

GPU & CMSSW Integration

The Service: Portability & Ease of Use

Konstantinos Samaras-Tsakiris

*School of Electrical and Computer Engineering
Aristotle University of Thessaloniki*

Dr. Vincenzo Innocente

CERN

Felice Pantaleo

CERN - University of Hamburg

Introduction

Introduction

Studies

- Electrical & Computer Engineering
- 3rd year Undergraduate student

Introduction

Studies

- Electrical & Computer Engineering
- 3rd year Undergraduate student

Interests

- Artificial Intelligence
- GPU computing
- Robotics

Project Goal

Incorporate GPUs into
CMS software framework

Project Goal

Incorporate GPUs into
CMS software framework

Tenets

- 1) Portability
- 2) Ease of Use
- 3) Efficiency

Project Goal

Incorporate GPUs into
CMS software framework

Tenets

- 1) Portability
- 2) Ease of Use
- 3) Efficiency

Solution

✓ GPU managing Service

Contents

- Problem Definition
- Service usefulness
- Service architecture
 - System overview
 - Elements
- Tests & Examples
- Challenges & Future Work

Problem Definition

Problem Definition

Problem Definition

- GPU accelerators
 - ✓ Massive parallelism

Problem Definition

- GPU accelerators
 - ✓ Massive parallelism
- But how to program them?

Problem Definition

- GPU accelerators
 - ✓ Massive parallelism
- But how to program them?
 - ✓ CUDA!

Well...

(not so fast)

```
1 float *a, *d_a;
2 a= (float*)malloc(sizeof(float));
3 cudaMalloc((void**) &d_a, sizeof(float))
4 *a= 1.0;
5 cudaMemcpy(d_a, a, sizeof(float),
             cudaMemcpyHostToDevice);
6 someKernel<<<grid,block>>>(d_a);
7 cudaMemcpy(a, d_a, sizeof(float),
             cudaMemcpyDeviceToHost);
8 cout << *a;
9 cudaFree(d_a); free(a);
```

Problem Definition

Problem Definition

x Cumbersome programming model

Problem Definition

- x Cumbersome programming model*
- x Too intricate*

Problem Definition

- x Cumbersome programming model*
- x Too intricate*
- x Not portable*
 - x Not even compiles with GCC*

Problem Definition

- x Cumbersome programming model*
- x Too intricate*
- x Not portable*
 - x Not even compiles with GCC*

→ On the other hand...

Example

Using CudaService

```
1 int size, param;  
2 cudaPointer<float> data(size);  
3 auto GPUResult= cudaService->  
4   cudaLaunch(size, someKernel_wrapper,  
5             param, data);  
6   ...<other work>...  
7 GPUResult.get();
```

Example

Using CudaService

```
1 int size, param;  
2 cudaPointer<float> data(size);  
3 auto GPUResult= cudaService->  
4   cudaLaunch(size, someKernel_wrapper,  
5             param, data);  
6   ...<other work>...  
7 GPUResult.get();
```

→ *Prettier, isn't it?*

Service Usefulness

Service Usefulness

- New capability
 - Write CUDA in CMSSW
 - Call from normal C++

Service Usefulness

- Portable
 - Handle common GPU failures
 - e.g. temporarily insufficient memory
 - Plug-in fallback CPU versions
 - Manual fallback, or...
 - *Luca's work* fits here

Service Usefulness

- Simplified model
 - Launch task → Get future
 - CudaPointer
 - ✓ Manages CUDA memory
 - Auto launch configuration
 - x Only for 1D configurations

CUDA Research

CUDA Research

- Unified Memory
 - Auto memory management
 - Future hardware support

CUDA Research

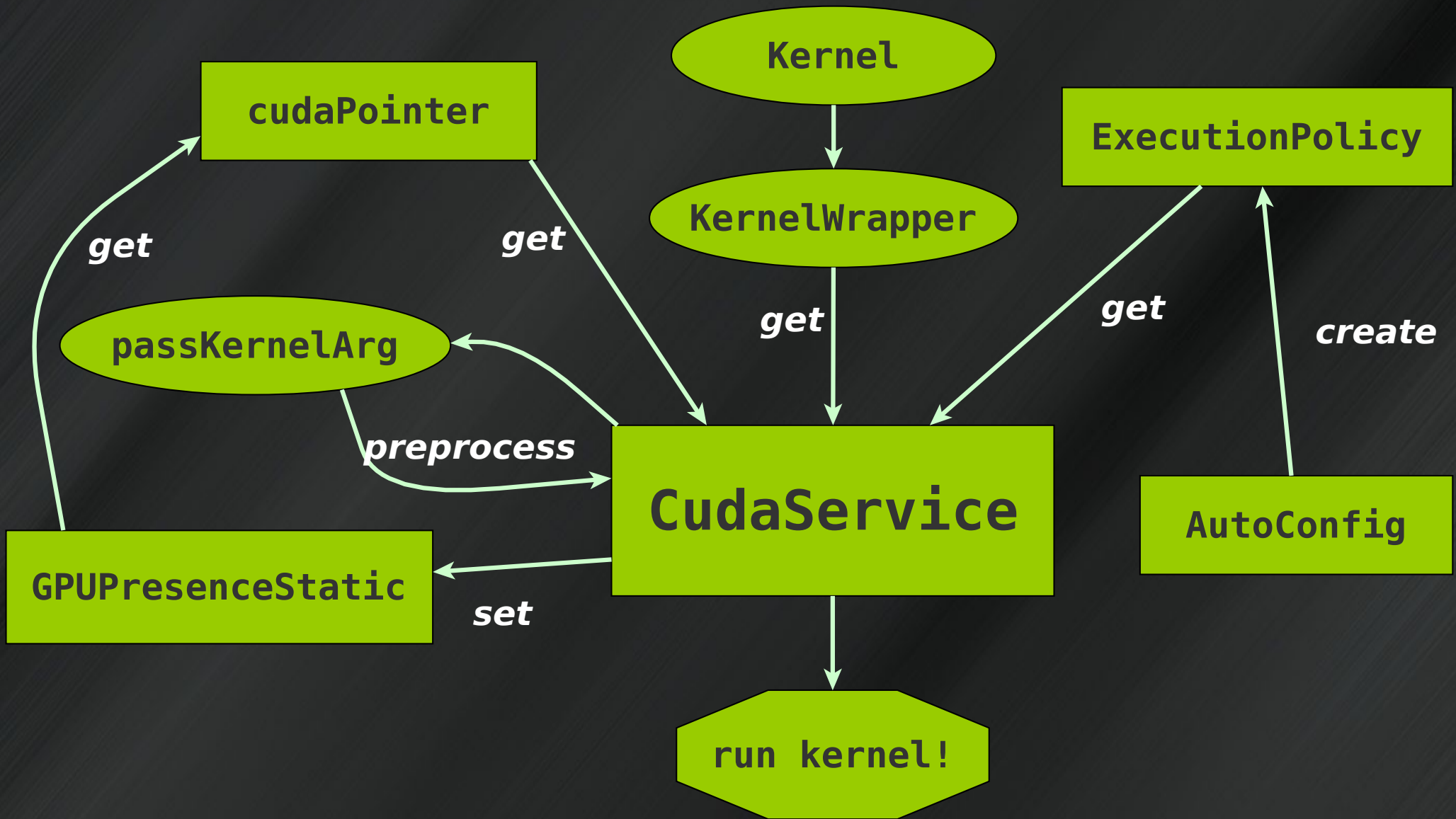
- Unified Memory
 - Auto memory management
 - Future hardware support
- Task parallelism – Streams
 - 1 per host thread (compiler option)
 - Overlap data movement & execution

CUDA Research

- Unified Memory
 - Auto memory management
 - Future hardware support
- Task parallelism – Streams
 - 1 per host thread (compiler option)
 - Overlap data movement & execution
- Auto kernel launch configuration

Service Architecture

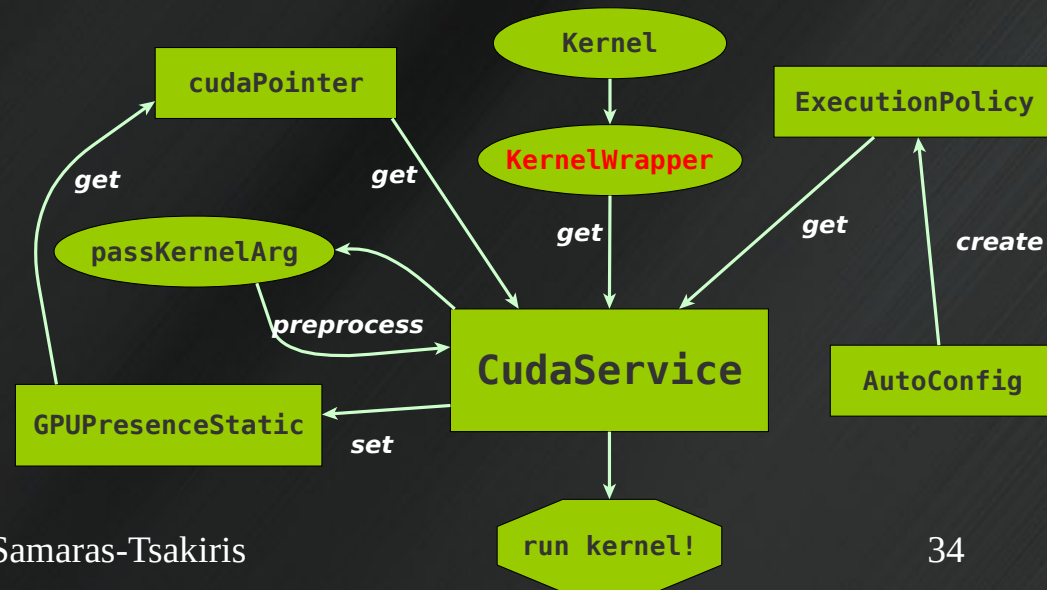
Overview



Let's check each element...

Kernel Wrappers

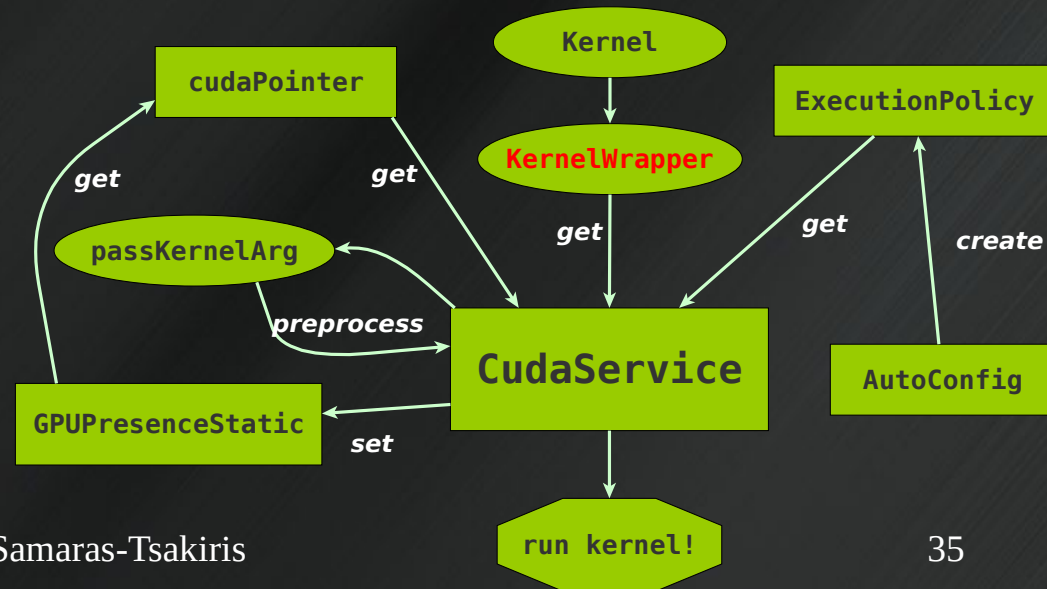
```
1  __global__ void kernel(int param);  
2  void Wrapper(bool gpu, int& launchSize,  
3     int param){  
4     auto execPol= cuda::AutoConfig()  
5         (launchSize, task_kern);  
6     if (gpu) kernel<<<execPol.getGridSize(),  
7         execPol.getBlockSize()>>>(param);  
8 }
```



Kernel Wrappers

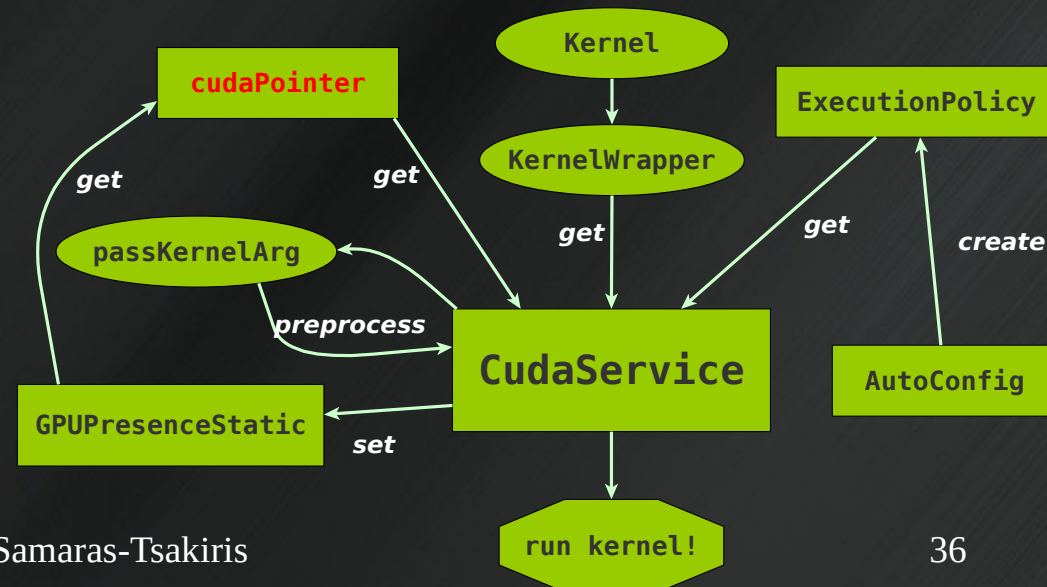
```
1  __global__ void kernel(int param);  
2  void Wrapper(bool gpu, int& launchSize,  
3     int param){  
4     auto execPol= cuda::AutoConfig()  
5         (launchSize, task_kern);  
6     if (gpu) kernel<<<execPol.getGridSize(),  
7         execPol.getBlockSize()>>>(param);  
8 }
```

User writes
Kernel & Wrapper



cudaPointer

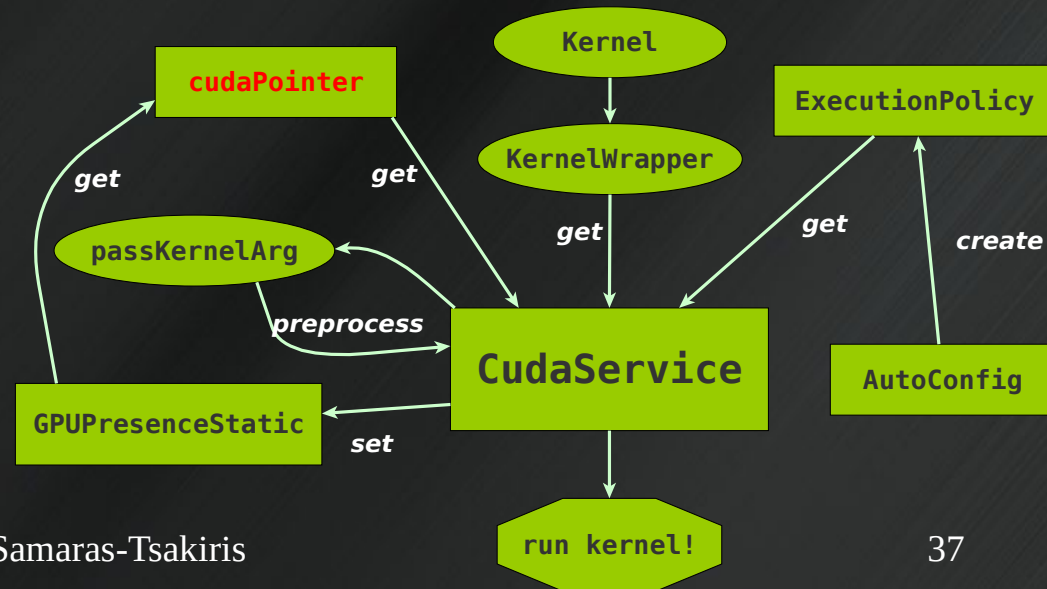
- Smart pointer → `std::unique_ptr`
- Construct and move semantics – no copy
- Handles stream attachment
- Portable allocation/deallocation
- 2 flavours:
 - Single objects
 - Arrays



cudaPointer

```
1 __global__ void kernel(int a, int* b,  
                        float* c)
```

should be called as...

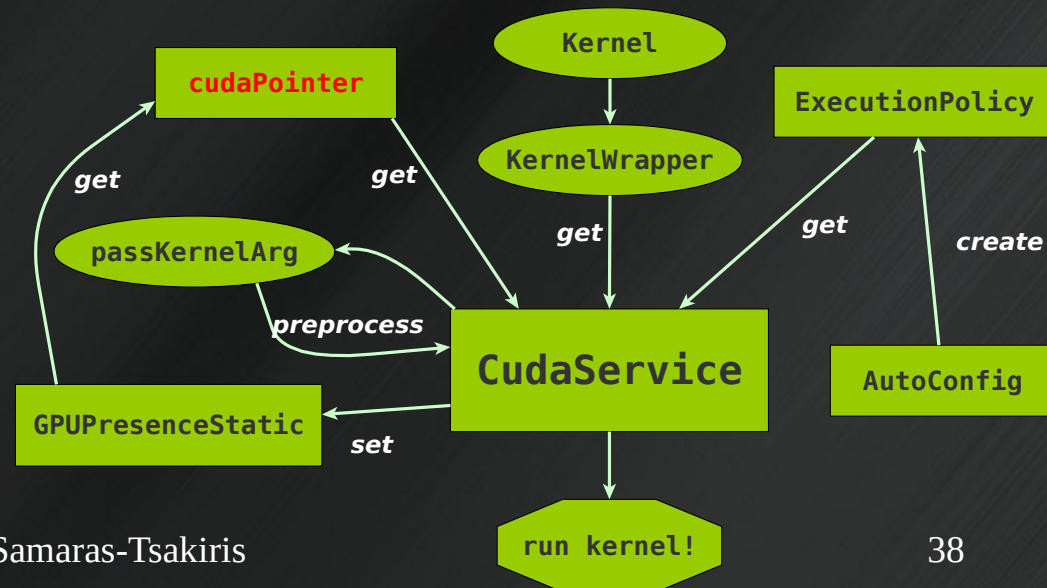


cudaPointer

```
1 __global__ void kernel(int a, int* b,  
                        float* c)
```

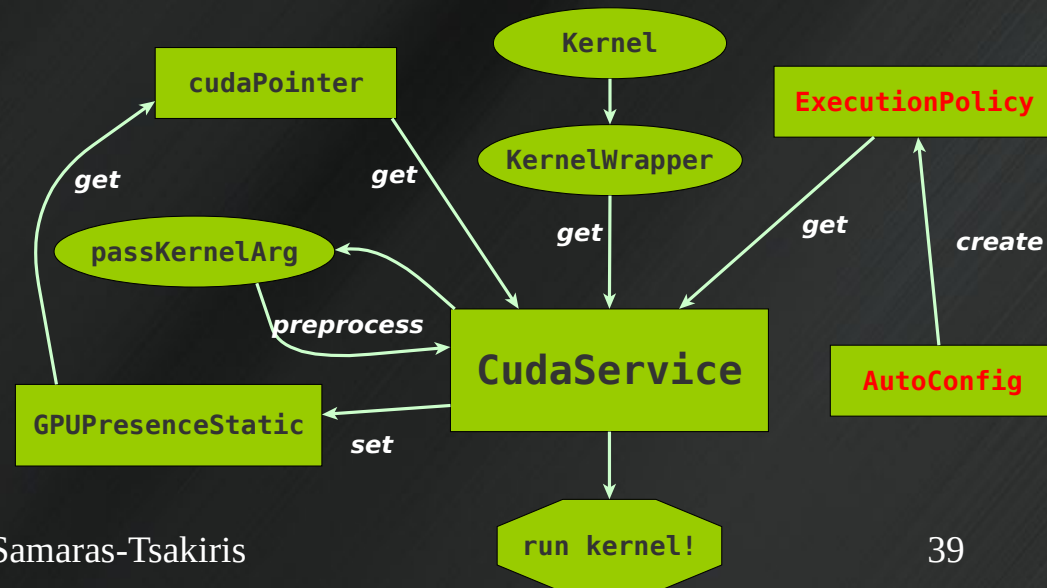
should be called as...

```
2 kernel_wrapper(int, cudaPointer<int[]>,  
3               cudaPointer<float[]>)
```



ExecutionPolicy

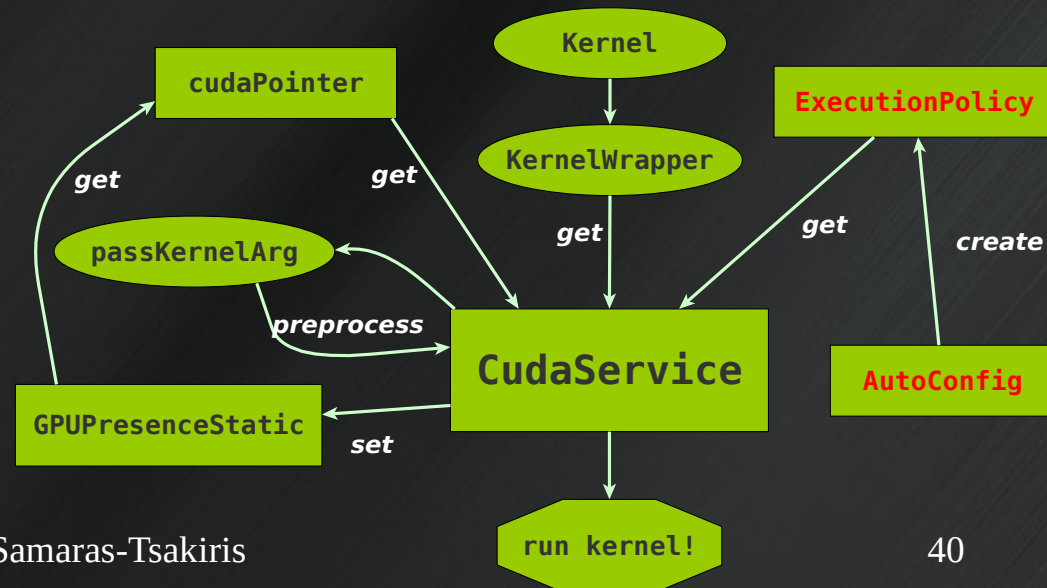
- Grid & Block size (multidimensional)
- AutoConfig:
 - Creates ExecutionPolicy
 - Automatically calculates block size (occupancy)



ExecutionPolicy

- Grid & Block size (multidimensional)
- AutoConfig:
 - Creates ExecutionPolicy
 - Automatically calculates block size (occupancy)

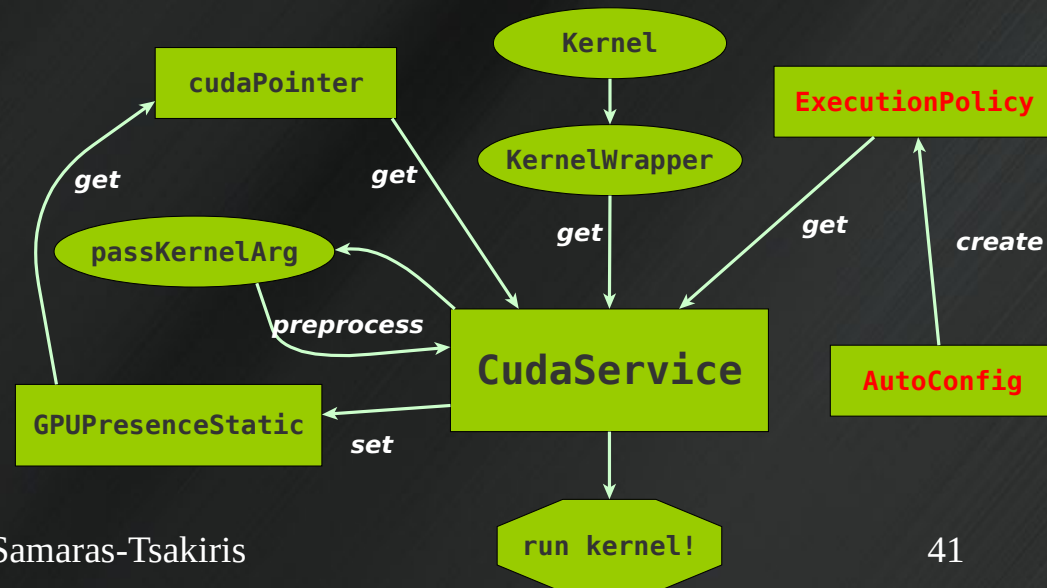
Based on
Mark Harris' HEMI



Example

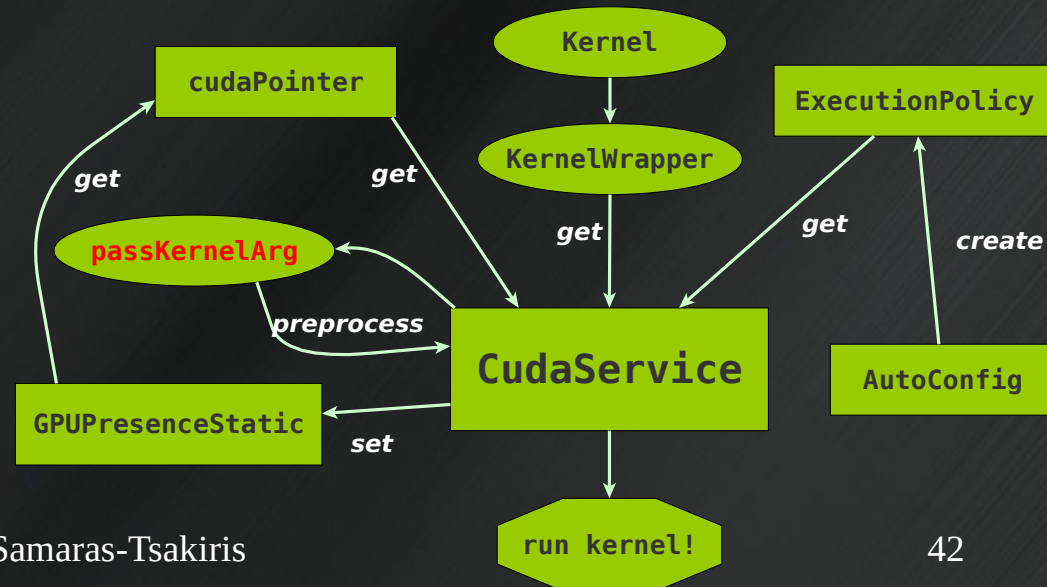
Creating an ExecutionPolicy

```
1 auto execPol= cuda::AutoConfig() (n,  
2                                     (void*) kernel);
```



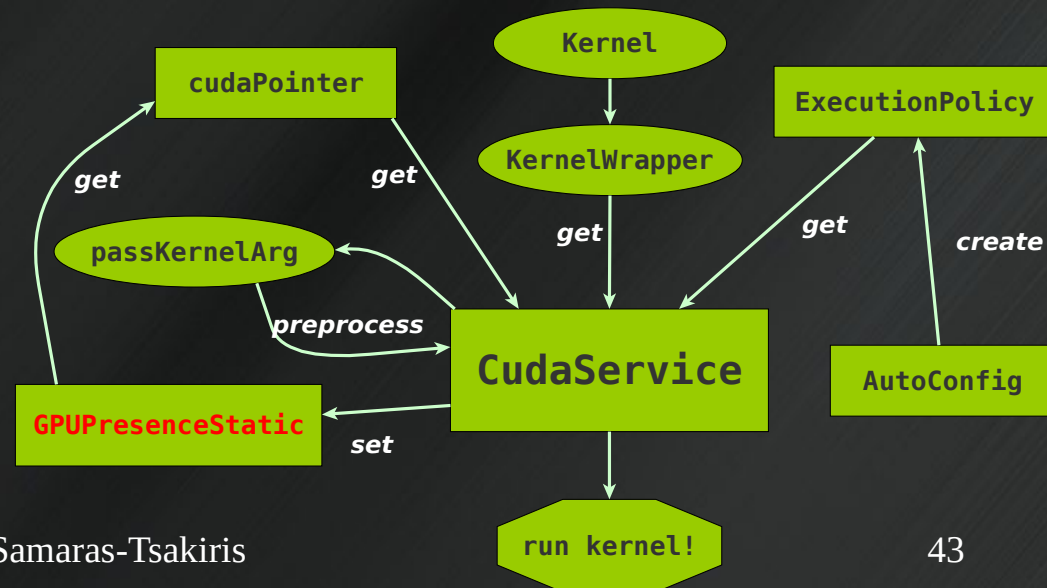
passKernelArg

- Pair of template functions
 - Template type identification – SFINAE
 - One for cudaPointer, one for everything else
- Signal cudaPointers → Passed to kernel
 - Change Stream attachment
- Siblings:
 - releaseKernelArg



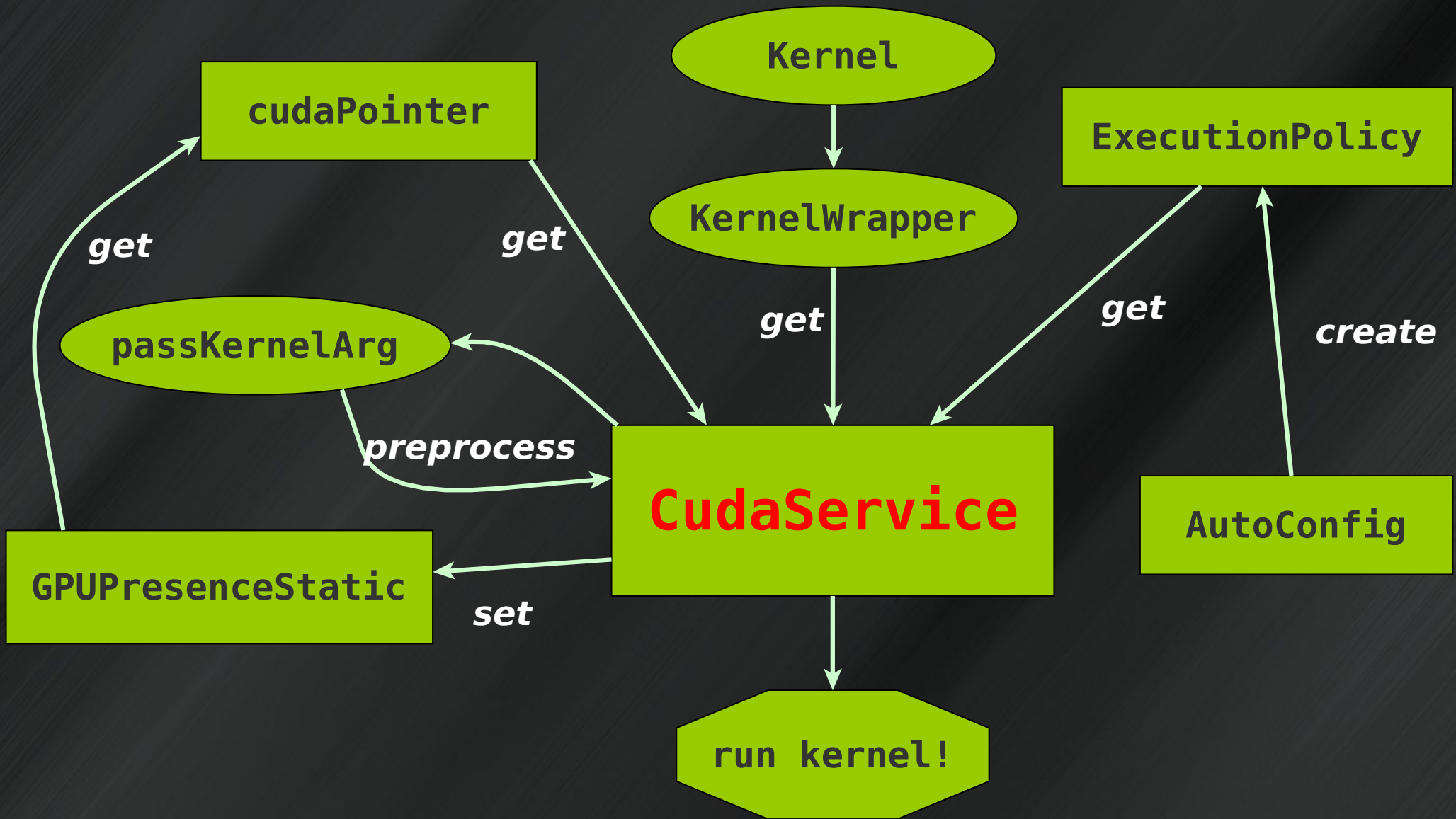
GPUPresenceStatic

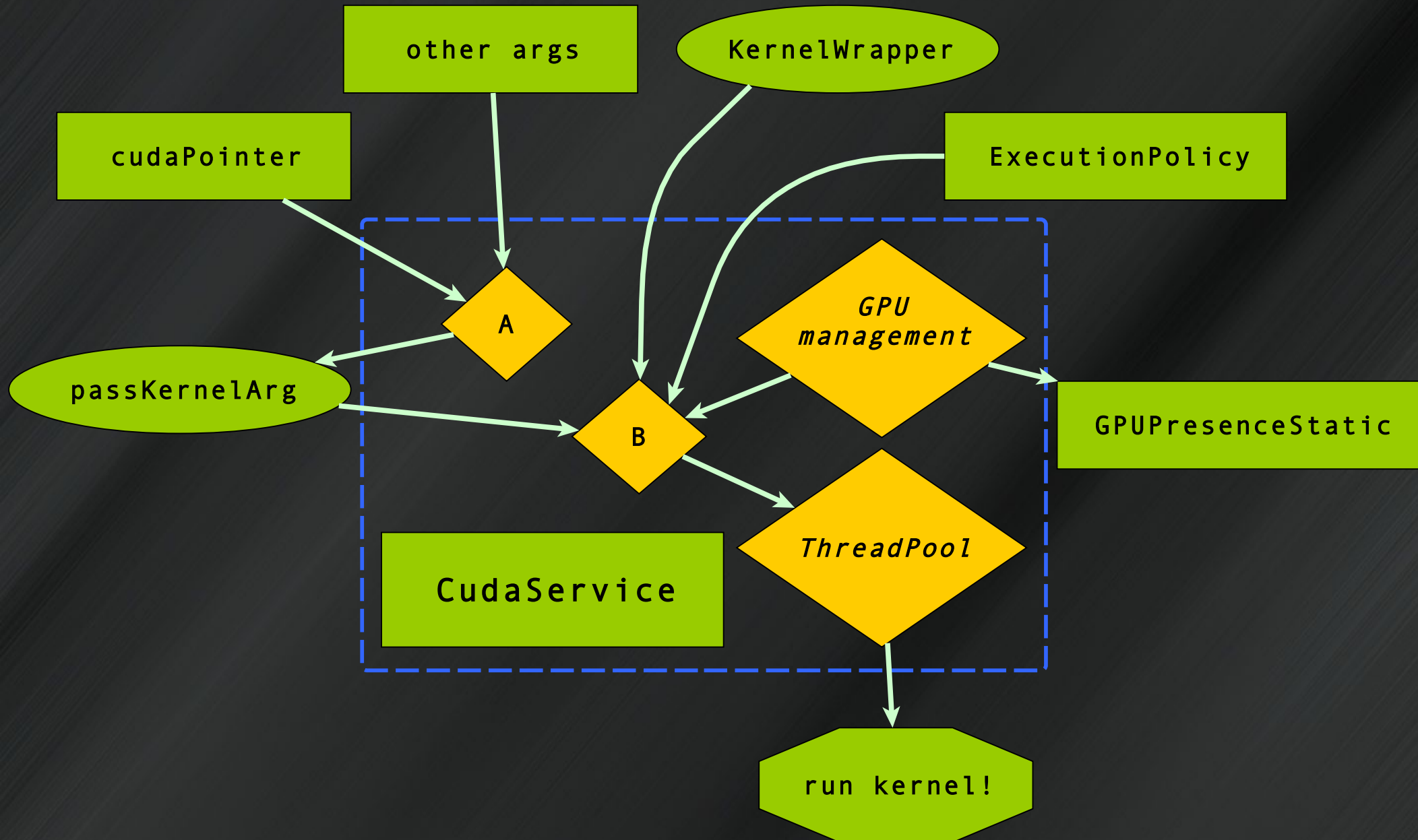
- Broadcasts GPU presence
- Conditionally accessible
 - Static member: status_
 - SET: only CudaService
 - SFINAE



Service Architecture

Inside CudaService

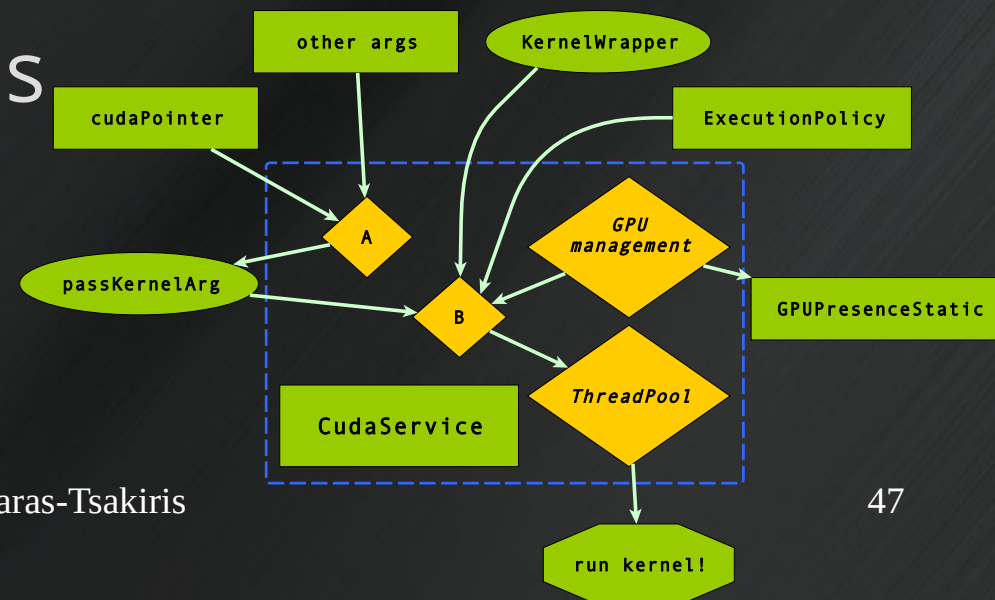




CudaService

Lifecycle

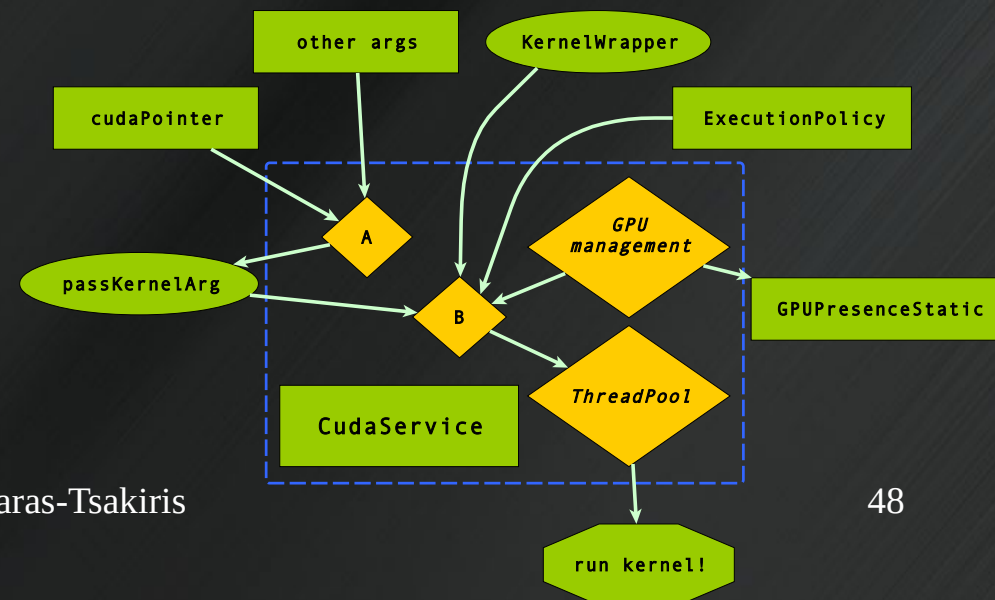
- “Default” constructor only
 - Checks number of GPUs present
 - Registers start/end work callbacks
- No copy/move semantics
- Shared between threads
 - Never > 1!
 - Thread-safe



CudaService

State

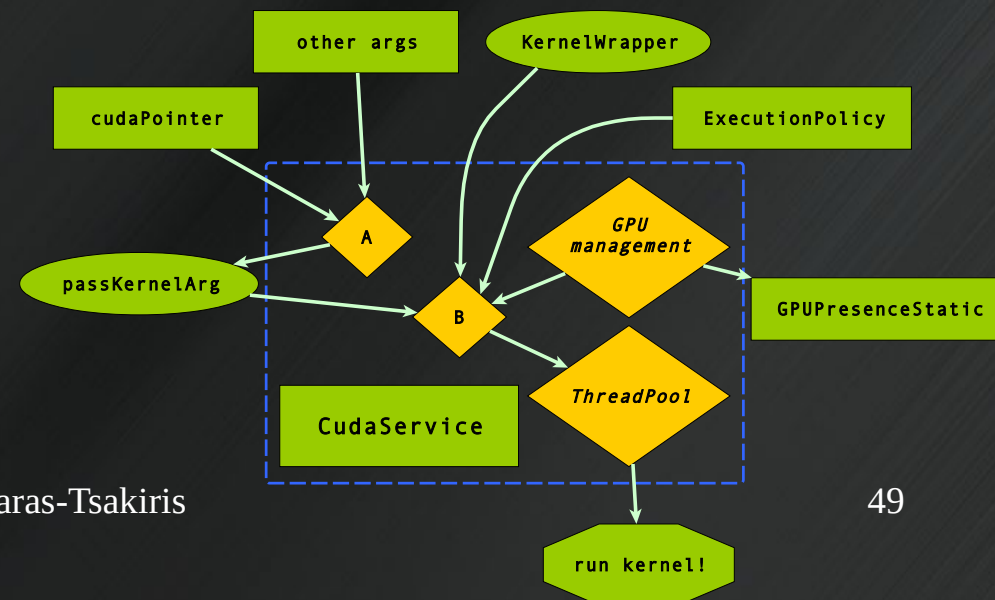
- Derives from ThreadPool
- Thread-safe
- Task store
 - `tbb::concurrent_bounded_queue`
- Waiting threads
 - `std::vector`



CudaService

API

- `cudaLaunch(launch type, wrapper, args...)`
 - Manual | Auto launch ← SFINAE
- `schedule(callable, args...)`
 - For any CPU task
- `GPUpresent()`



Test-driven development

a.k.a. "Why should I trust you?"

Unit Tests

- Auto | manual | multidim launch configuration
- cudaPointer
 - cudaPointer to arithmetic struct
 - Struct containing cudaPointers: Not working yet!
- CPU task latency benchmark
- GPU kernel latency benchmark
- Share service with task (CPU task)
 - Validates recursive task launch

Launch configurations

Auto

```
1 cudaService->cudaLaunch(n, task_auto,  
2                          param).get();
```

Manual

```
3 auto execPol= cudaConfig::ExecutionPolicy  
4   (blockSize, (n-1+blockSize)/blockSize);  
5 cudaService->cudaLaunch(execPol, task_man,  
6                          param).get();
```

Multidimensional (manual)

```
7 execPol.setBlockSize({blockSize, blockSize  
8                      }).autoGrid({n,m});
```

CPU task latency

```
1 start= chrono::steady_clock::now();
2 for(register int task=0; task<threadN; task++)
3     futVec[task]= cudaService->schedule([] () {
4         for (register short k=0; k<2; k++)
5             cout<<" ";
6     });
7 for(auto&& elt: futVec) elt.get();
8 end = chrono::steady_clock::now();
```

CPU task latency

```
1 start= chrono::steady_clock::now();
2 for(register int task=0; task<threadN; task++)
3     futVec[task]= cudaService->schedule([] () {
4         for (register short k=0; k<2; k++)
5             cout<<" ";
6     });
7 for(auto&& elt: futVec) elt.get();
8 end = chrono::steady_clock::now();
```

2.2 μ s

GPU task latency

Nontrivial kernel

```
1 start= chrono::steady_clock::now();
2 for(register int task=0; task<threadN; task++)
3     futVec[task]= cudaService->cudaLaunch(
4         execPol,nontrivial,param,dataIn,dataOut);
5 for(auto&& elt: futVec) elt.get();
6 end = chrono::steady_clock::now();
```

GPU task latency

Nontrivial kernel

```
1 start= chrono::steady_clock::now();
2 for(register int task=0; task<threadN; task++)
3     futVec[task]= cudaService->cudaLaunch(
4         execPol,nontrivial,param,dataIn,dataOut);
5 for(auto&& elt: futVec) elt.get();
6 end = chrono::steady_clock::now();
```

$20 \mu s$

What do ***20μs*** mean?

- Kernel execution time?
- Data copy time?
- *Service overhead?*

20 μs

What do ***20μs*** mean?

- Kernel launch in traditional CUDA

✓ ***~10μs***

→ Kernel execution time?

→ Data copy time?

→ *Service overhead?*

20 μs

What do ***20μs*** mean?

- Kernel launch in traditional CUDA
 - ✓ ***~10μs***
 - Kernel execution time?
 - Data copy time?
- Thread pool latency
 - ✓ ***2μs***
 - *Service overhead?*

20 μs

GPU task latency

Trivial kernel

```
1 start= chrono::steady_clock::now();
2 for(register int task=0; task<threadN; task++)
3     futVec[task]= cudaService->cudaLaunch(
4         execPol, trivial);
5 for(auto&& elt: futVec) elt.get();
6 end = chrono::steady_clock::now();
```

As little as launching a kernel in traditional CUDA!

GPU task latency

Trivial kernel

```
1 start= chrono::steady_clock::now();
2 for(register int task=0; task<threadN; task++)
3     futVec[task]= cudaService->cudaLaunch(
4         execPol, trivial);
5 for(auto&& elt: futVec) elt.get();
6 end = chrono::steady_clock::now();
```

9.5 μ s

As little as launching a kernel in traditional CUDA!

Latency Hiding

- Related to number of ThreadPool threads
 - Optimal: 3-5 per GPU
 - Much higher latency when only 1 thread in pool
 - ✓ More threads (3-5) hide some latency
- Related to kernel execution/data transfer overlap?
 - ✓ Needs profiling

Integration Test

- Change physics code to call CudaService
- Python cfg script for cmsRun
- Call cmsRun with 4 threads
- Compare GPU with CPU results

Integration Test

Original CPU loop

```
1 for (unsigned int subcl_idx = 0;  
2     subcl_idx < meanExp; subcl_idx++) {  
3     if (cls[subcl_idx] != 0) {  
4         clx[subcl_idx] /= cls[subcl_idx];  
5         cly[subcl_idx] /= cls[subcl_idx];  
6     }  
7     cls[subcl_idx] = 0;  
8 }
```


Integration Test

Kernel Replacement

```
1  __global__ void original_kernel(unsigned
2      meanExp, float* cls, float* clx, float* cly)
3  {
4      unsigned i= blockDim.x*blockIdx.x+threadIdx.x;
5      if (i<meanExp) {
6          if (cls[i] != 0) {
7              clx[i] /= cls[i];
8              cly[i] /= cls[i];
9          }
10         cls[i]= 0;
11     }
12 }
```

Integration Test

Kernel Wrapper

```
1 void original_wrapper(bool gpu, unsigned&
2   launchSize, unsigned meanExp, float* cls,
3   float* clx, float* cly)
4 {
5   if(!gpu) original_CPU(meanExp, cls, clx, cly);
6   else{
7     auto execPol= cuda::AutoConfig()
8       (launchSize, (void*)original_kernel);
9     original_kernel<<<execPol.getGridSize(),
10      execPol.getBlockSize()>>>(
11      meanExp, cls, clx, cly);
12   }
13 }
```

Integration Test

Service call code

```
1 edm::Service<edm::service::CudaService>
2     cudaService;
3 cudaPointer<float> cls(meanExp),
4     clx(meanExp),
5     cly(meanExp);
6 auto GPUresult= cudaService->
7     cudaLaunch(meanExp, original_wrapper,
8     meanExp, cls, clx, cly);
9 GPUresult.get();
```

PASS!

PASS!

- ✓ Without GPU, uses CPU fallback

Documentation

Github

<https://github.com/Oblynx/cmssw/tree/kernelLaunch/FWCore/Services>

Documentation

Github

<https://github.com/Oblynx/cmssw/tree/kernelLaunch/FWCore/Services>

CMS Twiki tutorial

<https://twiki.cern.ch/twiki/bin/view/Main/CudaService>

Documentation

Github

<https://github.com/Oblynx/cmssw/tree/kernelLaunch/FWCore/Services>

CMS Twiki tutorial

<https://twiki.cern.ch/twiki/bin/view/Main/CudaService>



Knowledge gained

- C++11
- CUDA
- Thread Safety
 - *“or how I learned to worry and hate shared state”*
- Template idioms -- SFINAE, enable_if
- How CMSSW works

Future Work

- cudaPointer enhancements
 - Allow structs containing cudaPointers
 - Work with Thrust for GPU data structures
 - Use GPU texture/constant memory
- Automate kernel wrapper generation
- Profile CudaService
 - Investigate latency hiding

Future Work

- Explore GPU Libraries
 - ArrayFire & Thrust
 - × CUDA not always best option!

*Thank you
for your attention!*