
void cpu_func (void *buffers[], void* /*cl_arg*/)
{
  const CaloCellAccessor& acc1 = dv_accessor<CaloCellAccessor>
                                   (buffers[0]);
        CaloCellAccessor& acc2 = dv_accessor<CaloCellAccessor>
                                   (buffers[1]);
  const size_t sz = acc1.size();
  STARPU_ASSERT_MSG (sz == acc2.size(), "Size mismatch.");
  for (unsigned int i = 0; i < sz; i++) {
    acc2.e(i)    = acc1.e(i) * 1.5;
    acc2.eta(i)  = acc1.eta(i);
    acc2.phi(i)  = acc1.phi(i);
    acc2.hash(i) = acc1.hash(i);
  }
}
```

Final example, cont.

```
void cuda_func (void* buffers[], void* /*args*/) {
  const CaloCellAccessor& acc1 =
          dv_accessor<CaloCellAccessor> (buffers[0]);
        CaloCellAccessor& acc2 =
          dv_accessor<CaloCellAccessor> (buffers[0]);
  const size_t sz = acc1.size();
  STARPU_ASSERT_MSG (sz == acc2.size(), "Size mismatch.");

  const unsigned int threads = 64;
  const unsigned int nblocks = (sz + threads-1) / threads;

  const CaloCellAccessor& devacc1 =
          dv_dev_accessor<CaloCellAccessor> (buffers[0]);
        CaloCellAccessor& devacc2 =
          dv_dev_accessor<CaloCellAccessor> (buffers[1]);
  cuda_copy<<<nblocks, threads, 0,
                          starpu_cuda_get_local_stream()>>>
    (sz, devacc1, devacc2);
  cudaStreamSynchronize (starpu_cuda_get_local_stream());
}
```

Final example, cont.

```
static __global__ void cuda_copy (unsigned int n,
       const CaloCellAccessor& acc1,  CaloCellAccessor& acc2)
{
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) {
    acc2.e(i) = acc1.e(i) * 1.5;
    acc2.eta(i) = acc1.eta(i);
    acc2.phi(i) = acc1.phi(i);
    acc2.hash(i) = acc1.hash(i);
  }
}
```

Final thoughts 1

- Useful/interesting exercise.
  - But only very exploratory prototyping — this code is not really useful.
  - Didn't have time to try any non-trivial processing on GPUs.
- StarPU is the closest fit so far to what we're doing in Athena.
  - But still pretty far from what we'd want.
  - At this point, thinking it's probably better to lift ideas/code from StarPU (and elsewhere) and incorporate into Athena than the other way around.
- xAOD-style data model looks usable from GPUs.
  - Still need to think about the best programming interface.
  - Can leverage the idea further. Example: a CaloCell container could have variables for eta, phi, etc. that are shared between events.

Final thoughts 2

- Some basic conceptual changes needed in Athena?
  - Multiple ways to produce an object in the scheduler.
    - ★ Need some sort of cost model?
    - ★ Device memory constraints on scheduling? Haven't seen this mentioned in StarPU literature.
  - Scheduler should be able to queue tasks in advance of when they're ready to run — avoid latency on device side.
    - ★ StarPU dedicates a CPU core to each device for scheduling — say that's needed to keep the device fed.
  - StoreGate manages multiple replicas of objects.
  - Copy operations declared to scheduler as algorithms?
- Will continue to play with this as time/resources allow.