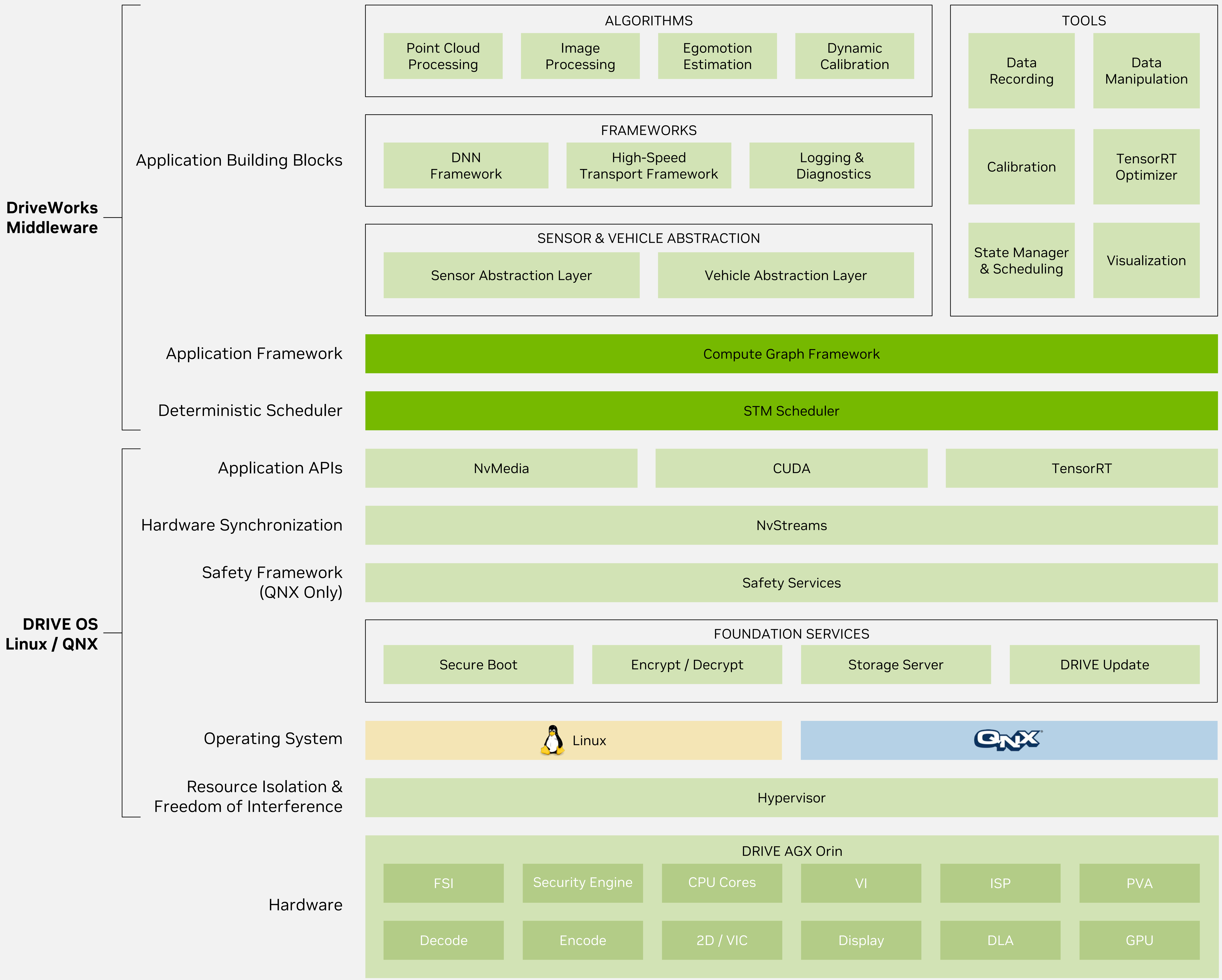
Performance Oriented Scheduling with System Task Manager

DRIVE SDK Supercharges AV Development

One architecture
DRIVE OS & DriveWorks
DRIVE AGX Orin



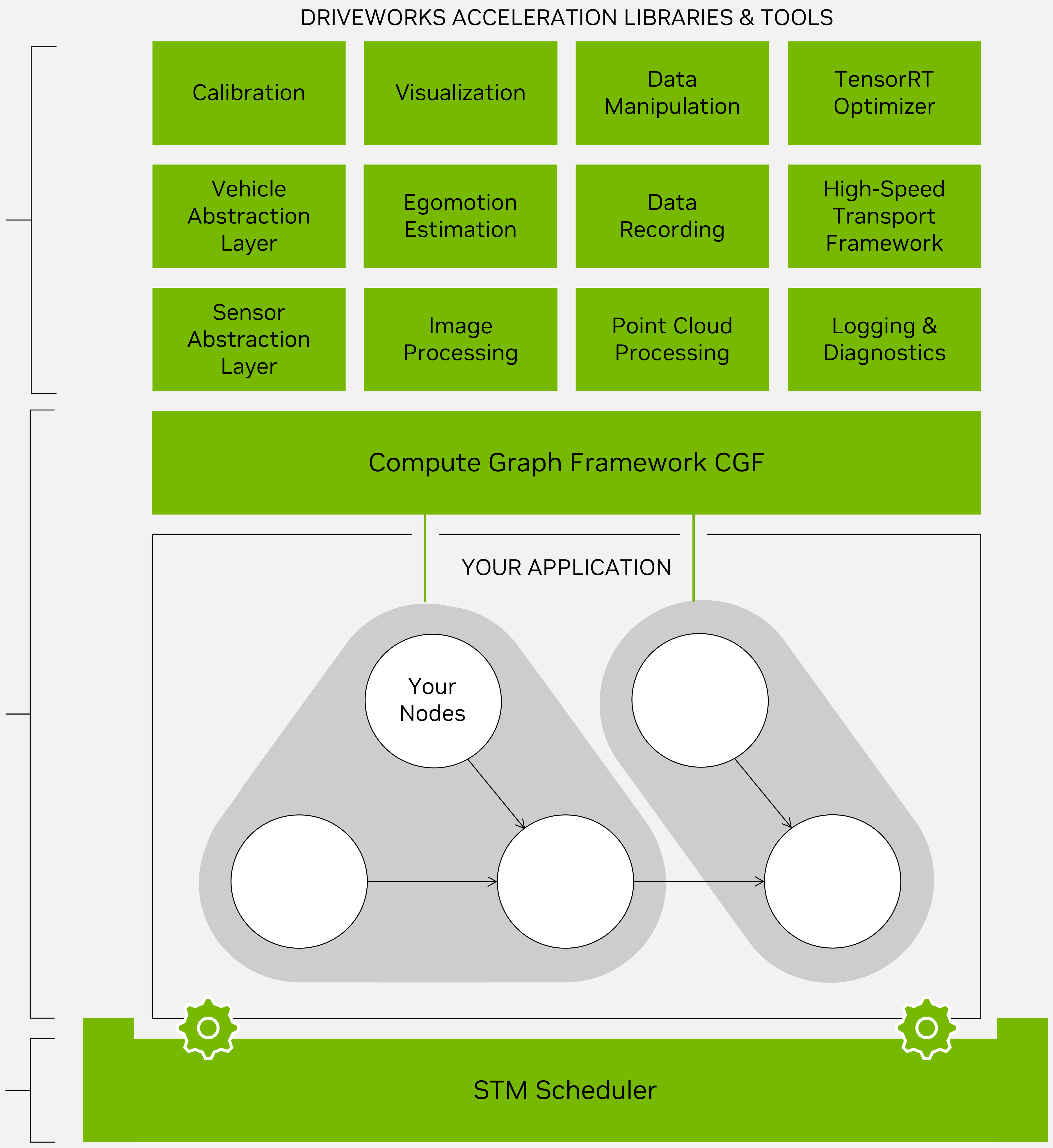
NVIDIA

DriveWorks — Comprehensive Middleware Solution

Rich Library of Algorithms and Tools
to accelerate your applications

Compute Graph Framework
to simplify complex computational pipelines

System Task Manager
to deterministically schedule application tasks



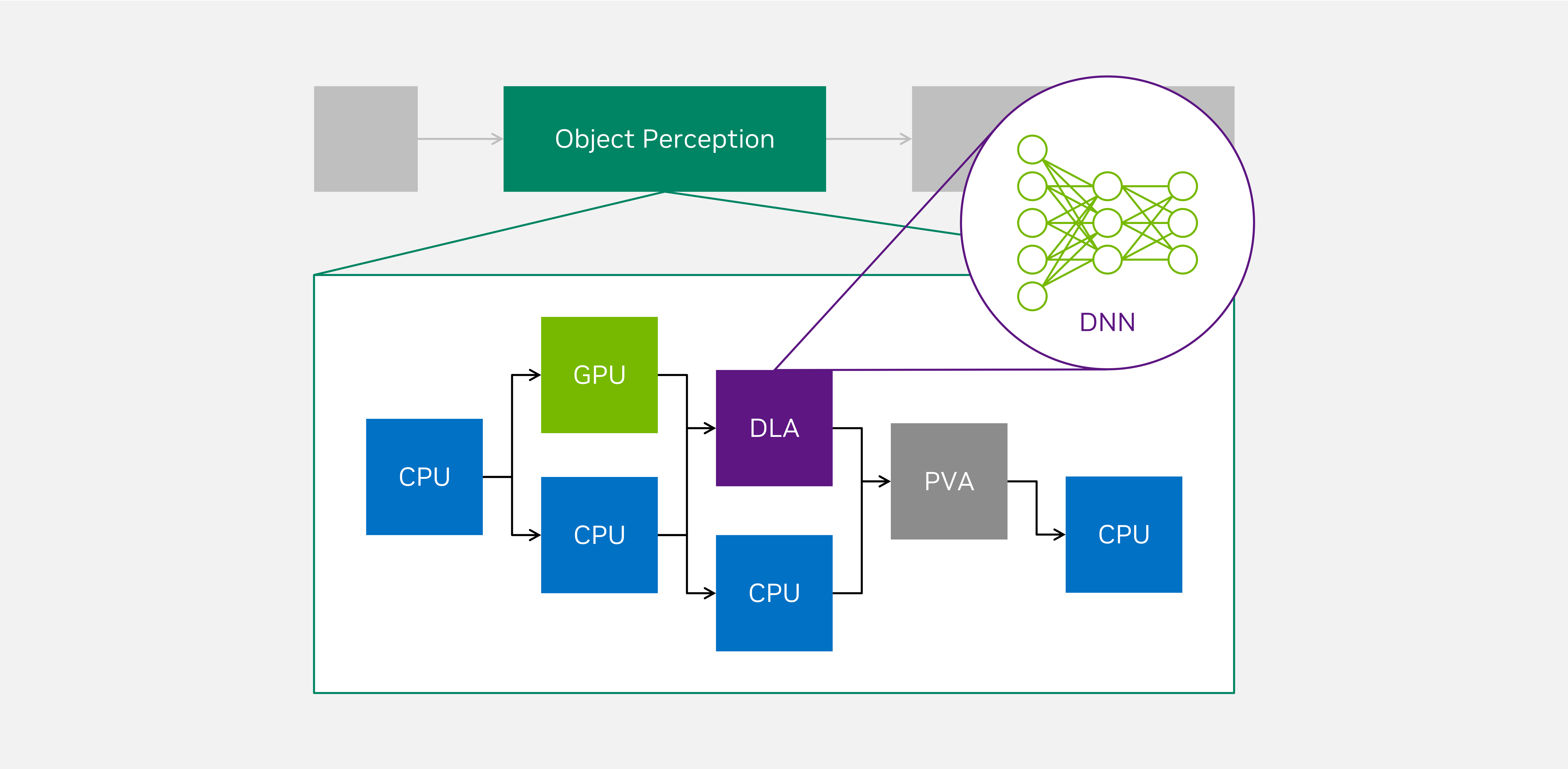
AV Workload

Complex (heterogenous) compute graph

*cartoon pipeline only for illustrative purposes

AV Workload

Each of these again a compute graph



AV Workload

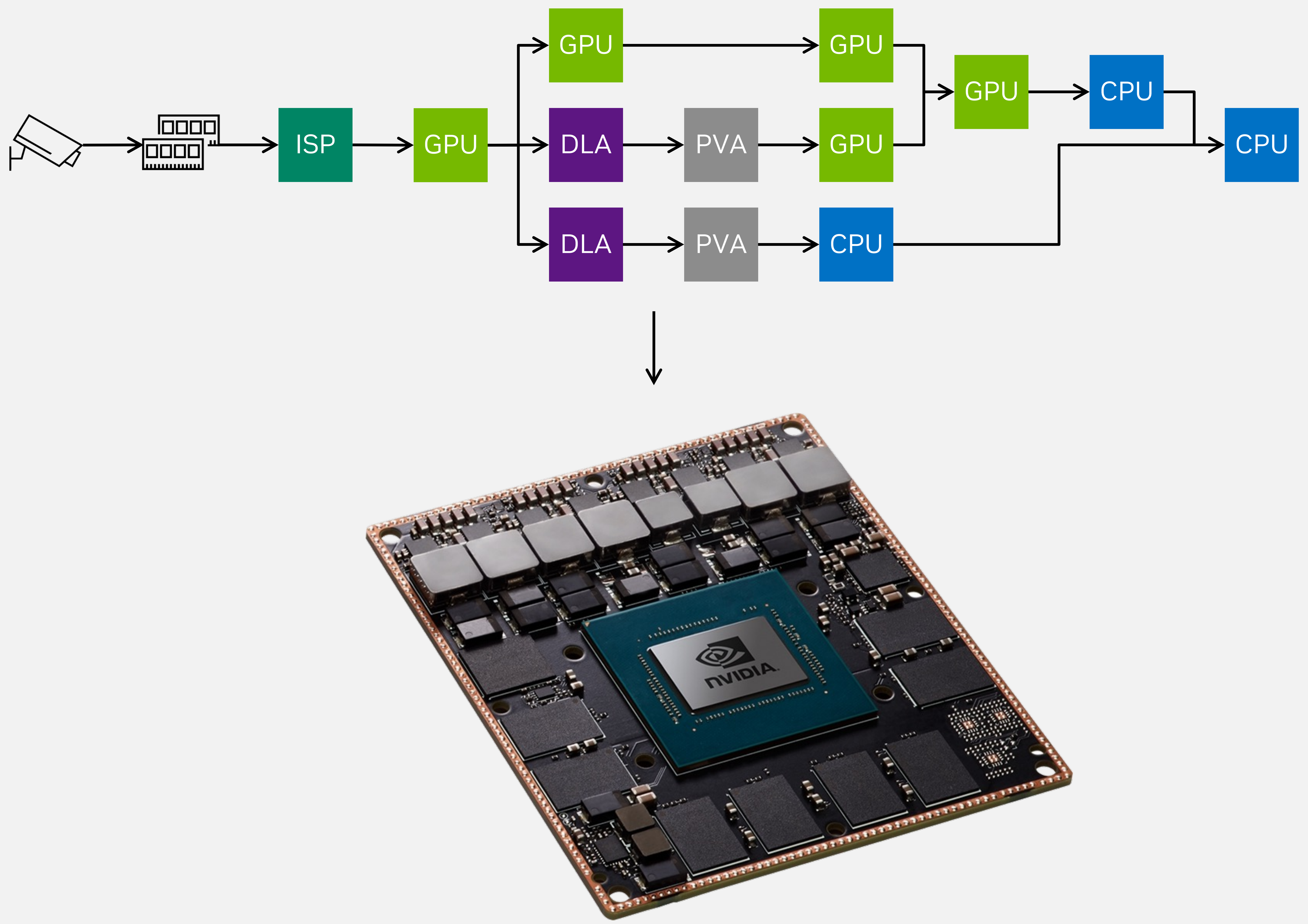
In real-time...

*latency bound only for illustrative purposes

Scheduling Problem

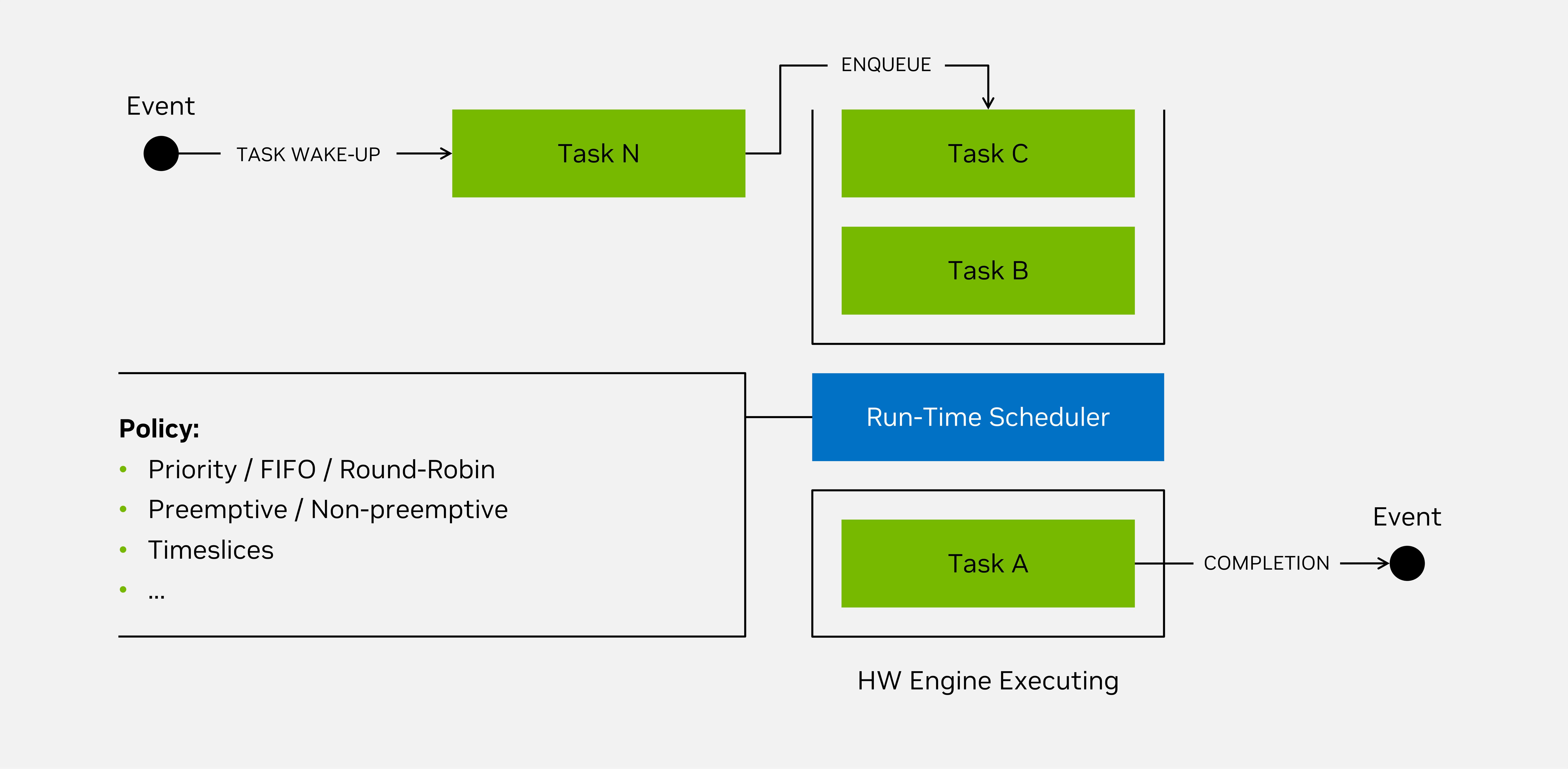
How to map…

- Constraints:
  - Latency / Worst-Case Response Time
  - Throughput / TOPS
  - Determinism
- Safety Considerations:
  - Program Flow-Monitoring
  - System Verification
  - ISO26262: “Manage / Restrict” use of interrupts, shared resources, concurrency, …



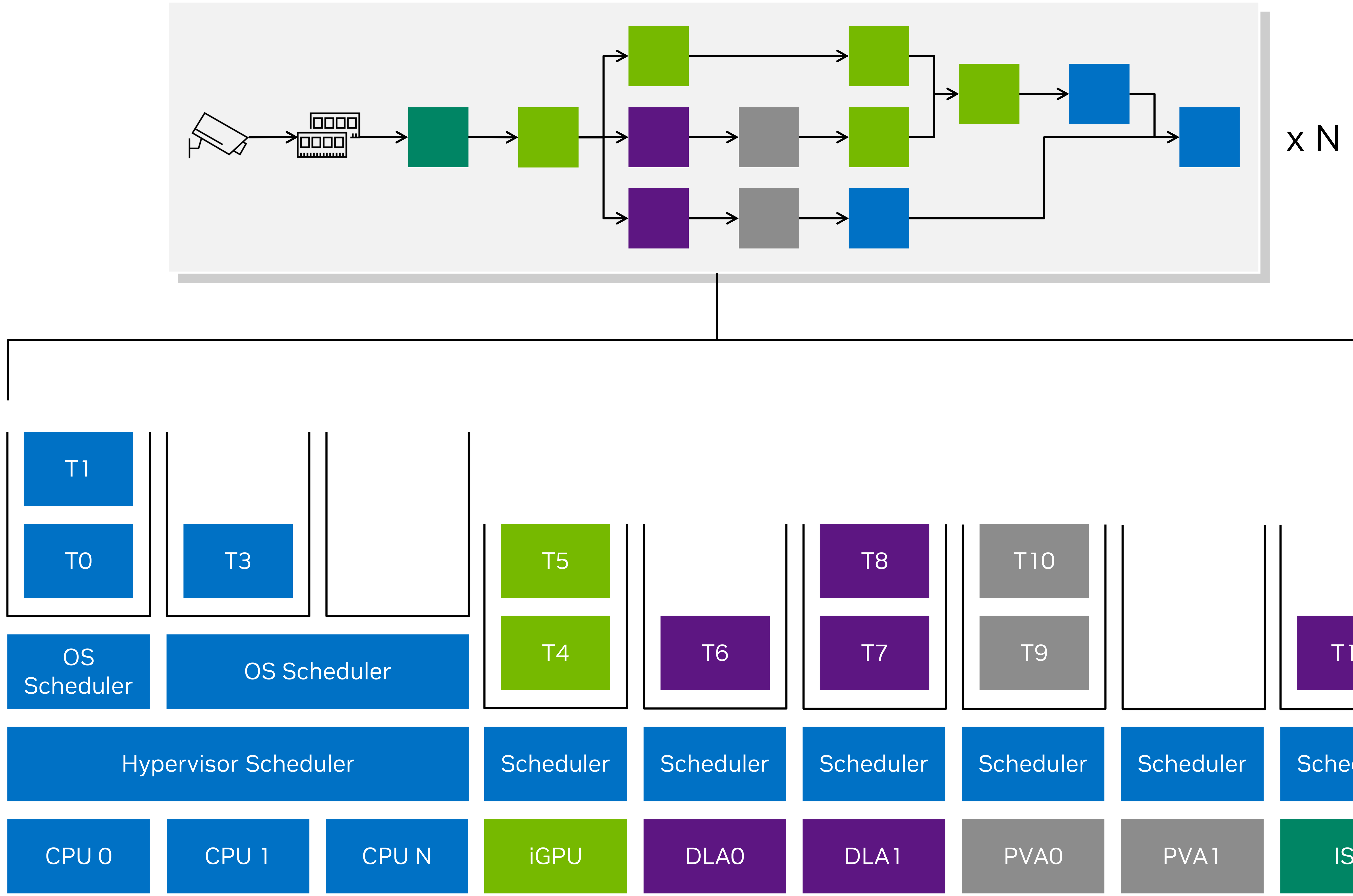
Traditional Scheduling View

Event-driven system



System-Level Scheduling Problem

How to guarantee determinism across many schedulers?



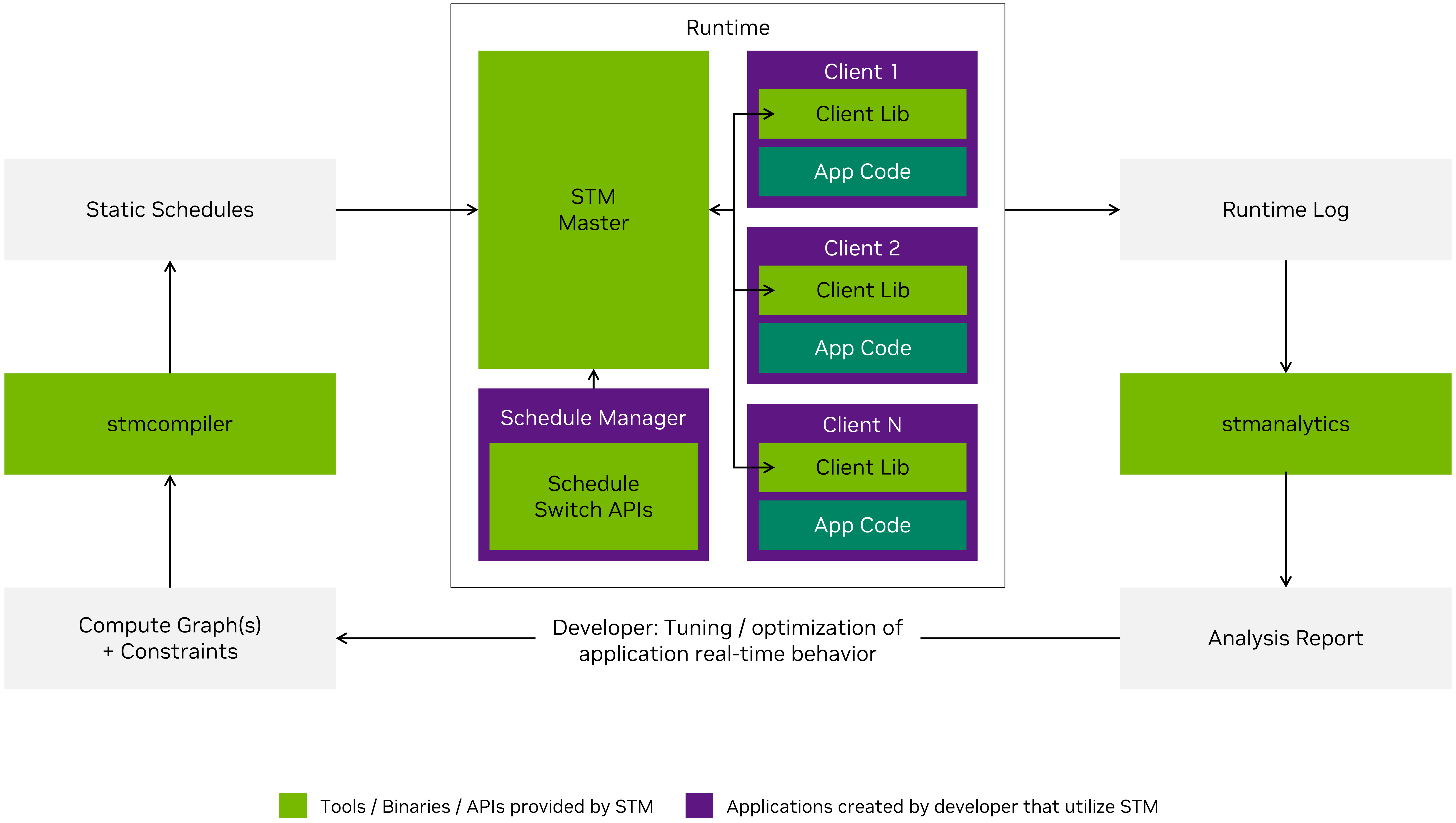
Introducing NVIDIA System Task Manager

Solution as part of NVIDIA DriveWorks

- User-Space
- Non-Preemptive
- Static Ordering
- Deterministic Performance
- Multi-Process
- Supports Heterogeneous Platforms

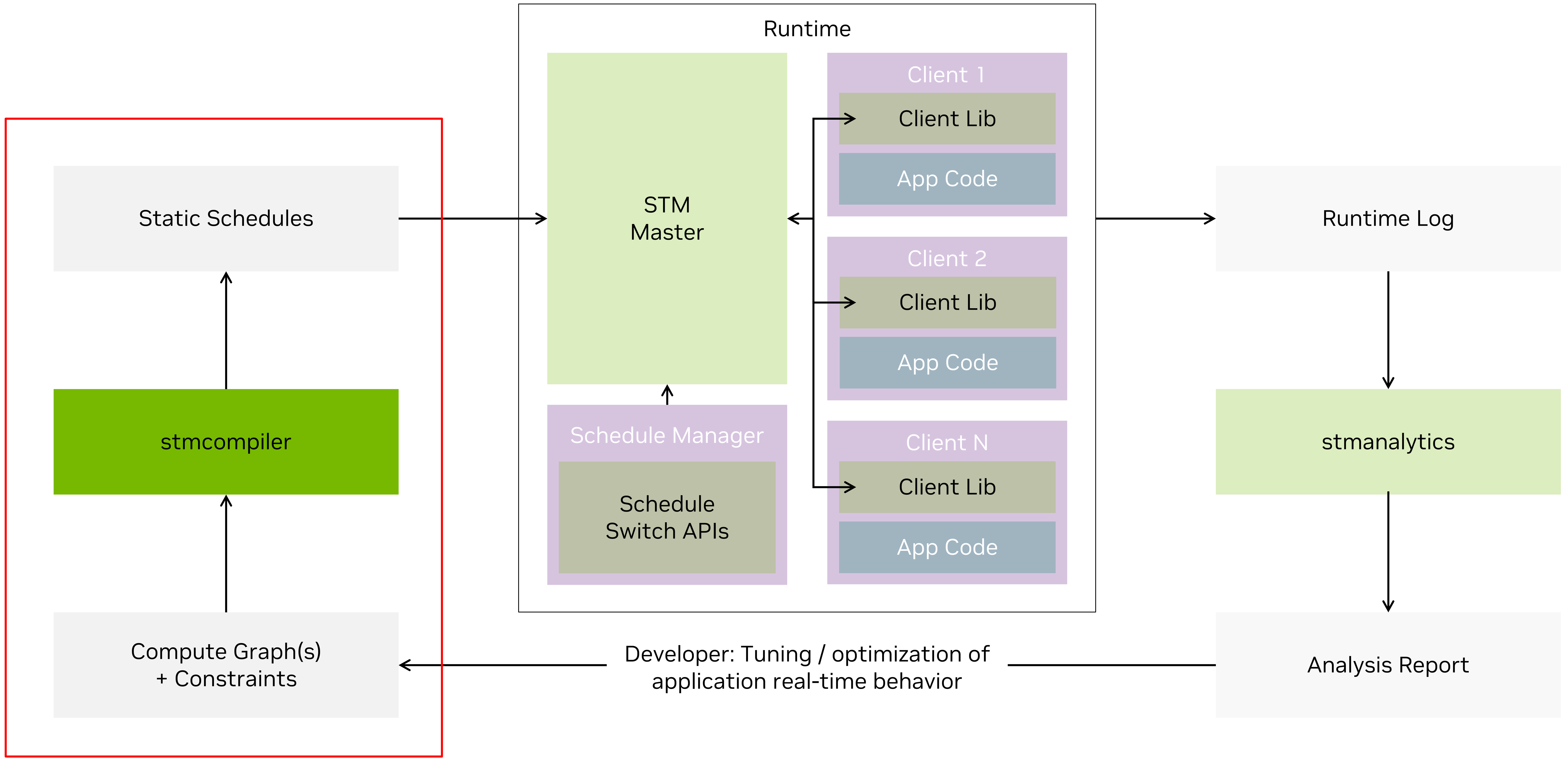
NVIDIA System Task Manager

Framework



Generating Static Schedules

STM Compiler

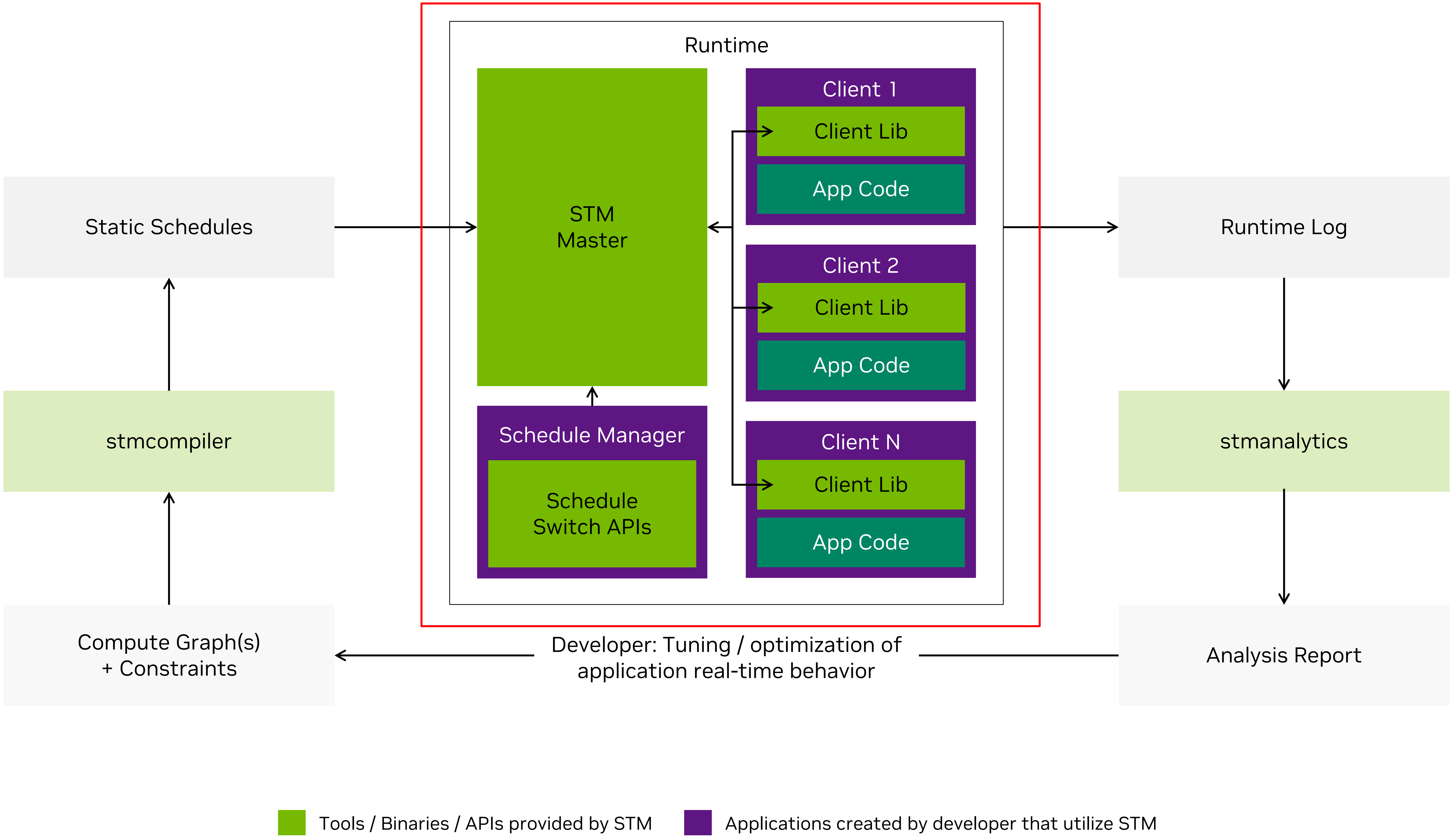


Tools / Binaries / APIs provided by STM

Applications created by developer that utilize STM

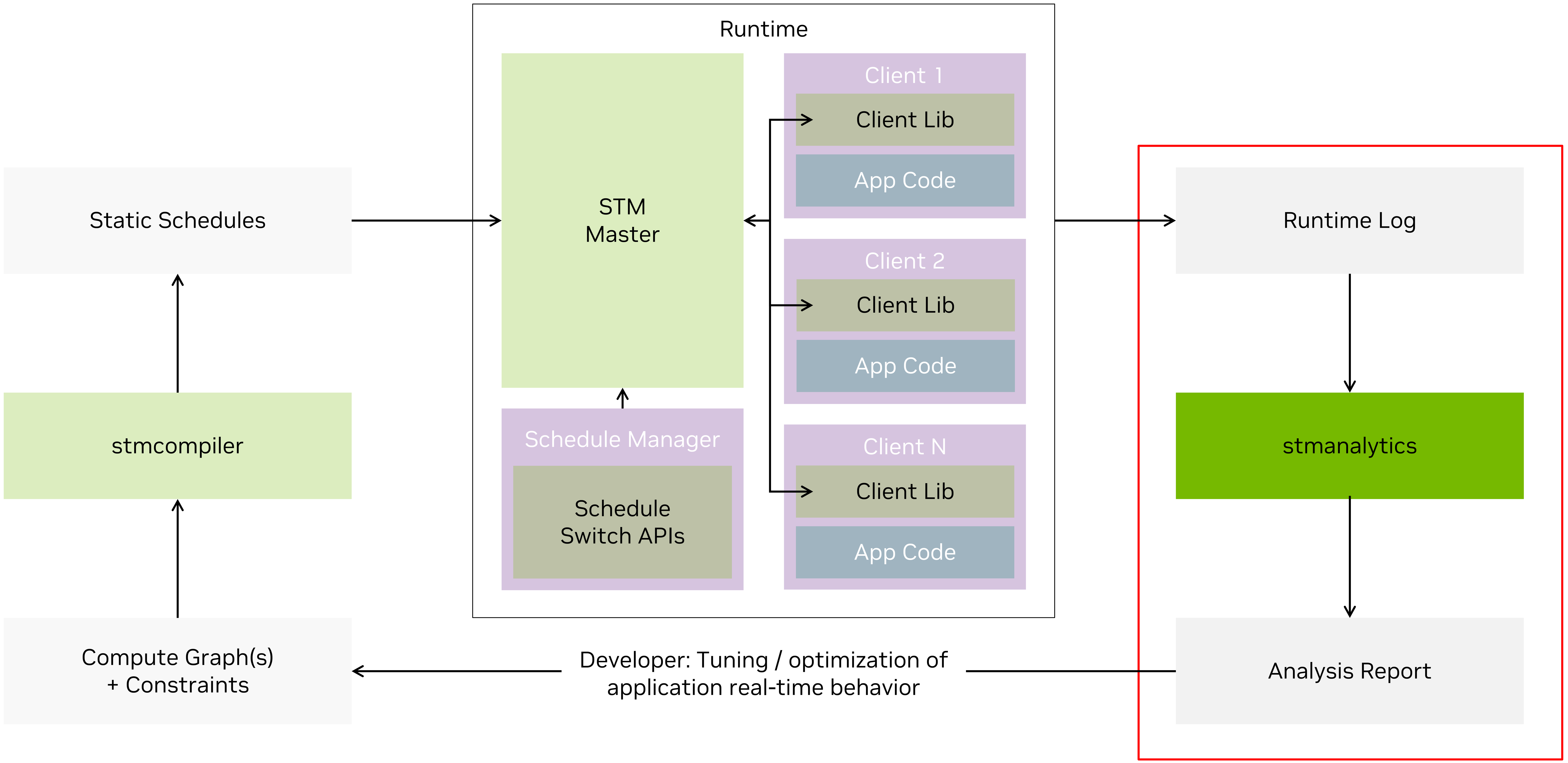
Executing Static Schedules

STM Runtime



Analyzing Execution Metrics

STM Analytics

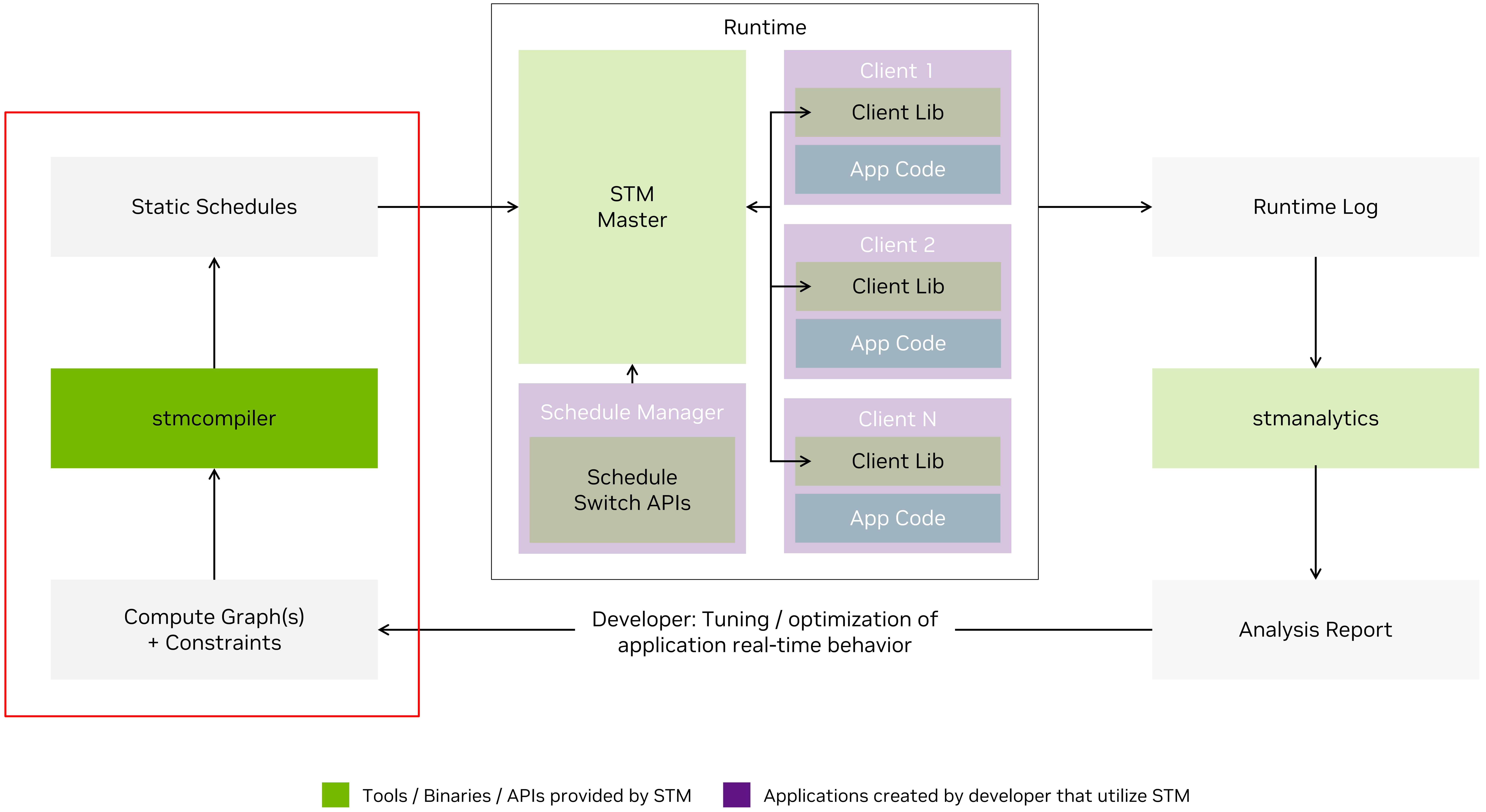


Tools / Binaries / APIs provided by STM

Applications created by developer that utilize STM

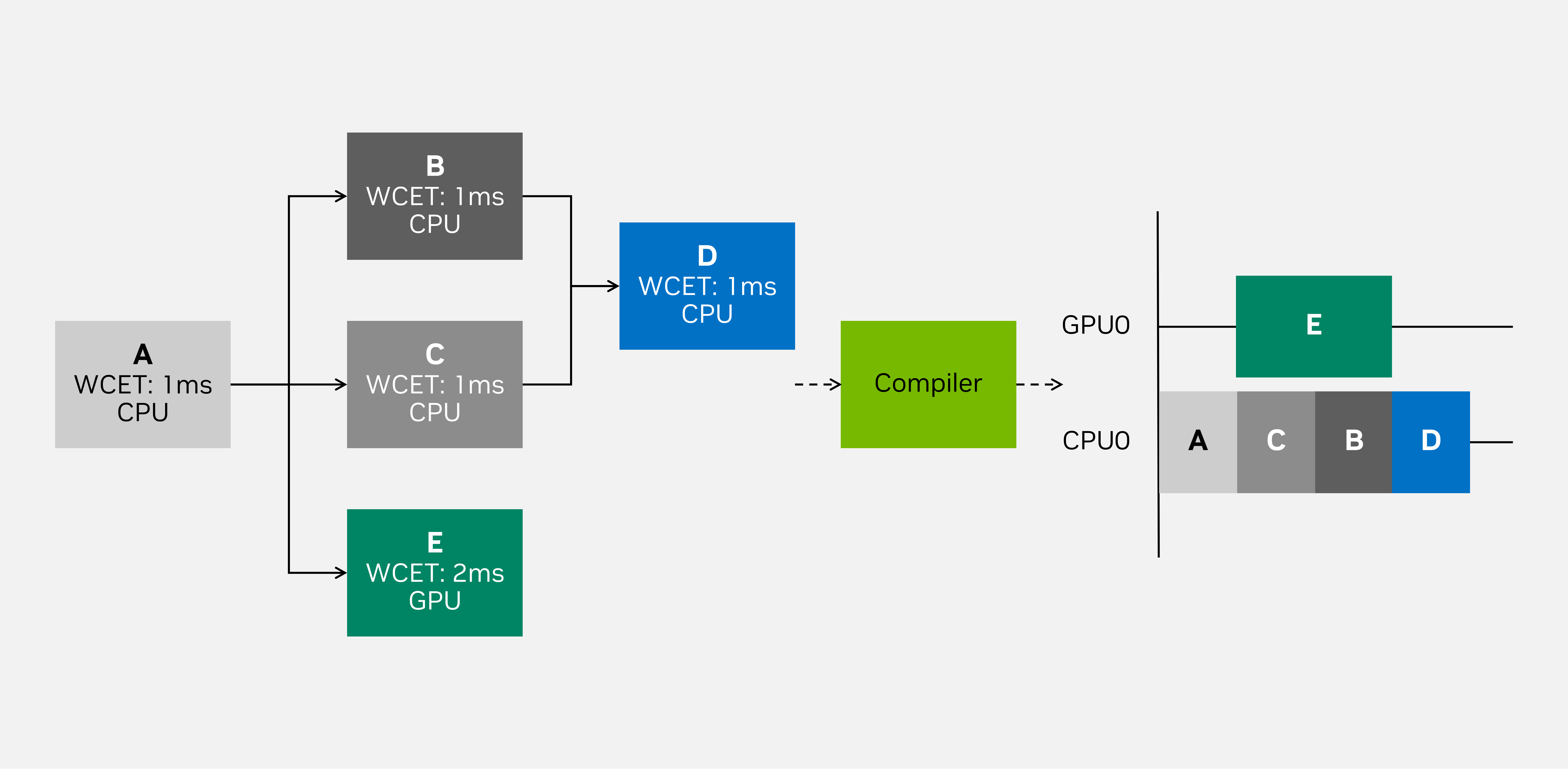
Generating Static Schedules

STM Compiler



Schedule Compilation

Converting requirements into static, recurring schedule

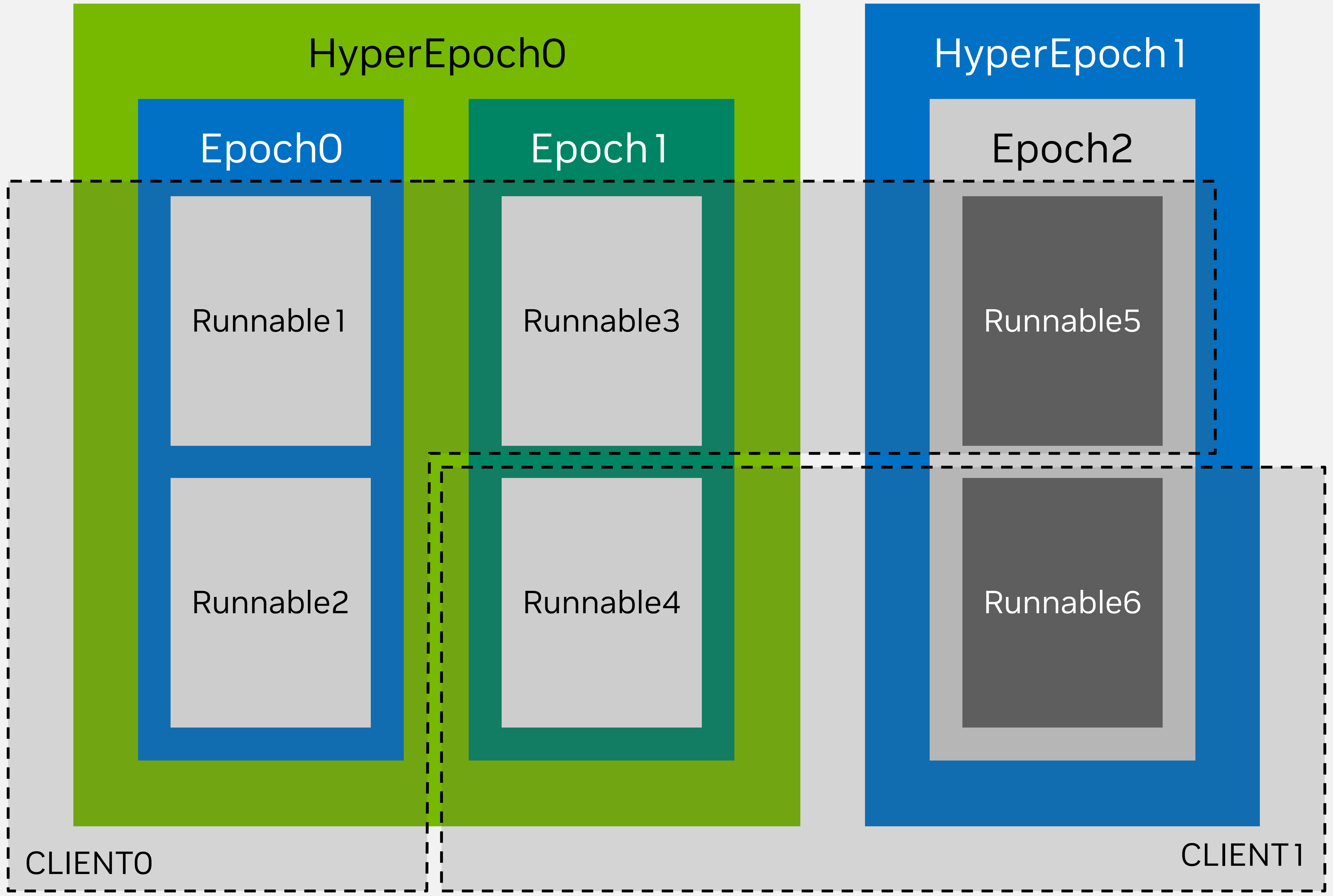


**WCET: Worst Case Execution Time*

Scheduling Primitives

- Runnable
- HyperEpoch
- Epoch
- Client
- WCET
- Dependency
- Resource

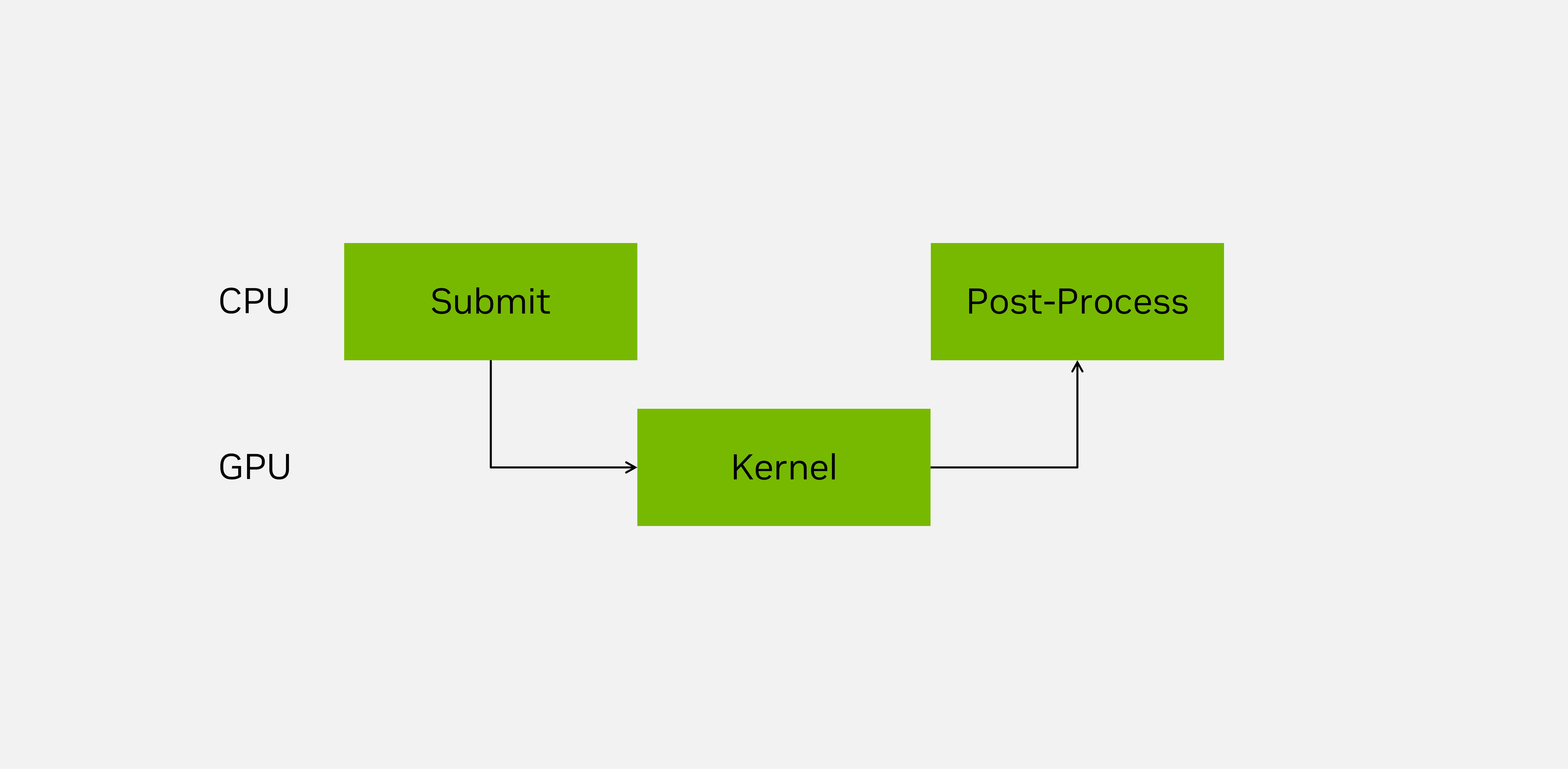
- Refer Documentation for details



Boundaries in NVIDIA DRIVE Scheduler

Tutorial

Simple GPU application with pre and post processing on CPU



Questions

- How many runnables?
- How many hyperepochs?
- How many epochs?
- How many clients?
- What are the wcets for each runnable?
- What are the dependencies between the runnables?

Graph Specifications, Global Resources and HyperEpochs

cpu_gpu_single_process.yml

```yaml
---
Version: 3.0.0           # Version number of this input yaml format
cpu_gpu_single_process: # Name of the DAG
  Identifier: 101       # Unique identifier of this schedule. Used when there are multiple schedules
  Resources:            # Global resources available in the system
    CPU: [CPU0]         # We define single CPU
    GPU: [iGPU]         # We define single GPU
  Hyperepochs:          # Hyperepoch definitions
  - hyperepoch0:        # We define single hyperepoch
      Period: 100ms     # The period for this hyperepoch is 100ms
      Epochs:           # Epoch definitions
      - epoch0:         # We define single epoch
```

Clients — Runnables

cpu_gpu_single_process.yml

```yaml
Clients:                                     # Client definitions
- GpuX:                                      # We define single client "GpuX"
    Resources:                               # Resource definitions for this client
      CUDA_STREAM:                           # Cuda Streams used by "GpuX"
      - CUDA_STREAMX: iGPU                   # We define single cuda stream and map it to "iGPU"
    Epochs:                                  # Epochs definitions for "GpuX"
    - hyperepoch0.epoch0:                    # Epoch 0 definition for "GpuX"
        Runnables:                           # Runnables present in "GpuX"
        - submit:                            # We define a pre-process runnable "submit"
            WCET: 10ms                       # We define a wcet of 10 ms for "submit"
            Dependencies: []                 # "submit" does not depend on any other runnable
            Submits: GpuX.kernel             # "submit" submits another runnable "kernel"
            Resources: [CPU0, CUDA_STREAM]   # "submit" uses resources "CPU0" and "CUDA_STREAM"
        - kernel:                            # We define the GPU runnable "kernel"
            WCET: 10ms                       # We define a wcet of 10 ms for "kernel"
            Dependencies: []                 # "kernel" does not depend on any other runnable
                                             # "kernel" implicitly depends on "submit"
            Resources: [iGPU]                # "kernel" requires "iGPU" to run
        - post_process:                      # we define another runnable "post_process"
            WCET: 10ms                       # we define a wcet of 10 ms for "post_process"
            Dependencies: [GpuX.kernel]      # "post_process" depends on "GpuX.kernel"
            Resources: [CPU0]                # "post_process" requires "CPU0" to run
```

STM Compiler Usage

stmcompiler.py -i cpu_gpu_single_process.yml

Produces compiled schedule in cpu_gpu_single_process.stm

[mosh] alb_d@X299-A: ~

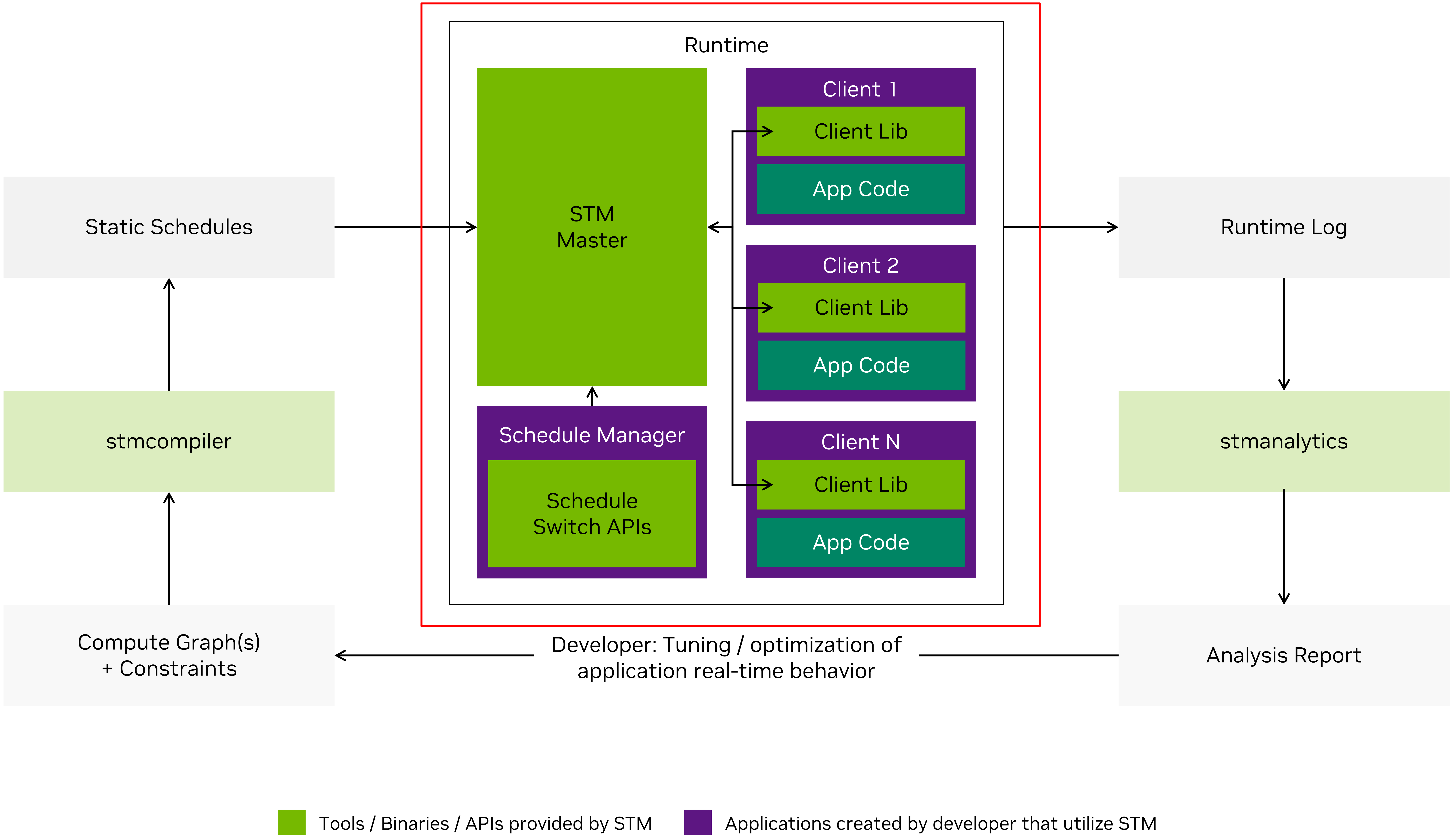
```
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
alb_d@X299-A:$
[webinar] 0:bash*                                    "X299-A" 14:23 24-Jul-23
```


cpu_gpu_single_process.stm
Highlight Critical Path
default.iGPU
default.CPU0
GpuX.default.CUDA_STREAMX
Resource
0
5
10
15
20
25
30
Time (ms)

NVIDIA.

Executing Static Schedules

STM Runtime



Runtime APIs

Init and deinit

```
void stmClientInit(const char* clientName);
```

- First call
- Initializes STM state
- Blocks for communication with stm_master

```
void stmClientExit(void);
```

- Cleans up STM state
- Last call

```
int main(int argc, const char** argv)
{
    (void)argc;
    (void)argv;

    cudaError_t cuddaErr = cudaStreamCreateWithFlags(&m_stream,
    cudaStreamNonBlocking); assert(cudaErr == cudaSuccess);

    // Init
    stmClientInit("GpuX");

    // Register runnables
    stmRegisterCudaSubmitter(submit, "submit", NULL);
    stmRegisterCpuRunnable(post_process, "post_process", NULL);

    // Register resources
    stmRegisterCudaResource("CUDA_STREAMX", m_stream);

    // Execute
    stmErrorCode_t stmErr = stmEnterScheduler();
    assert(stmErr == STM_SUCCESS);

    // Cleanup
    stmClientExit();
}
```

Runtime APIs

Registration

```
stmErrorCode_t stmRegisterCpuRunnable (stmRunnable_t func, const char* const runnableId, void* userdata);
```

```
stmErrorCode_t
stmRegisterCudaSubmitter(stmCudaSubmitter_t func, const char* const runnableId, void* userdata);
```

- Register function pointers and map them to string names
- Last param is arg to pass while invoking the function pointers
- Names must be same as in the yaml

```
stmErrorCode_t stmRegisterCudaResource(const char* resourceName, cudaStream_t stream);
```

- Similarly, map created resources to string names
- Names must be same as in the yaml

+ Similar for many other engines!

```
int main(int argc, const char** argv)
{
    (void)argc;
    (void)argv;

    cudaError_t cuddaErr = cudaStreamCreateWithFlags(&m_stream,
    cudaStreamNonBlocking); assert(cudaErr == cudaSuccess);

    // Init
    stmClientInit("GpuX");

    // Register runnables
    stmRegisterCudaSubmitter(submit, "submit", NULL);
    stmRegisterCpuRunnable(post_process, "post_process", NULL);

    // Register resources
    stmRegisterCudaResource("CUDA_STREAMX", m_stream);

    // Execute
    stmErrorCode_t stmErr = stmEnterScheduler();
    assert(stmErr == STM_SUCCESS);

    // Cleanup
    stmClientExit();
}
```

Runtime APIs

Execution

`stmErrorCode_t stmEnterScheduler(void);`

- Main function within which execution happens
- Blocking call
- To exit, either send SIGTERM to stm_master, or an asynchronous call to stmExitScheduler() from within the same process, or specify max number of hyperepoch frames as param to stm_master

```
int main(int argc, const char** argv)
{
    (void)argc;
    (void)argv;

    cudaError_t cuddaErr = cudaStreamCreateWithFlags(&m_stream,
    cudaStreamNonBlocking); assert(cudaErr == cudaSuccess);

    // Init
    stmClientInit("GpuX");

    // Register runnables
    stmRegisterCudaSubmitter(submit, "submit", NULL);
    stmRegisterCpuRunnable(post_process, "post_process", NULL);

    // Register resources
    stmRegisterCudaResource("CUDA_STREAMX", m_stream);

    // Execute
    stmErrorCode_t stmErr = stmEnterScheduler();
    assert(stmErr == STM_SUCCESS);

    // Cleanup
    stmClientExit();
}
```

Runtime Execution

Execution of the application

stm_master -s cpu_gpu_simple.stm -l logfile.stmdat -e 10

Mandatory Arguments: schedule file generated by stmcompiler

Optional Argument: logfile to generate for post run analytics

Optional Argument: terminate after 10 hyperepoch frames

stm_client:
No arguments since this is user created and we created one without any arguments required

File Edit Selection View Go Run Terminal Help

main.cpp - runtime-demo - Visual Studio Code

EXPLORER

OPEN EDITORS
- main.cpp cpu_gpu_demo\client
- gpu.cu cpu_gpu_demo\client

RUNTIME-DEMO
- cmake
- cpu_gpu_demo
  - client
    - CMakeLists.txt
    - gpu.cu
    - gpu.hpp
    - main.cpp
  - CMakeLists.txt
  - cpu_gpu_simple_pfm.json
  - cpu_gpu_simple.json
  - cpu_gpu_simple.py
  - gpu_multistream_multiprocess.yml
- CMakeLists.txt
- schedule.stm

OUTLINE

TIMELINE

main.cpp | gpu.cu

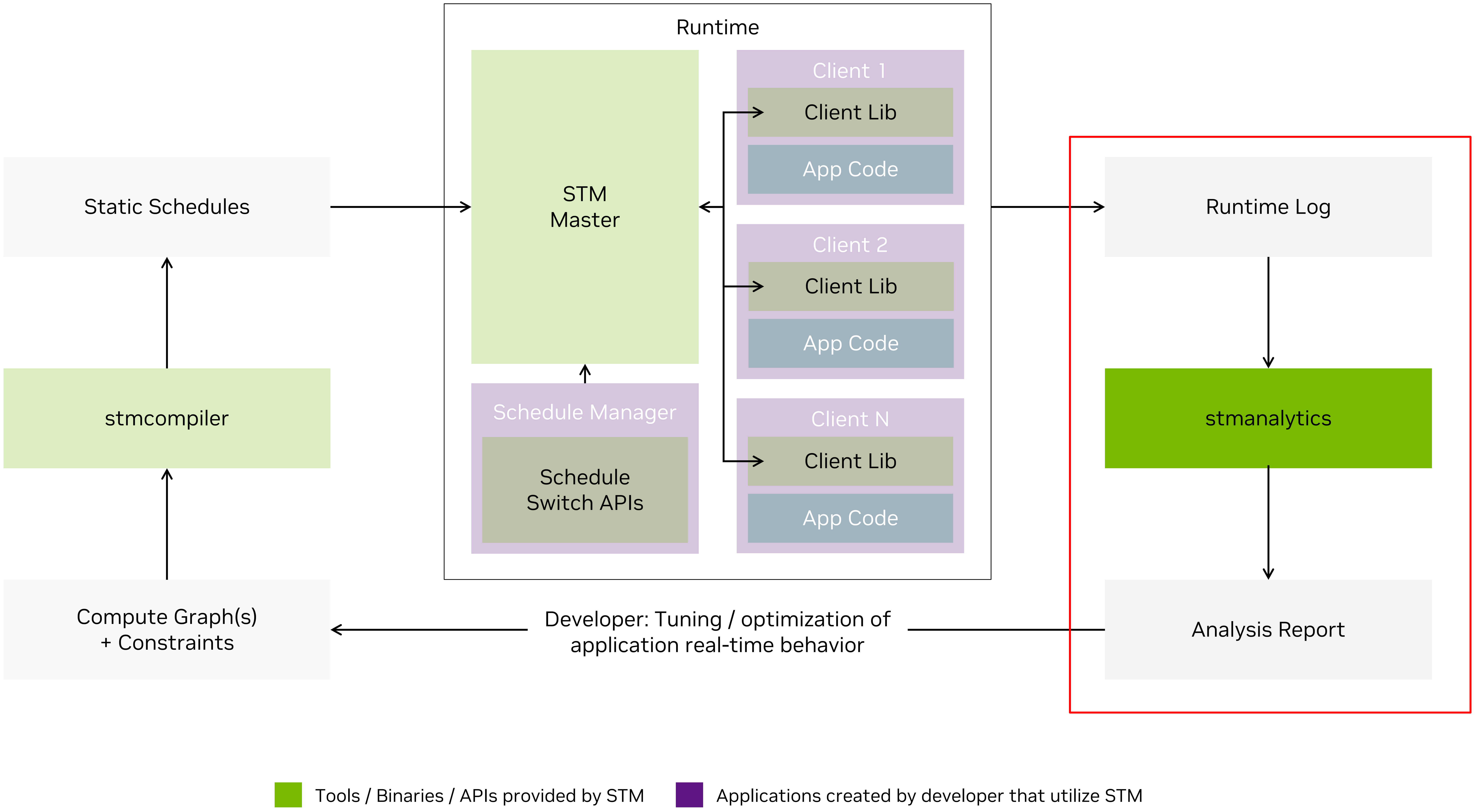
cpu_gpu_demo > client > main.cpp > main(int, const char **)

```cpp
}

int main(int argc, const char** argv)
{
    (void)argc;
    (void)argv;

    cudaError_t cudaErr = cudaStreamCreateWithFlags(&m_stream, cudaStreamNonBlocking);
    assert(cudaErr == cudaSuccess);

    // Init
    stmClientInit("GpuX");

    // Register runnables
    stmRegisterCudaSubmitter(submit, "submit", NULL);
    stmRegisterCpuRunnable(post_process, "post_process", NULL);

    // Register resources
    stmRegisterCudaResource("CUDA_STREAMX", m_stream);

    // Execute
    stmErrorCode_t stmErr = stmEnterScheduler();
    assert(stmErr == STM_SUCCESS);

    // Cleanup
    stmClientExit();
}
```

0 0 8 CMake: [Debug]: Ready No Kit Selected Build [all] Run CTest Ln 69, Col 16 Spaces: 4 UTF-8 LF {} C++ Win32

Analyzing Execution Metrics

STM Analytics



Analytics

- Post run info on how long each runnable and the entire frame took
- Helpful to get insight on what engines are being over / under budgeted and direct dev work to those
- Also provides info on overhead due to scheduling via STM when possible
- Necessary for getting accurate WCETs to provide as input to stmcompiler

Analytics

```
stmanalytics -s cpu_gpu_simple.stm -l logifle.stmdat -f html
```

Mandatory Arguments: schedule file and log file

Optional Argument: Output format

x1@NV-3Q5KNN3: ~

x1@NV-3Q5KNN3:~$

Analytics Example

Summary | Traces | KPIs | FPS

Epoch Info

Name	Scheduled Period	Mean Duration	# Executed Frames	# Skipped Frames	# Missing Frames	# Overruns
hyperepoch0.epoch0[0]	100000.00 us	244.73 us	10	0	0	0

Client: GpuX

Runnables

Name	SchedTime (us)	ExecTime (us)					#Runs	#Overruns
		Min	Avg	Max	99P	Sdev		
GpuX.submit	10000.00 us	37.37 us	39.13 us	43.26 us	43.15 us	1.88 us	10	0
GpuX.kernel	1000.00 us	2.91 us	3.06 us	4.03 us	3.94 us	0.32 us	10	0
GpuX.post_process	1000.00 us	6.87 us	7.46 us	7.81 us	7.81 us	0.29 us	10	0

Runnables With Overhead

Name	SchedTime (us)	ExecTime (us)					#Runs	#Overruns
		Min	Avg	Max	99P	Sdev		
GpuX.submit	10000.00	126.11 us	135.31 us	184.55 us	180.48 us	16.84	10	

Summary Information

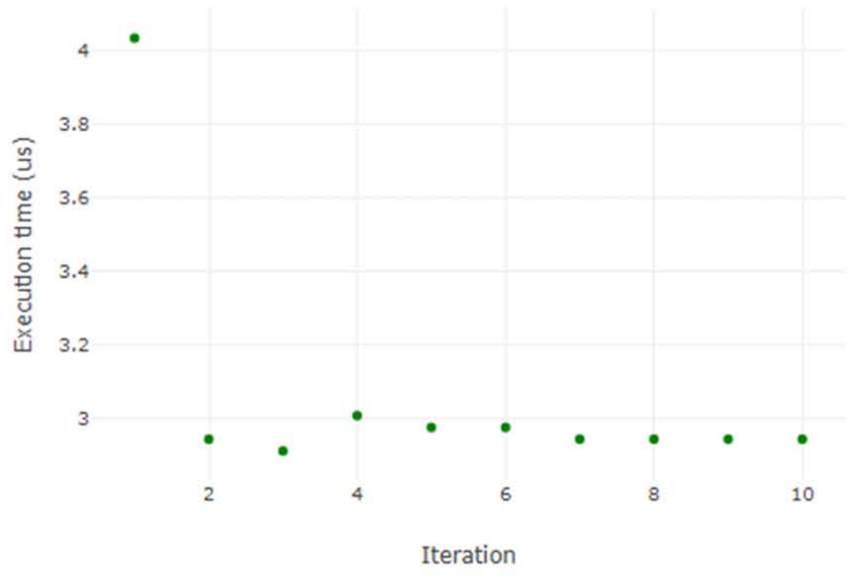
GpuX.kernel

CSV

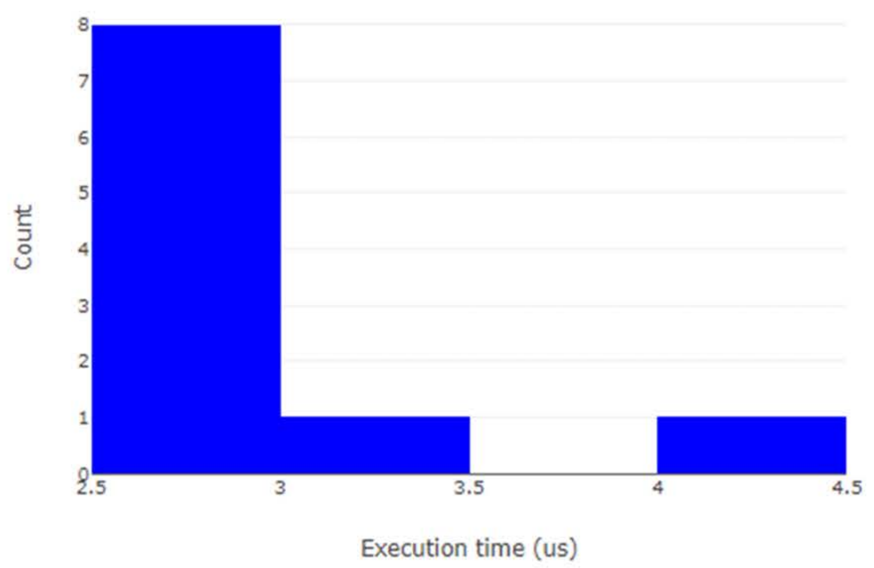
Execution time distribution in nanoseconds.

Num Frames Executed	10
Num Frames Skipped	0
Num Frames Missing	0
Scheduled Max Duration	1000.00 us
Mean Duration	3.06 us
Standard Deviation of Duration	0.32 us
Min Duration	2.91 us
Max Duration	4.03 us
99th Percentile Duration	3.94 us
Num Overruns	0
Overrun Fraction	0.00
Mean Overprovision	996.94 us
Mean Overprovision Fraction	1.00

Execution time across iterations



Execution time distribution



Per-Runnable Information

Get Started with DRIVE SDK

Extensive documentation and training material available on NVIDIA Developer

Learn More

- Visit the DRIVE training page for webinars and other resources
- Check out information related to DRIVE Hyperion, DRIVE AGX Orin and DRIVE SDK

Get Access

- Join the DRIVE AGX SDK Program on NVIDIA Developer
- Read the docs for DRIVE OS and DriveWorks documentation
- Download DRIVE OS which includes DriveWorks, NvMedia, CUDA, cuDNN and TensorRT

Contact Us

- Contact your distributor or NVIDIA's Automotive Team

NVIDIA®