EXECUTORS ON CUDA GRAPHS

Jared Hoberock, March 12, 2019
AGENDA

Quick Executors review
Goals for project
Kernel Executor
Graph Executor
Conclusions
Diverse Libraries

sort(...) \hspace{1cm} \text{for\_each(...)}

sgemm(...) \hspace{1cm} \text{train\_network(...)}

your\_favorite\_library\_function(...)

Multiplicative Explosion

Diverse Resources

Operating System Threads

Thread pool schedulers

OpenMP runtime

SIMD vector units

GPU runtime

Fibers

3
Diverse Libraries

- sort(…)
- for_each(…)
- sgemm(…)
- train_network(…)
- your_favorite_library_function(…)

Uniform Abstraction

Executors

Diverse Resources

- Operating System Threads
- Thread pool schedulers
- OpenMP runtime
- SIMD vector units
- GPU runtime
- Fibers
EXECUTORS
Like allocators for threads

Abstraction for creating threads*
Programmers need to control where applications execute
  • Locality is critical to performance
Programmers need a uniform interface
  • Dealing with multiple different execution APIs is complicated
  • A single API organizes things
GRAPHS EXPLORATION

Project goals

Explore how to target graph runtimes (esp. CUDA Graphs) from Executors API
Explore Executors usage within a large application
  ● QMCPACK: Open-source quantum chemistry simulator
Explore “Senders & Receivers”
  ● C++ proposal for lazy execution on Executors
  ● wg21.link/P1194

Non-goal: Did not want to focus on the design an explicit graph abstraction
PROTOTYPE

Two executors

kernel_executor
- Implemented with traditional CUDA kernel launches
- Eager

graph_executor
- Implemented with CUDA graphs
- Associated ensemble of “Senders”
- Lazy
// a cuda_context owns resources
cuda_context ctx;

// get a CUDA stream from somewhere
cudaStream_t stream = ...

// create a kernel_executor
kernel_executor ex(ctx, stream);

// launch a kernel
ex.bulk_execute(...);

// wait for all kernels to finish
ex.wait();
grid_index shape = ...

ex.bulk_execute([] __device__ (grid_index idx, ...)
{
    int block_idx = idx[0].x;
    int thread_idx = idx[1].x;

    printf(“Hello world from thread %d in block %d
”, thread_idx, block_idx);
},
shape,
...
);
grid_index shape = ...

ex.bulk_execute([] __device__ (grid_index idx, int& grid_shared, int& block_shared)
{
  int block_idx = idx[0].x;
  int thread_idx = idx[1].x;

  printf(“Hello world from thread %d in block %d\n”, thread_idx, block_idx);
},
shape,
[] __host__ __device__ { return 42; }, // single variable shared by all threads
[] __host__ __device__ { return 13; } // shared variable per block of threads
);
// get a CUDA stream from somewhere
cudaStream_t stream = ..

// create a graph_executor from the stream
graph_executor ex(stream);

// the root of the graph
void_sender root_node;

// make a kernel launch depend on the root
kernel_sender kernel = ex.bulk_then_execute(..., root_node);

// submit the kernel for execution
kernel.submit();

// wait for the kernel to finish
kernel.sync_wait();
Each method of `graph_executor` produces a different type of Sender targeting CUDA graphs

- `kernel_sender` ⇒ `cudaGraphAddKernelNode`
- `copy_sender` ⇒ `cudaGraphAddMemcpyNode`
- `host_sender` ⇒ `cudaGraphAddHostNode`

Senders represent nodes in a lazy task graph

- Mediate dependencies
- “Sends” its result down to its children

Senders are lazy

- Task description is separate from task submission
LAZY EXECUTION
Separating work description from submission

Proceeds in two* stages
Description: executor.bulk_then_execute() et al.
- Interacts with Senders to lazily describe work
Submission: sender.submit()
- Traverses sender DAG and communicates work to CUDA Graphs API
- Instantiates graph
- Launches graph
class graph_executor {
    private:
        cudaStream_t stream() const;
    ...

gpublic:
    template<class Function, class Sender>
    host_sender host_then_execute(Function f, Sender& predecessor) const {
        auto node_parameters_function = [=]()
        {
            // package f into parameters for the host node
            cudaHostNodeParams result = ...

            return result;
        };

        return host_sender{stream(), node_parameters_function, std::move(predecessor)};
    }
    ...
};
host_sender

Example sender implementation

class host_sender {
  private:
      std::function<cudaHostNodeParams()> node_params_function_;  
      any_sender predecessor_;  

      ...

  protected:
      cudaGraphNode_t insert(cudaGraph_t g) const {
          // insert the predecessor
          cudaGraphNode_t predecessor_node = predecessor_.insert(g);

          // generate the node parameters
          cudaHostNodeParams node_params = node_params_function_();

          // introduce a new host node
          cudaGraphNode_t result_node;
          cudaGraphAddHostNode(&result_node, g, &predecessor_node, 1, &node_params);

          return result_node;
      }
};
OVERHEAD

DAXPY Bandwidth versus Implementation on Titan V

- CUDA Kernel
- CUDA Graph
- Kernel Executor
- Graph Executor

Bandwidth (GB/s, higher is better)

Array Size
CONCLUSIONS

Enhancement opportunities

CUDA Graphs
- No memory management
- No deferred parameters
- Leads to out-of-band communication

Senders & Receivers
- Two-stage is awkward for systems like CUDA Graphs
- No support for replay
- Not clear how Receivers would leverage systems like CUDA Graphs