CUDA Kernels with C++

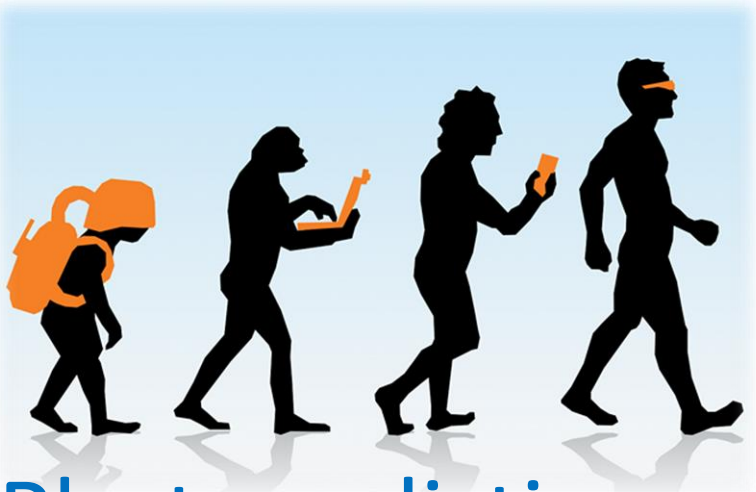
Michael Gopshtein

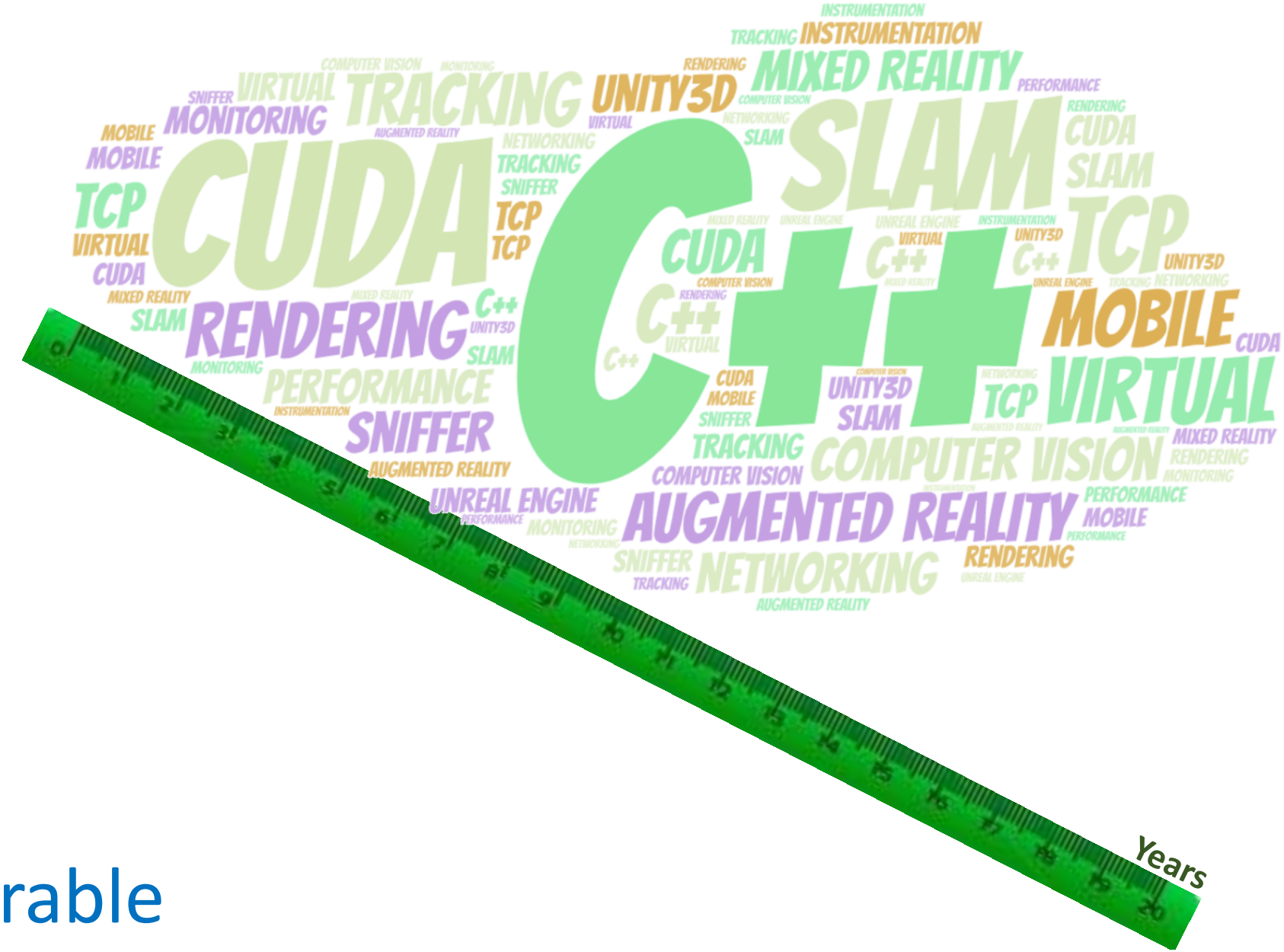
Sep 2018

About me

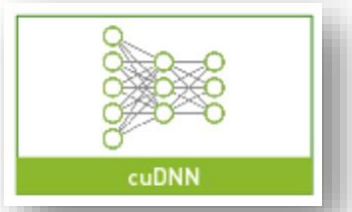
EYEWAY

ALL IS ILLUMINATED





Photorealistic wearable Augmented Reality experience.


cuDNN


cuFFT


Thrust
THRUST LIBRARY

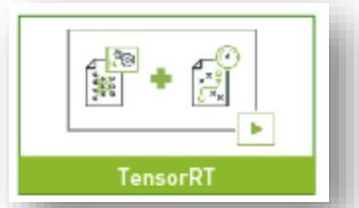

python


NVIDIA
CUDA

C++


COMPUTEWORKS
LANGUAGE
SUPPORT

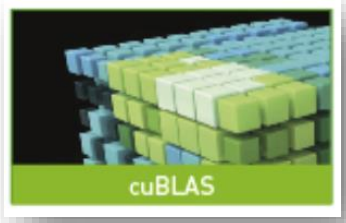

C


TensorRT


Fortran


LLVM
COMPILER
INFRASTRUCTURE

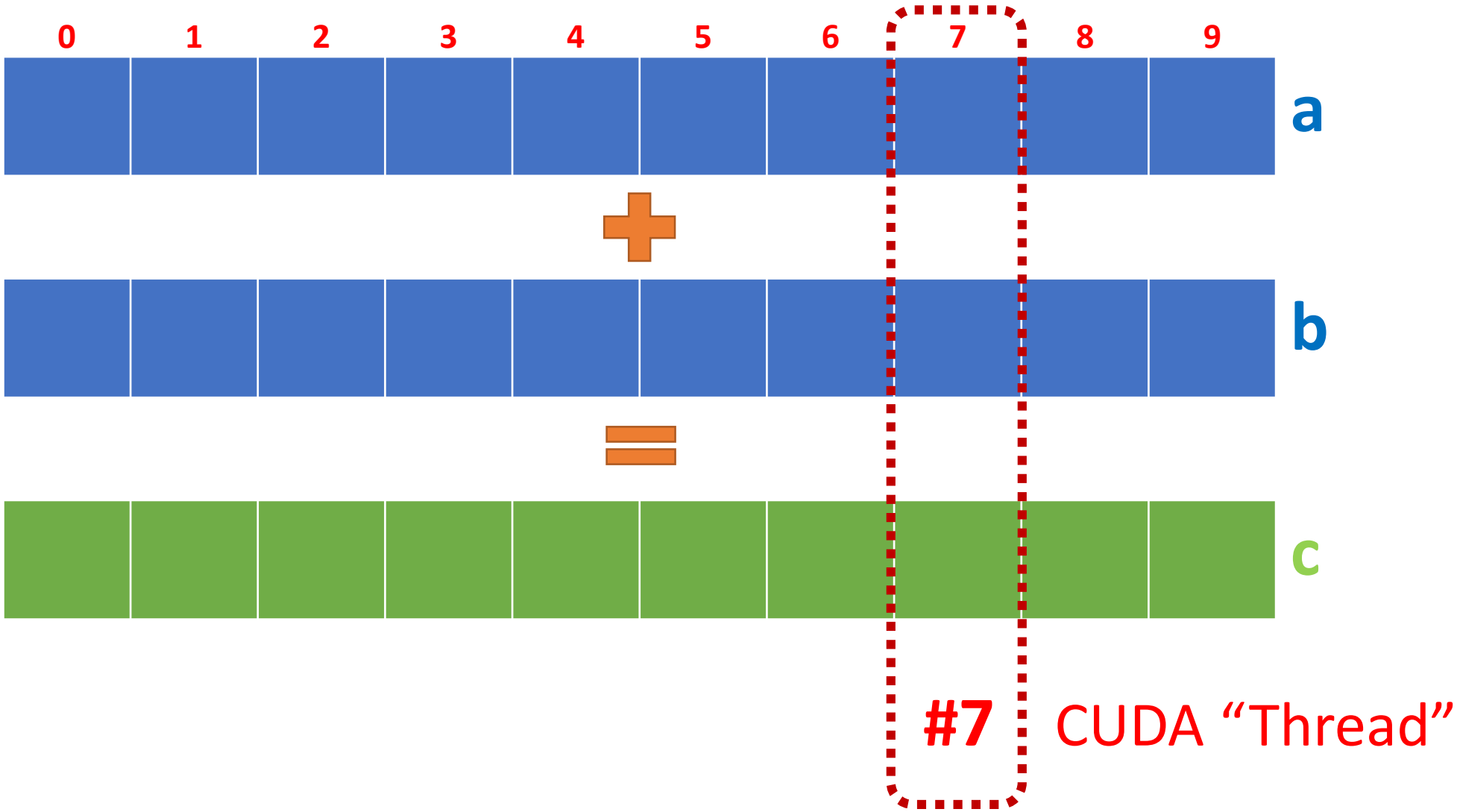

NVIDIA NPP


cuBLAS


NVIDIA®
Nsight™


NVIDIA CODEC SDK

Setting the Ground



Vector Addition in CUDA

Device

```
__global__ void addKernel(int *c, const int *a, const int *b) {
    int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
    c[idx] = a[idx] + b[idx];
}
```

Kernel/Device Code

Host

```
int main() {
  //…
  addKernel<<<blocks, 32>>>(dev_c, dev_a, dev_b);
  //…
}
```

Host/CPU Code

Vector Addition in CUDA

```
__global__ void addKernel(int *c                int *a, const int *b) {
    int idx = (blockIdx.x * b                .x) + threadIdx.x;
    c[idx] = a[idx] + b[
}

int main() {
  //…
  addKernel<<<blocks, 32>>>(dev_c, dev_a, dev_b);
  //…
}
```

GPU Accelerated Computing with C and C++

This is C

Where are the pluses?


Professional
CUDA C
Programming

Different Ways to Use C++

CUDA C/C++

OpenACC
OpenMP

Other libraries

Vector Addition in CUDA

```
__global__ void addKernel(int *c, const int *a, const int *b) {
    int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
    c[idx] = a[idx] + b[idx];
}

int main() {
  //…
  addKernel<<<blocks, 32>>>(dev_c, d
  //…
}
```

What if we have *float* arrays?

C Way

Device

```
__global__ void addKernel(int *c, const int *a, const int *b) {
    int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
    c[idx] = a[idx] + b[idx];
}

__global__ void addKernelF(float *c, const float *a, const float *b) {
    int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
    c[idx] = a[idx] + b[idx];
}
```

In C++ it's easy!

Device

```
template<typename T>
__global__ void addKernel(T *c, const T *a, const T *b) {
    int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
    c[idx] = a[idx] + b[idx];
}
```

Host

```
addKernel<int><<<blocks, 32>>>(dev_c, dev_a, dev_b);

addKernel<<<blocks, 32>>>(dev_c, dev_a, dev_b);
```

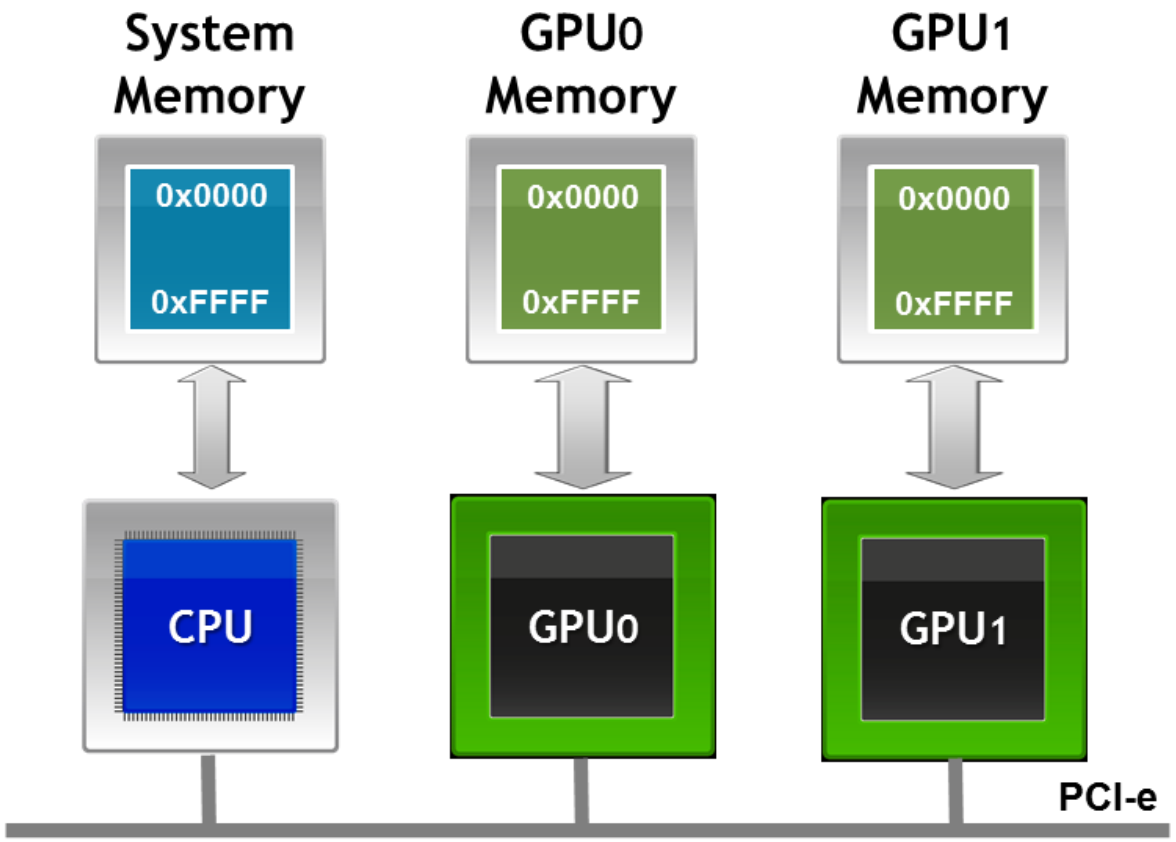
Device

Host

```
template<typename T>
__global__ void addKernel(T *c, const T *a, const T *b) {
    int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
    c[idx] = a[idx] + b[idx];
}
```

```
addKernel<<<blocks, 32>>>(dev_c, dev_a, dev_b);
```

What memory does it point to?



cudaMalloc allocates memory on the GPU

cudaMemcpy copies the vectors to/from GPU

Compiles, but fails in runtime

Device

Host

```
template<typename T>
__global__ void addKernel(T *c, const T *a, const T *b) {…}

int main() {
  const int a[SIZE] = {1, 2, … };
  const int b[SIZE] = {10, 20, …};
  int c[SIZE];

  addKernel<<<blocks, 32>>>(c, a, b);
}
```

Let’s use explicit device memory pointers

Device

```
template<typename T>
__global__ void addKernel(
  DevicePtr<T> c,
  DevicePtr<const T> a,
  DevicePtr<const T> b
){…}
```

```
template<typename T>
class DevicePtr {
  T *_p = nullptr;

public:
  __device__ __host__ __inline__ explicit DevicePtr(T *p) : _p(p) {}

  //…
};

template<typename T>
__host__ inline auto MakeDevicePtr(T* p) {
  return DevicePtr<T>(p);
}
```

Host

Host

Explicit creation from raw **T***

Convenience global function

Simple usage

Host

```
void process(const int *a, const int *b, int *c) {
  int *aDev;
  cudaMalloc(&aDev, LEN);
  cudaMemcpy(aDev, a, LEN, cudaMemcpyHostToDevice);
  //… same for bDev(alloc+copy) and cDev(alloc)

  addKernel<<<blocks, 32>>>(MakeDevicePtr(cDev),
    MakeDevicePtr(aDev), MakeDevicePtr(bDev));

  cudaMemcpy(c, cDev, LEN, cudaMemcpyDeviceToHost);
  cudaFree(aDev);  // free bDev, cDev
}
```

Even simpler usage

Host

```
void process(const int *a, const int *b, int *c) {
  auto aDev = DeviceMemory<int>::AllocateElements(NUM);
  CopyElements(aDev, a, NUM);
  //… same for bDev(alloc+copy) and cDev(alloc)

  addKernel<<<blocks, 32>>>(cDev, aDev, bDev);

  CopyElements(c, cDev, LEN);
}
```

Host

```
template<typename T>
class DeviceMemory {
  T *_p = nullptr;
  DeviceMemory(std::size_t bytes) { cudaMalloc(&_p, _bytes); }

public:
  static DeviceMemory AllocateElements(std::size_t n) {return {n*sizeof(T)}; }
  static DeviceMemory AllocateBytes(std::size_t bytes) {return {bytes}; }
  ~DeviceMemory() { if (_p) {cudaFree(_p);} }

  operator DevicePtr<T>() const {
    return DevicePtr<T>(_p);
  }
};
```

```
DevicePtr<int> a;
DevicePtr<const int> b = a;
DevicePtr<int> c = b;
DevicePtr<char> d = a;
```

<type_traits>

DevicePtr<**T1**> ➔ DevicePtr<**T**>

iff

T1* ➔ **T***

```
template<typename T>
class DevicePtr {
  T *_p = nullptr;

  template<typename T1,
           typename = std::enable_if_t<std::is_convertible_v<T1*, T*>>>
  __device__ __host__ DevicePtr(const DevicePtr<T1> &dp)
     : _p(dp.get())
  {}
};
```

Static Polymorphism

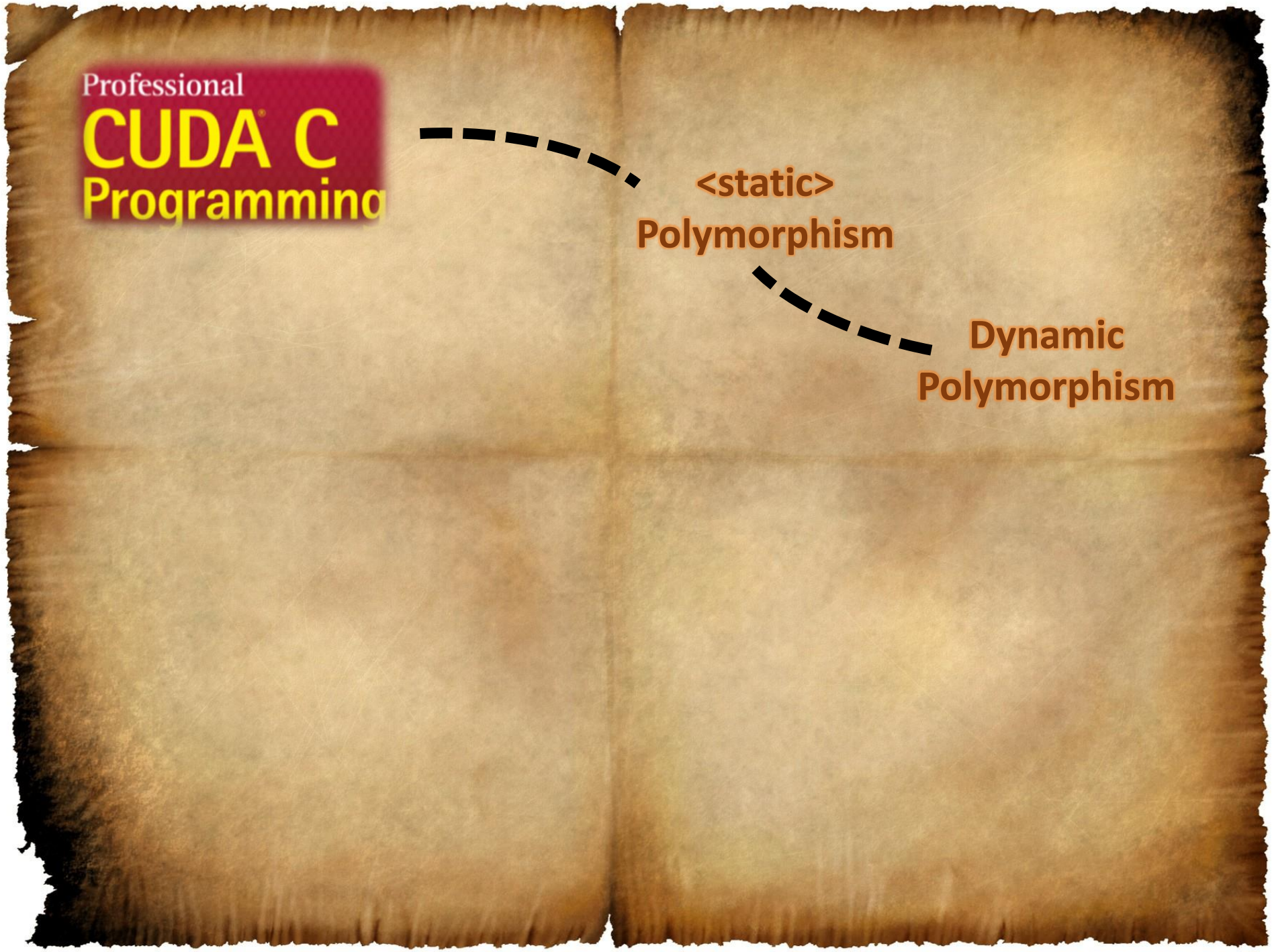
Device

```
template<typename T>
struct BinaryOpPlus {
  __device__ T operator()(T t1, T t2) const override { return t1 + t2; }
};

template<template<typename> typename OP, typename T> __global__
void addKernel(DevPtr<T> c, DevPtr<const T> a, DevPtr<const T> b) {
  OP<T> op;
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = op(a[i], b[i]);
}

addKernel<BinaryOpPlus><<<blocks, 32>>>(cDev, aDev, bDev);
```


Professional
CUDA C
Programming
<static>
Polymorphism


Professional
CUDA C
Programming
<static>
Polymorphism
Dynamic
Polymorphism

Device

```
template<typename T>
struct BinaryOp {
  virtual __device__ T operator()(T t1, T t2) const = 0;
};
template<typename T>
struct BinaryOpPlus : public BinaryOp<T> {
  __device__ T operator()(T t1, T t2) const override { return t1 + t2; }
};
```

Device

```
template<typename T>
__device__ void worker(DevPtr<T> c, DevPtr<const T> a, DevPtr<const T> b, const BinaryOp<T> &op) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = op(a[i], b[i]);
}
```

Device

```
template<typename T>
__global__ void addKernel(DevPtr<T> c, DevPtr<const T> a, DevPtr<const T> b) {
  worker(c, a, b, BinaryOpPlus<T>{});
}
```

It is not allowed to pass as an argument to a ***__global__*** *function an object of a class derived from virtual base classes*

but the business logic is in the CPU code…

Solution 1: from template to virtual

Device

```
template<typename T>
struct BinaryOpPlus : public BinaryOp<T>;

template<typename T> __device__
void worker(DevPtr<T> c, DevPtr<const T> a, DevPtr<const T> b,
                  const BinaryOp<T> &op);

template<template<typename> typename OP, typename T> __global__
void addKernel(DevPtr<T> c, DevPtr<const T> a, DevPtr<const T> b) {
  worker(c, a, b, OP<T>{});
}

addKernel<BinaryOpPlus><<<blocks, 32>>>(cDev, aDev, bDev);
```

Solution 2: dynamic allocation

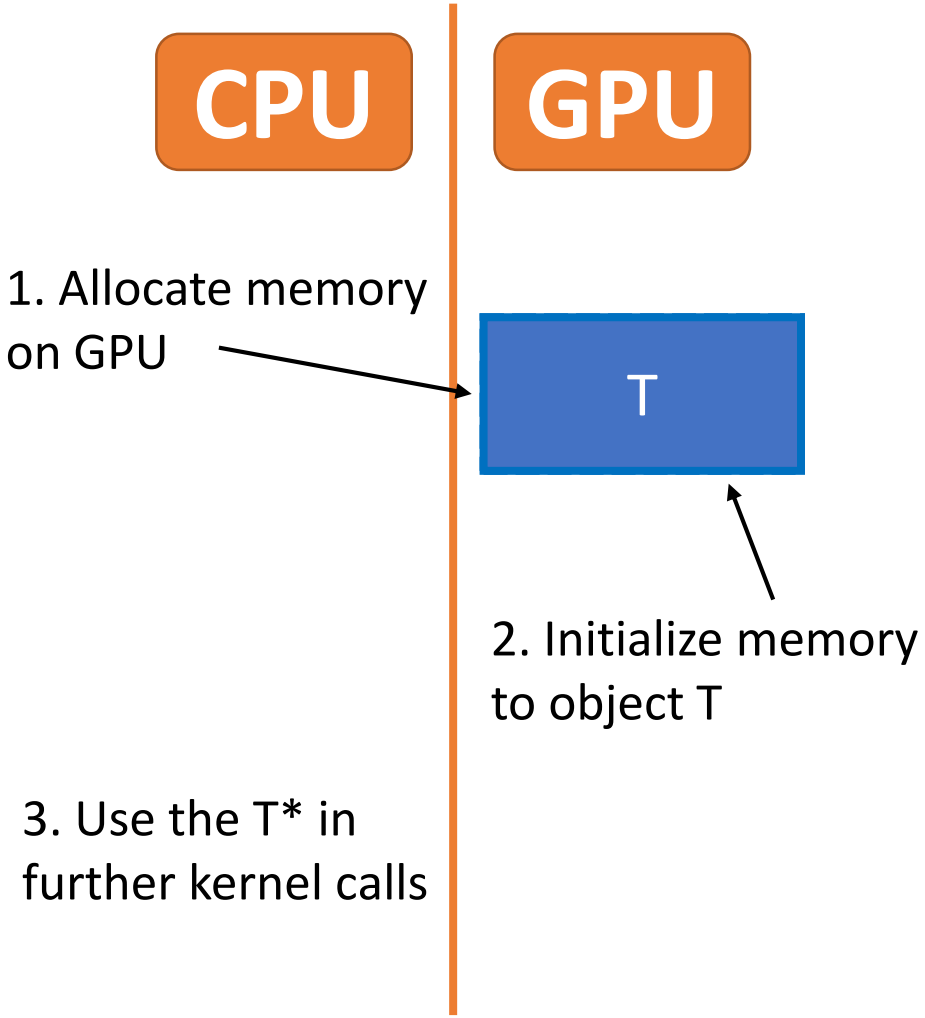
Device

```
template<typename T>
struct BinaryOpPlus : public BinaryOp<T>;

template<typename T> __global__
void addKernel(DevPtr<T> c, DevPtr<const T> a, DevPtr<const T> b,
               DevPtr<const BinaryOp<T>> op)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = (*op)(a[i], b[i]);
}

// create BinaryOpPlus on device
addKernel<<<...>>>(cDev, aDev, bDev, op);
```

Dynamic allocation in CUDA



Dynamic allocation in CUDA

CPU | **GPU**

1. Allocate memory on GPU

2. Initialize memory to object T

3. Use the T* in further kernel calls

```
template<typename T>
class DevObject {
  DevMemory<T> _p;

  DevObject()
    : _p(DeviceMemory<T>::AllocateElements(1))
  {

  }
};
```

Dynamic allocation in CUDA

CPU | **GPU**

1. Allocate memory on GPU

2. Initialize memory to object T

3. Use the T* in further kernel calls

```
namespace detail {
  template<typename T, typename... ARGS> __global__
  void AllocateObject(DevPtr<T> p, ARGS... args) {
    new (p.get()) T(args...);
  }
}


template<typename T>
class DevObject {
  DevMemory<T> _p;

  template<typename... ARGS>
  DevObject(ARGS... args)
    : _p(DeviceMemory<T>::AllocateElements(1))
  {
    detail::AllocateObject<T><<<1, 1>>>(_p, args...);
    cudaDeviceSyncronize();
  }
};
```

Device

Host

Dynamic allocation in CUDA

CPU | **GPU**

1. Allocate memory on GPU

2. Initialize memory to object T

3. Use the T* in further kernel calls

```
template<typename T>
class DevObject {
  DevMemory<T> _p;

  operator DevicePtr<T>() const {
    return DevicePtr<T>::FromRawDevicePtr(_p);
  }
};
```

Host

Dynamic allocation in CUDA

CPU

1. Allocate memory on GPU

3. Use the T* in further kernel calls

4. Release

GPU

2. Initialize memory to object T

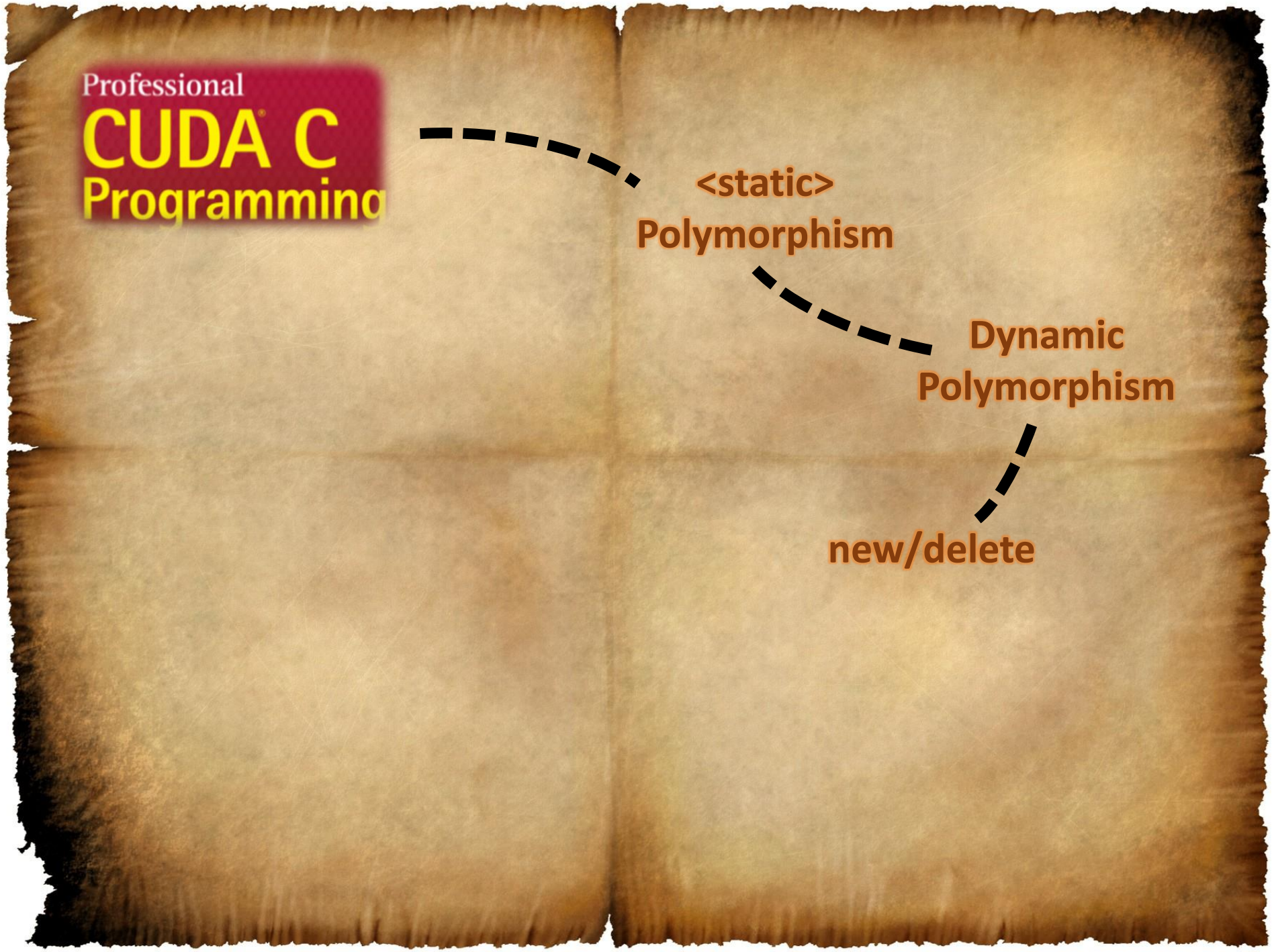
```
namespace detail {
  template<typename T> __global__
  void DeleteObject(DevPtr<T> p) {
    p->~T();
  }
}
```

```
template<typename T>
class DevObject {
  DevMemory<T> _p;

  ~DevObject()
  {
    if (_p) {
      detail::DeleteObject<T><<<1, 1>>>(_p);
      cudaDeviceSyncronize();
    }
  }
};
```

Device

Host


Professional
CUDA C
Programming
<static>
Polymorphism
Dynamic
Polymorphism
new/delete

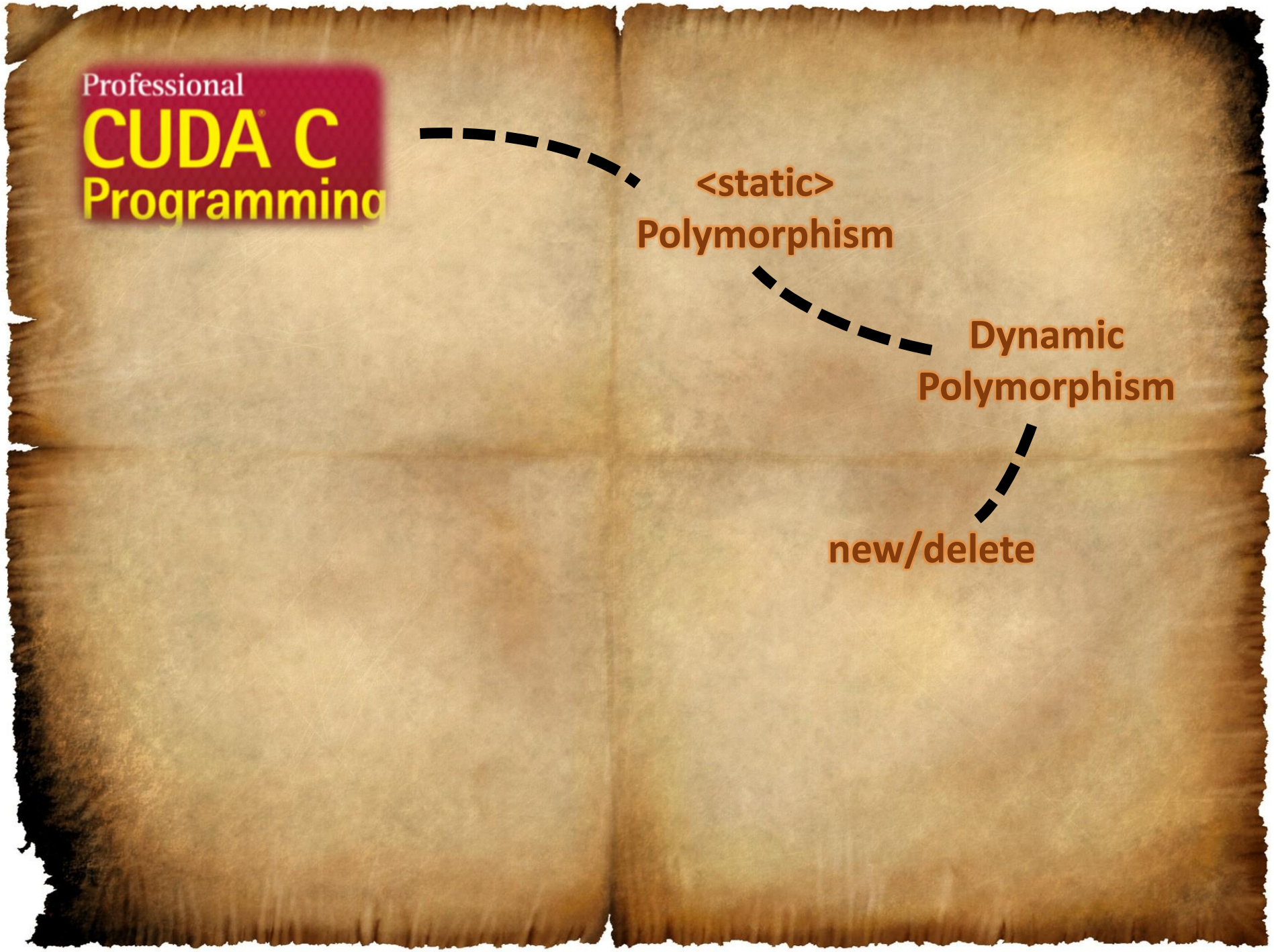
You can just use

malloc*/*free

and

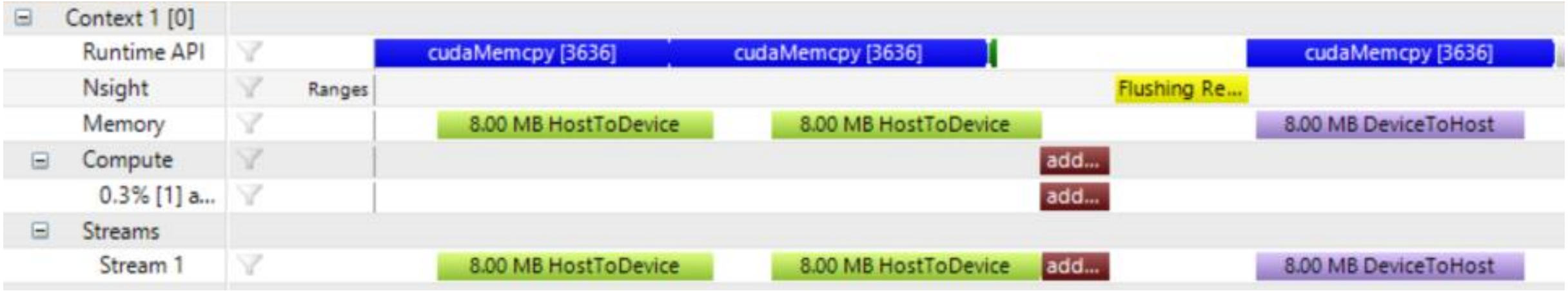
new*/*delete

in the kernel code

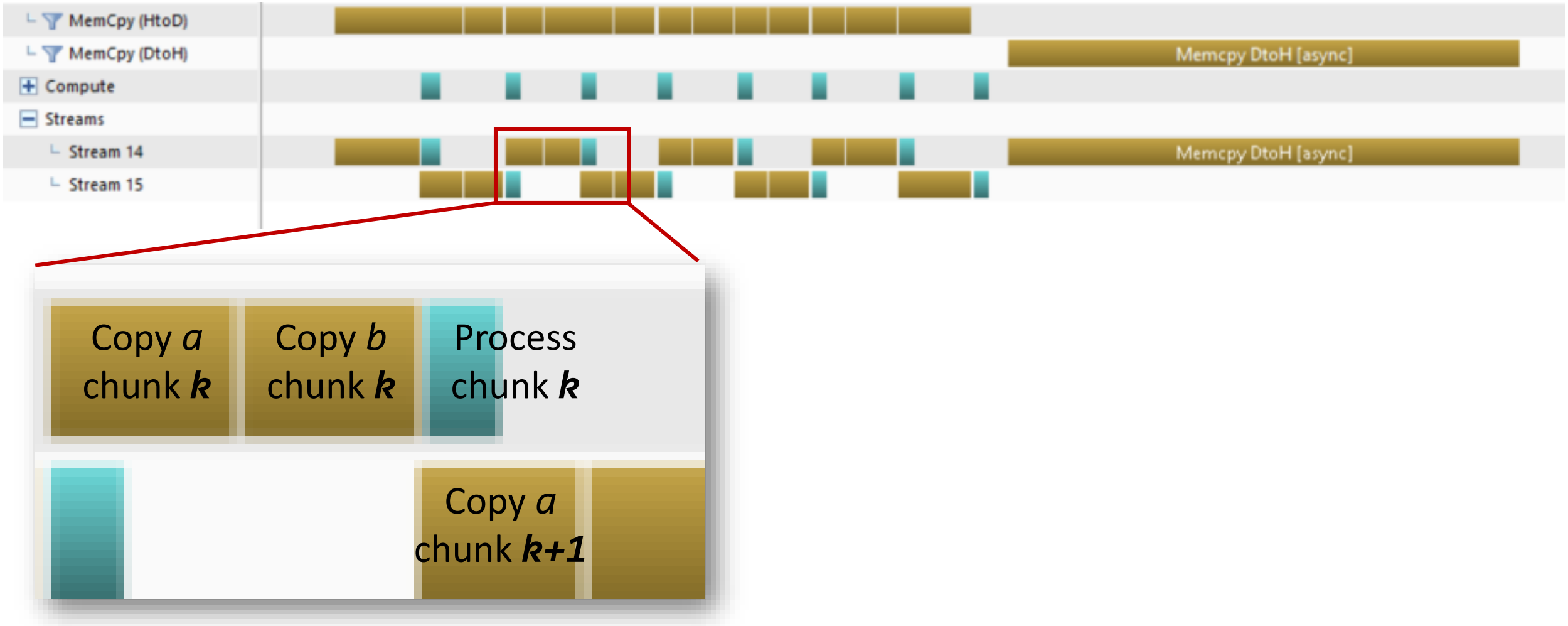

Professional
CUDA C
Programming
<static>
Polymorphism
Dynamic
Polymorphism
new/delete

Memory in CUDA

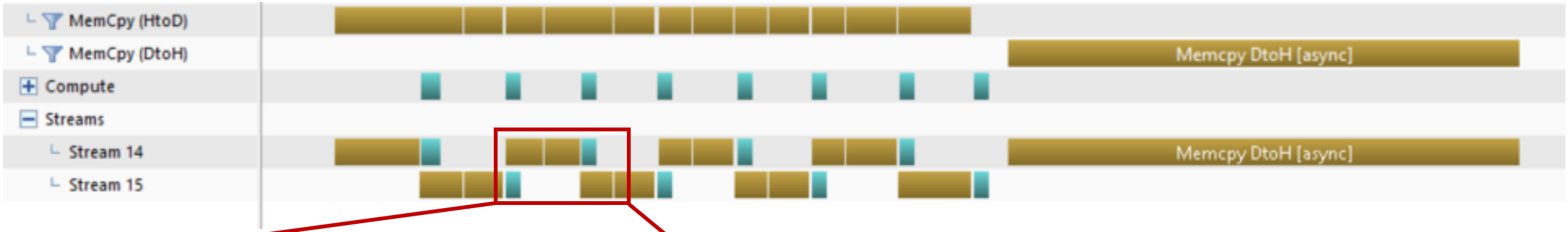
Synchronous Copy

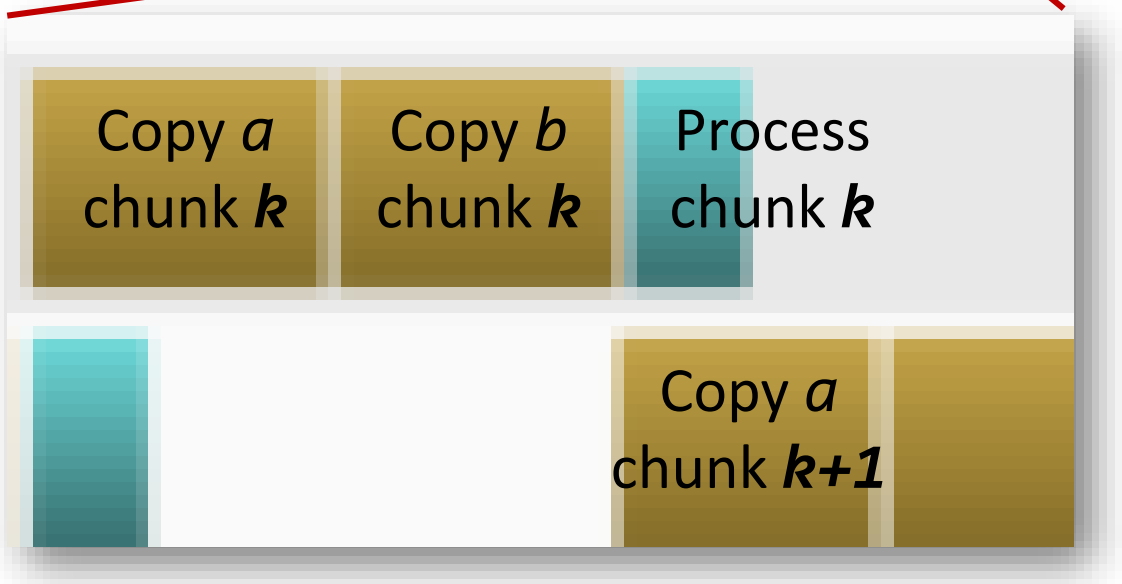


Asynchronous Copy



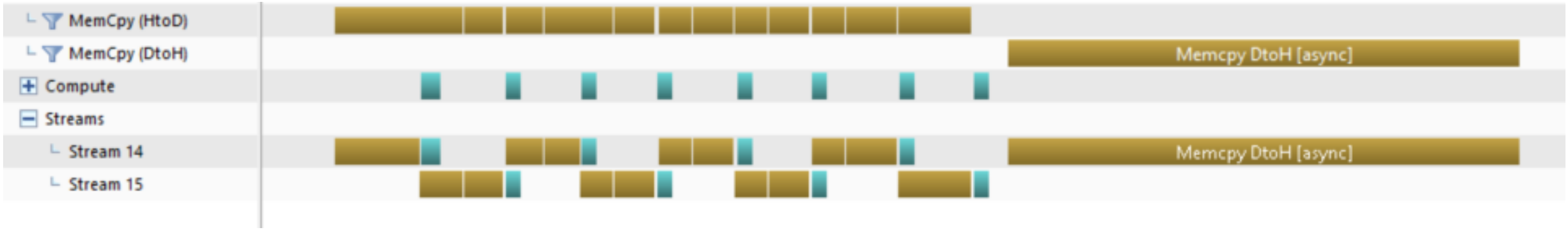
Asynchronous Copy





`cudaMemcpyAsync`

Asynchronous Copy



Pinned Host Memory

`cudaMemcpy`**`Async`**

`cuda`**`Host`**`Alloc`
`cuda`**`Host`**`Register`

Zero Copy Memory

Zero Copy Memory

1. Pin Memory (`cudaHostRegisterMapped`)
   `cudaHostAlloc`
   `cudaHostRegister`

Zero Copy Memory

1. Pin Memory (`cudaHostRegisterMapped`)
   `cudaHostAlloc`
   `cudaHostRegister`

2. `cudaHostGetDevicePointer`

3. Launch the kernel

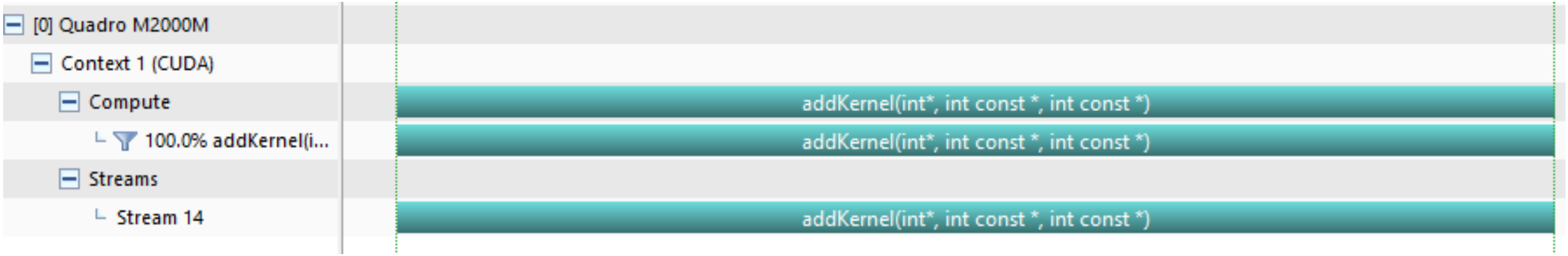
Zero Copy Memory

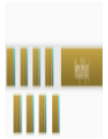
1. Pin Memory (`cudaHostRegister`**`Mapped`**)
   `cudaHostAlloc`
   `cudaHostRegister`

2. `cudaHost`**`tDevicePointer`**

3. Launch the kernel

Unified Virtual Addressing *(UVA)*





2.2 msec (Hand Made)

70.1 msec (Zero Copy)

Unified Memory

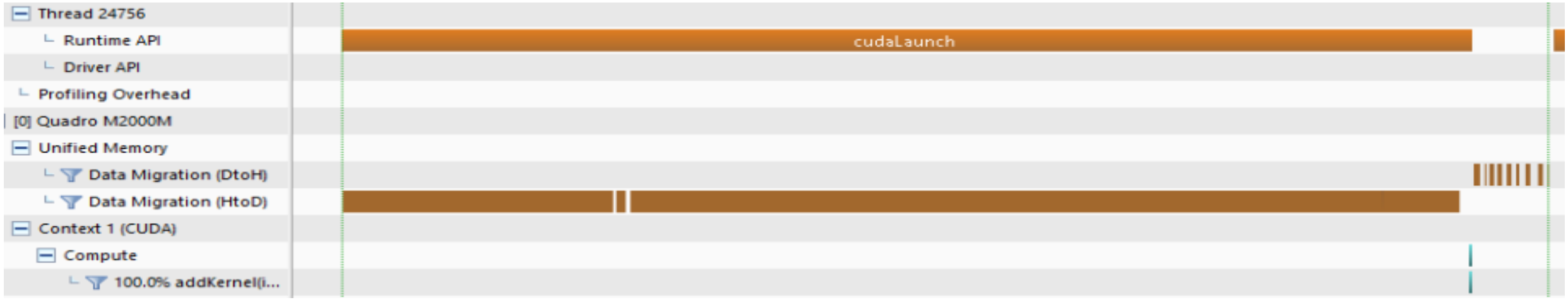
- Starting from CUDA 6
- Not to be confused with UVA!

Unified Memory

`cudaMallocManaged`

Memory migration:

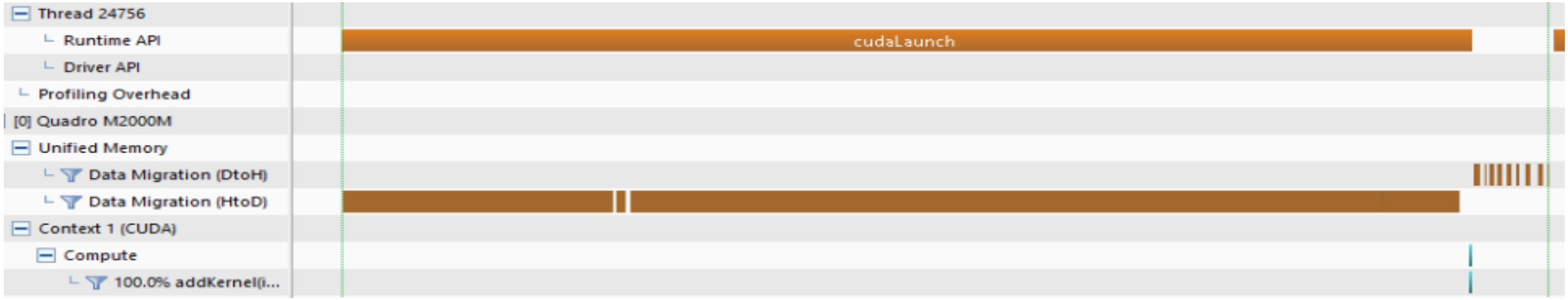
1. Data resides on the Host memory
2. Device code tries to read the memory
3. Page Fault → the driver *migrates* the memory to Device
   - Copy
4. CPU Page is swapped out



2.2 msec (Hand Made)

70.1 msec (Zero Copy)

85 msec (Unified Memory)



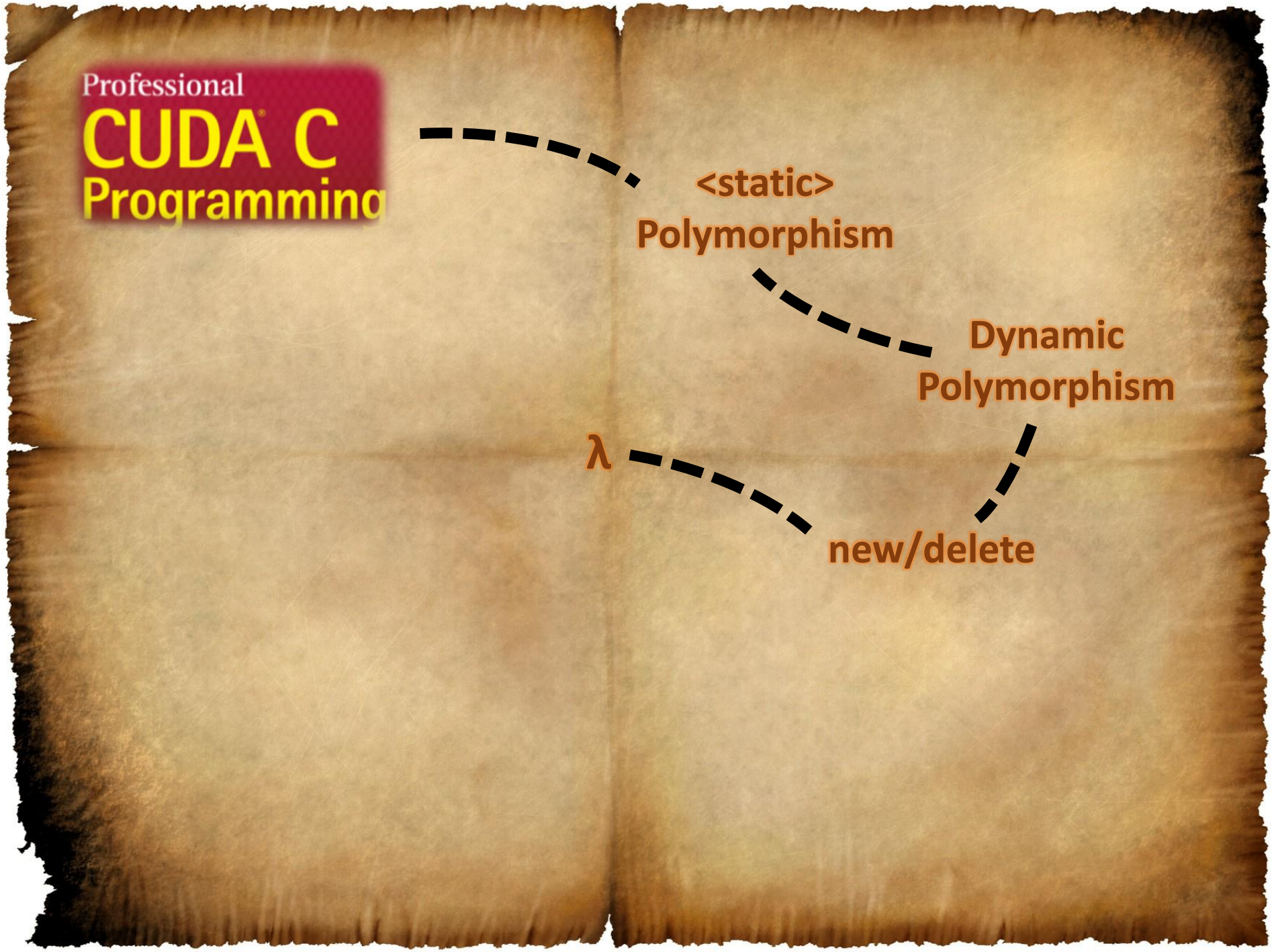
2.2 msec (Hand Made)

70.1 msec (Zero Copy)

85 msec (Unified Memory)

`cudaMemPrefetchAsync, cudaMemAdvise`

	Performance	Simplicity	No pinned memory	Sparse Access
Regular Copy	✓		✓	
Async Copy	✓✓			
Zero Copy	✗	✓		✓
Unified Memory	✗✓	✓✗	✓	✓


Professional
CUDA C
Programming
<static>
Polymorphism
Dynamic
Polymorphism
λ
new/delete

Simplest lambda

Device

```
template<typename T>
__global__ void addKernel(DevPtr<T> c, DevPtr<const T> a, DevPtr<const T> b) {
  int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
  auto op = [](auto a, auto b){ return a + b; };
  c[idx] = op(a[idx], b[idx]);
}
```

Regular capture rules apply

Device

```
template<typename T>
__global__ void addKernel(DevPtr<T> c, DevPtr<const T> a, DevPtr<const T> b) {
  int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
  auto op = [&]{ return a[idx] + b[idx]; };
  c[idx] = op();
}
```

Lambda parameters!!

Device

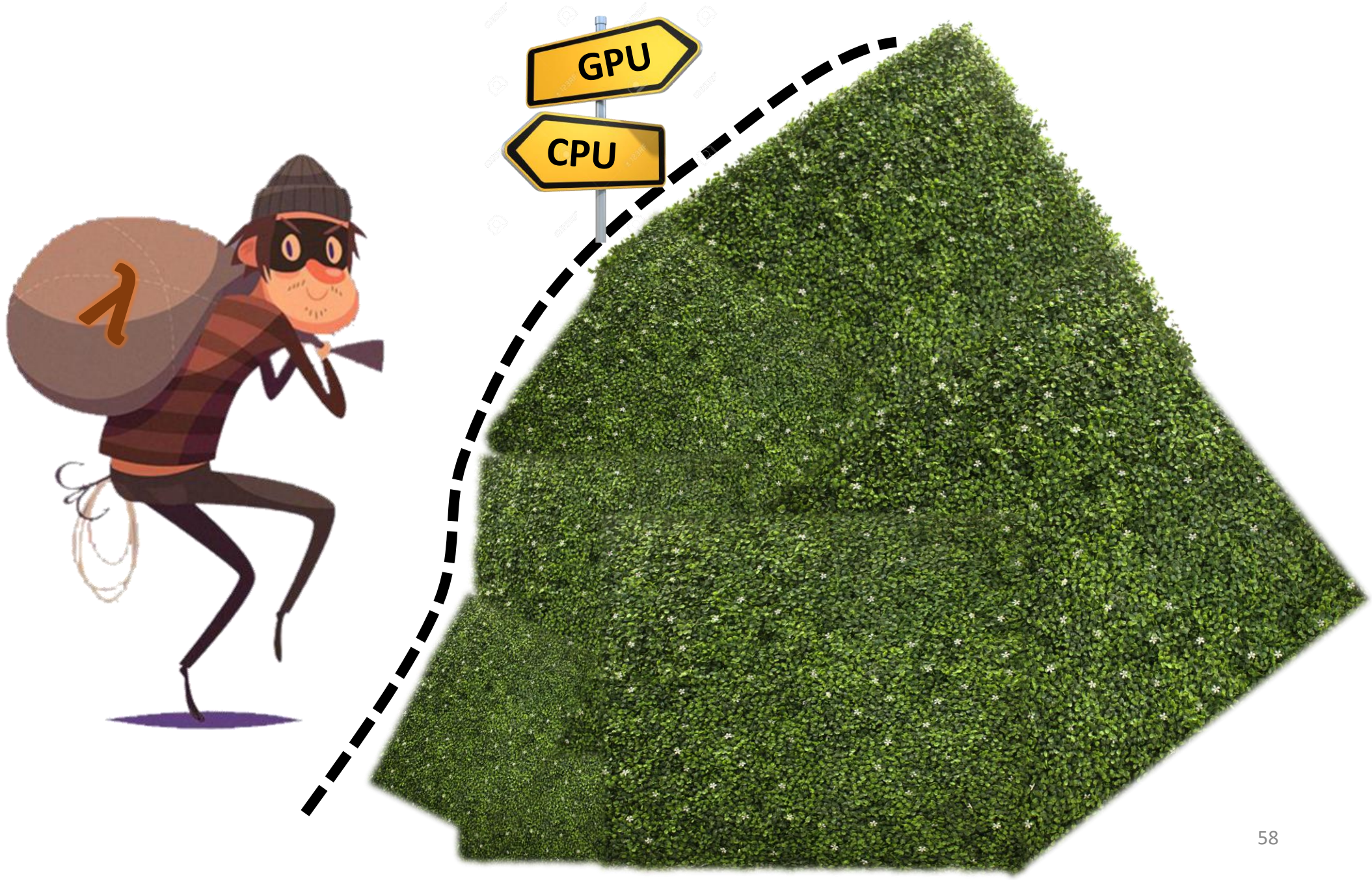
```
template<typename T, typename OP>
__global__ void addKernel(DevPtr<T> c, DevPtr<const T> a, DevPtr<const T> b,
                          OP op) {
  int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
  c[idx] = op(a[idx], b[idx]);
}
```

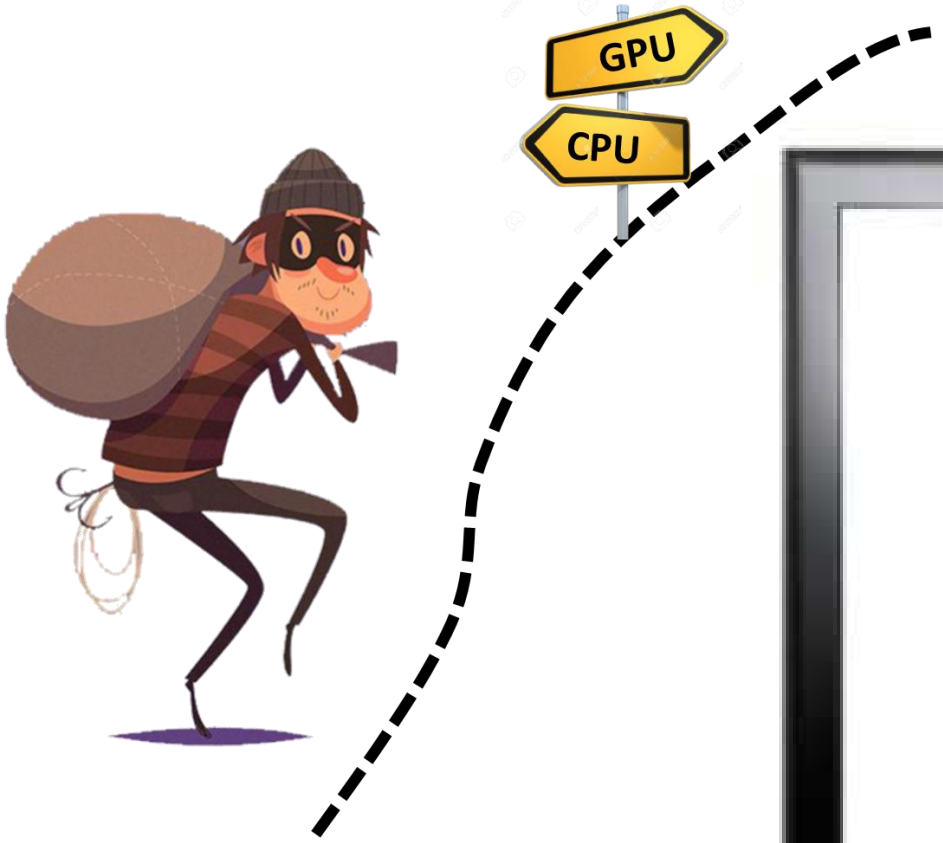
Host

```
int main() {
  //…
  auto op = [] __device__ (auto a, auto b){ return a + b; };
  addKernel<<<blocks, 32>>>(cDev, aDev, bDev, op);
  //…
}
```

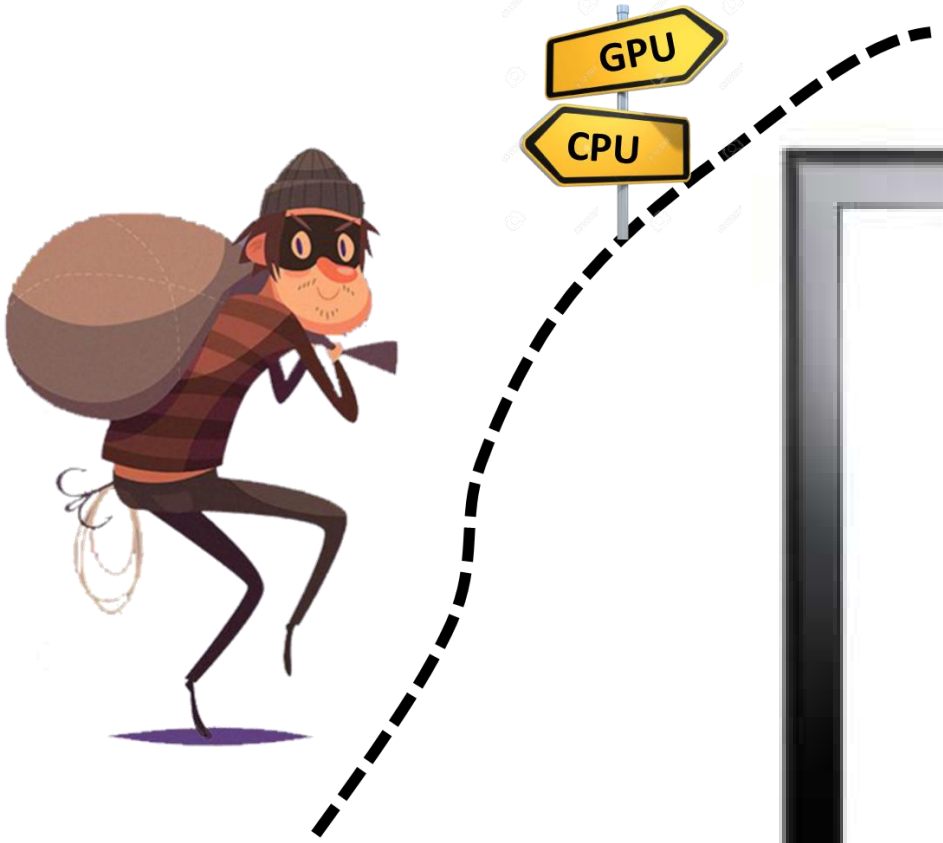
Note the *__device__* keyword

Requires *--expt-extended-lambda* compilation flag

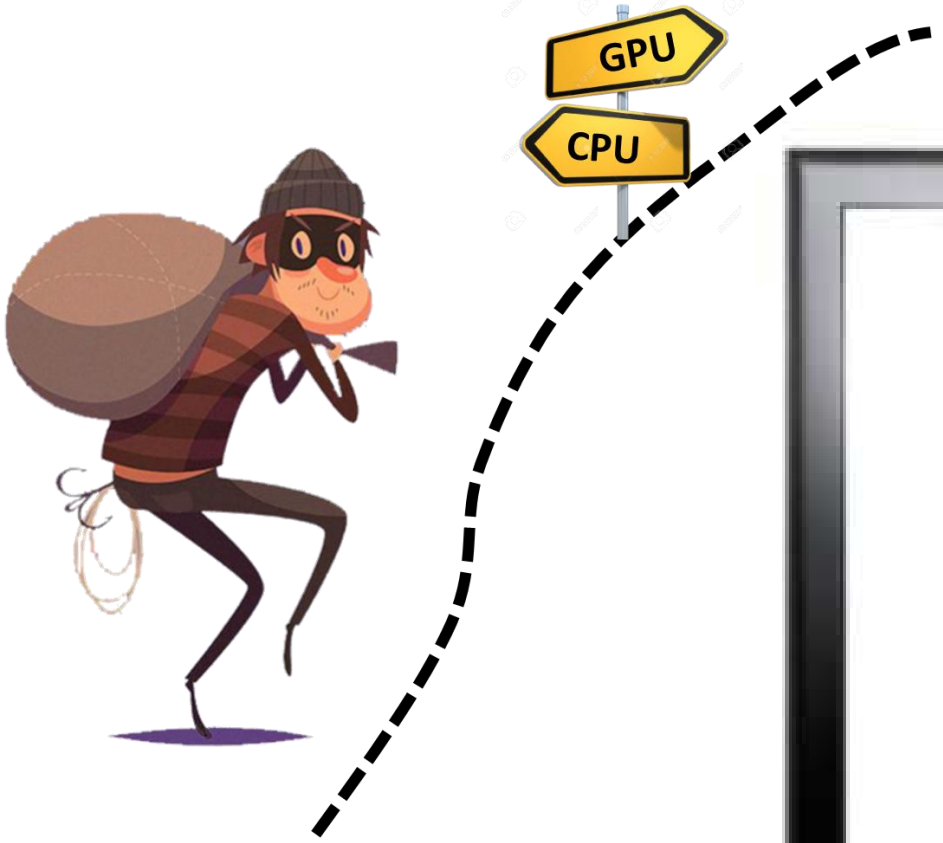

GPU
CPU
λ



Capture by reference!!



Capture by reference!!


GPU
CPU


this pointer!!

```
struct OP {
  int _i;
  explicit OP(int i) : _i(i) {}

  template<typename TC, typename TAB, typename DIM>
  void apply(TC &cDev, TAB &aDev, TAB &bDev, DIM blocks) {
    auto op = [this] __device__ (auto a, auto b){ return a + b + _i; };
    addKernel<<<blocks, 32>>>(cDev, aDev, bDev, op);
  }
};

int main() {
  //…
  OP op{42};
  op.apply(cDev, aDev, bDev, blocks);
  //…
}
```

Host

Host

```
struct OP {
  int _i;
  explicit OP(int i) : _i(i) {}

  template<typename TC, typename TAB, typename DIM>
  void apply(TC &cDev, TAB &aDev, TAB &bDev, DIM blocks) {
    auto op = [*this] __device__ (auto a, auto b){ return a + b + _i; };
    addKernel<<<blocks, 32>>>(cDev, aDev, bDev, op);
  }
};
```

Host

```
int main() {
  //…
  OP op{42};
  op.apply(cDev, aDev, bDev, blocks);
  //…
}
```

Host

```
struct OP {
  int _i;
  explicit OP(int i) : _i(i) {}

  template<typename TC, typename TAB, typename DIM>
  void apply(TC &cDev, TAB &aDev, TAB &bDev, DIM blocks) {
    auto op = [*this] __device__ (auto a, auto b){ return a + b + _i; };
    addKernel<<<blocks, 32>>>(cDev, aDev, bDev, op);
  }
};
```

Host

```
int main() {
  //…
  OP op{42};
  op.apply(cDev, aDev, bDev, blocks);
  //…
}
```

Host

```
struct OP {
  int _i;
  explicit OP(int i) : _i(i) {}

  auto make_op() {
    return [*this] __device__ (auto a, auto b){ return a + b + _i; };
  }
};

int main() {
  //…
  OP op{42};
  ?????
  //…
}
```

Host

Host

```
struct OP {
  int _i;
  explicit OP(int i) : _i(i) {}

  auto make_op() {
    return [*this] __device__ (auto a, auto b){ return a + b + _i; };
  }
};
```

Host

```
int main() {
  //…
  OP op{42};
  addKernel<<<blocks, 32>>>(cDev, aDev, bDev, op.make_op());
  //…
}
```

Host

Attempt 1

error : The enclosing parent function (“make_op") for an extended __device__ lambda must not have deduced return type

```
struct OP {
  int _i;
  explicit OP(int i) : _i(i) {}

  std::function<int(int, int)> make_op() {
    return [*this] __device__ (auto a, auto b){ return a + b + _i; };
  }
};

int main() {
  //…
  OP op{42};
  addKernel<<<blocks, 32>>>(cDev, aDev, bDev, op.make_op());
  //…
}
```

Host

Host

Attempt 2

error : calling a __host__ function("std::_Func_class<int > ::operator () const") from a __global__ function…

```
#include <nvfunctional>
```

```
struct OP {
  int _i;
  explicit OP(int i) : _i(i) {}

  nvstd::function<int(int, int)> make_op() {
     return [*this] __device__ (auto a, auto b){ return a + b + _i; };
  }
};

int main() {
  //…
  OP op{42};
  addKernel<<<blocks, 32>>>(cDev, aDev, bDev, op.make_op());
  //…
}
```

Host

Host

Attempt 3

Compiles but fails at runtime

… cannot be passed from host code to device code (and vice versa) at run time …

Device

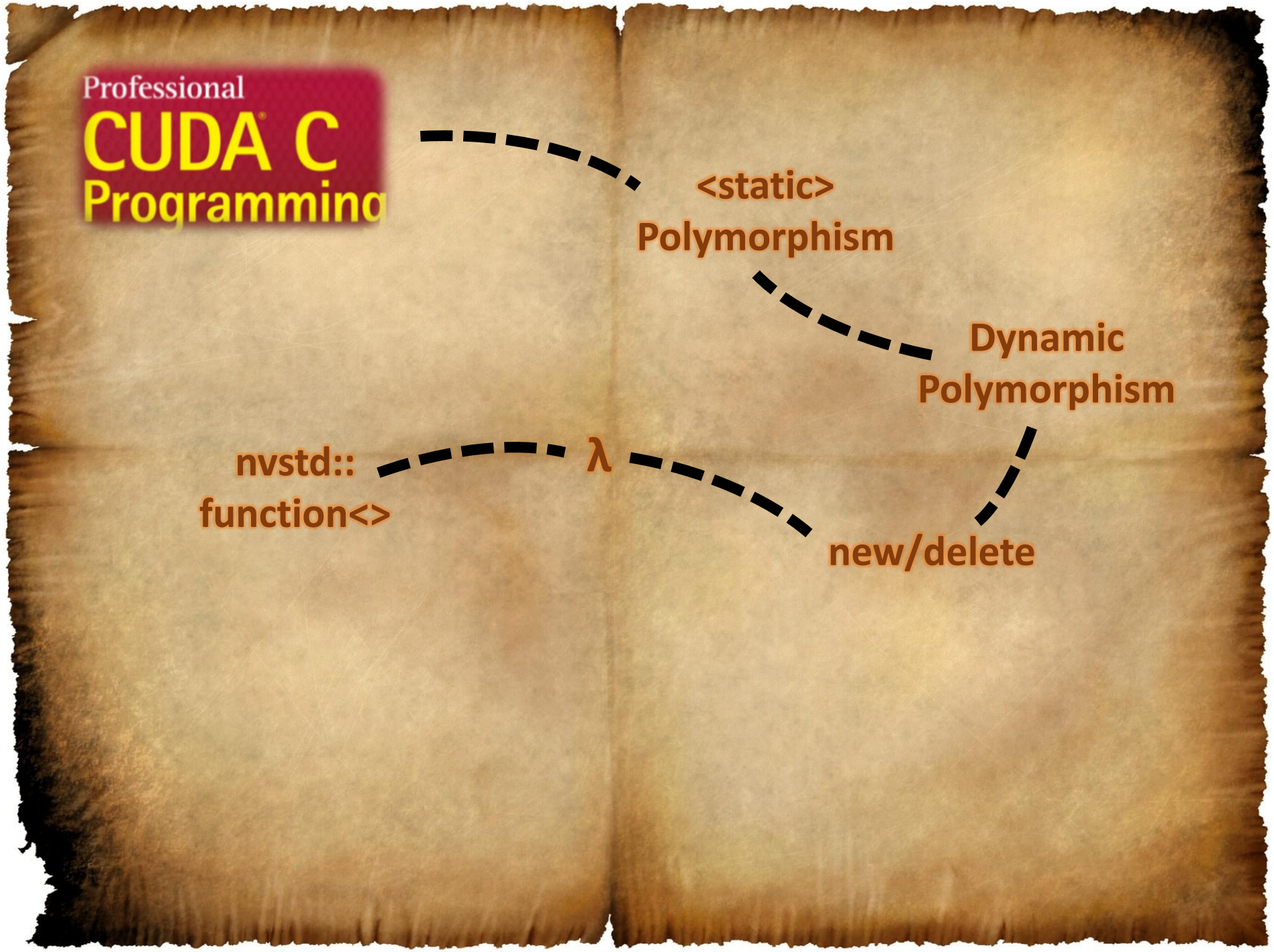
```
struct OP {
  int _i;
  explicit OP(int i) : _i(i) {}

  nvstd::function<int(int, int)> __device__ __host__ make_op() {
    return [this] (auto a, auto b){ return a + b + _i; };
  }
};
```

Pass the whole object to kernel, create the function using **make_op** in the kernel

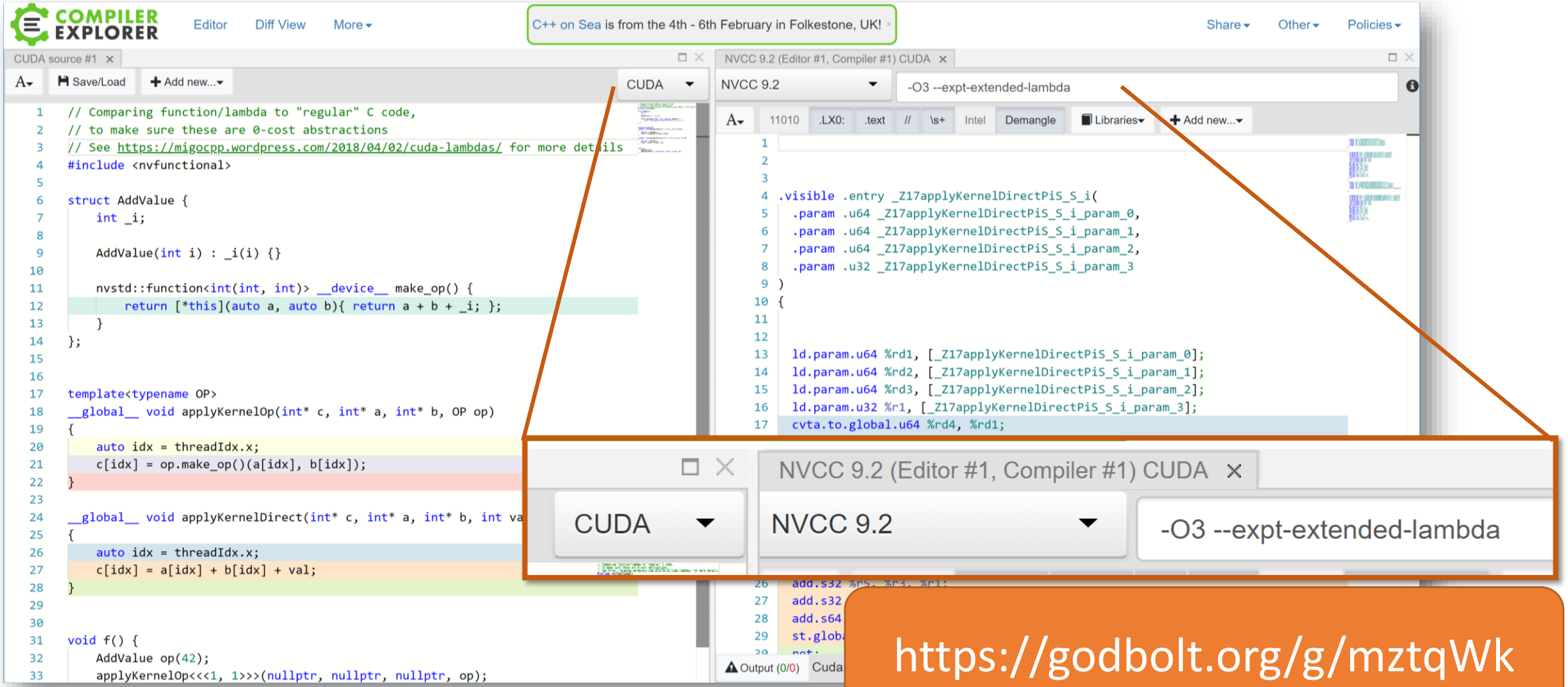
Host

```
int main() {
  //…
  OP op{42};
  addKernel<<<blocks, 32>>>(cDev, aDev, bDev, op);
  //…
}
```


Professional
CUDA C
Programming
<static>
Polymorphism
Dynamic
Polymorphism
nvstd::
function<>
λ
new/delete

But are these really Zero-Overhead (runtime) abstractions?

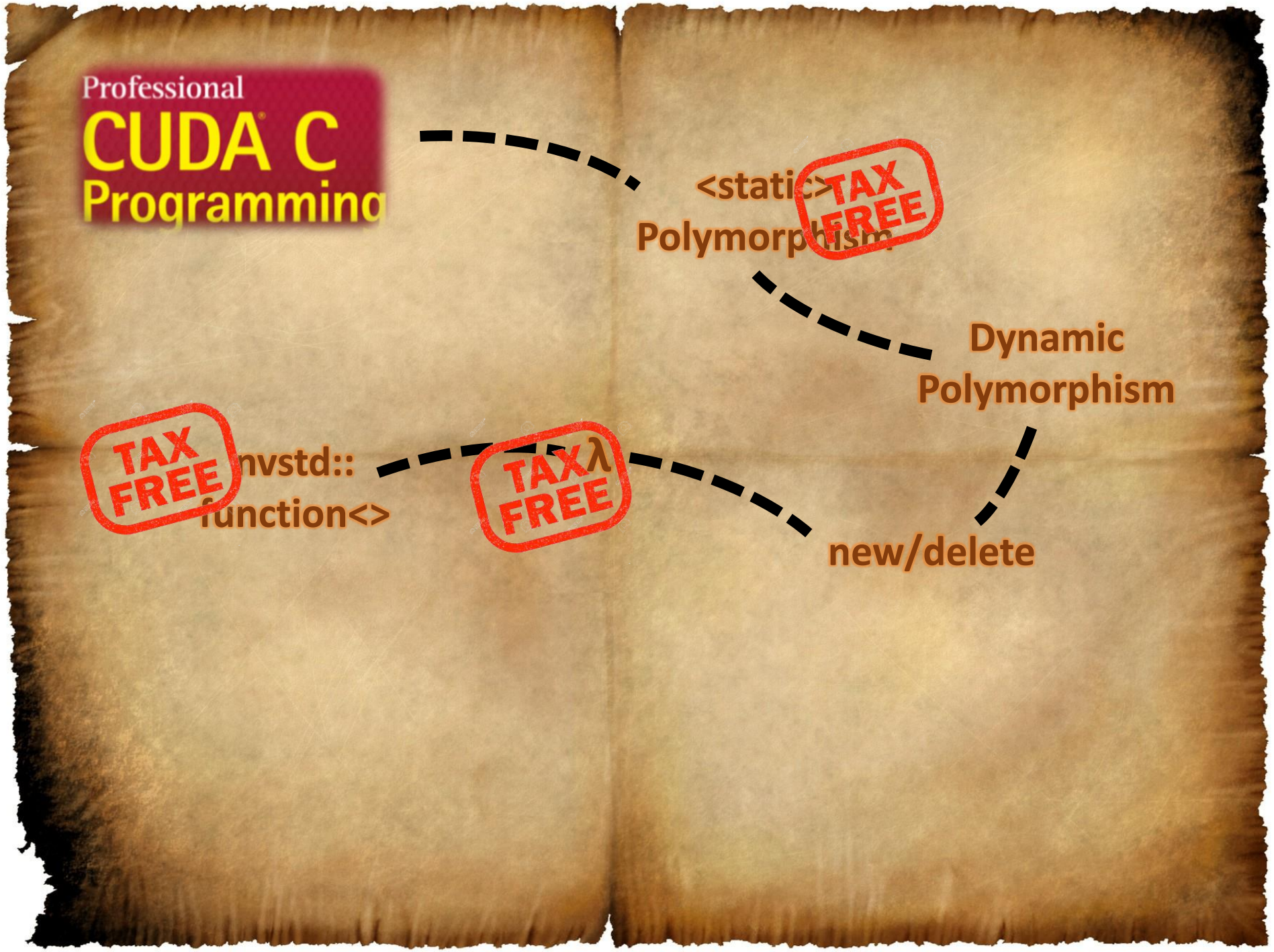
Godbolting CUDA



cuobjdump

```
C:\cuda\Release> cuobjdump lambda.cu.obj –sass
…
Function : _Z17applyKernelDirectN7cudacpp12DeviceVectorIiEES1_S1_i
.headerflags    @"EF_CUDA_SM30 EF_CUDA_PTX_SM(EF_CUDA_SM30)“

 /*0008*/   MOV R1, c[0x0][0x44];                    /* 0x2800400110005de4 */
 /*0010*/   S2R R0, SR_TID.X;                        /* 0x2c00000084001c04 */
 /*0018*/   MOV32I R7, 0x4;                          /* 0x180000001001dde2 */
 /*0020*/   ISCADD R2.CC, R0, c[0x0][0x150], 0x2;    /* 0x4001400540009c43 */
…
```


Professional
CUDA C
Programming
<static>
Polymorphism
TAX FREE
Dynamic
Polymorphism
TAX FREE
nvstd::
function<>
TAX FREE
λ
new/delete

Professional
CUDA C
Programming

<static>
Polymorphism

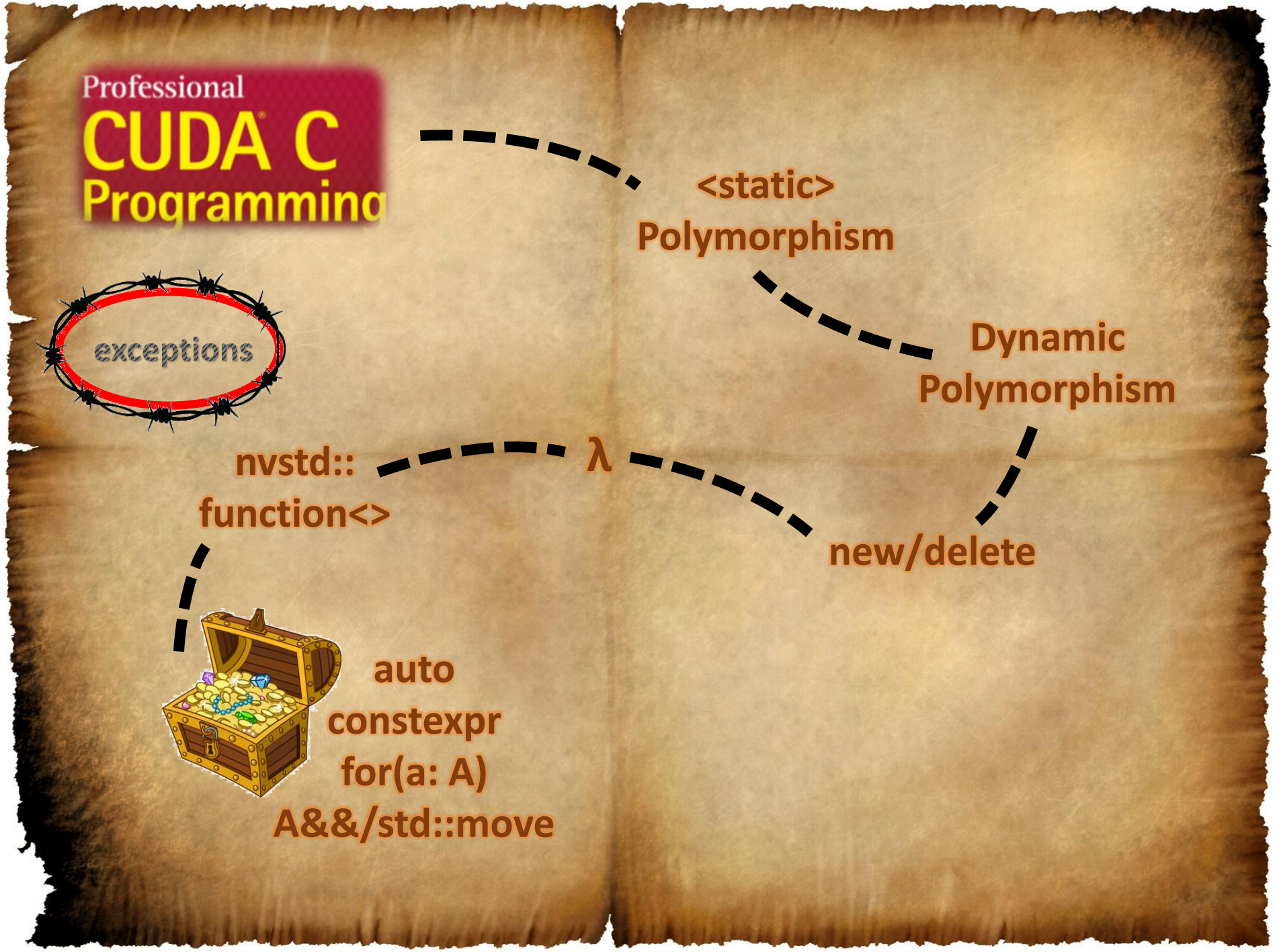
Dynamic
Polymorphism

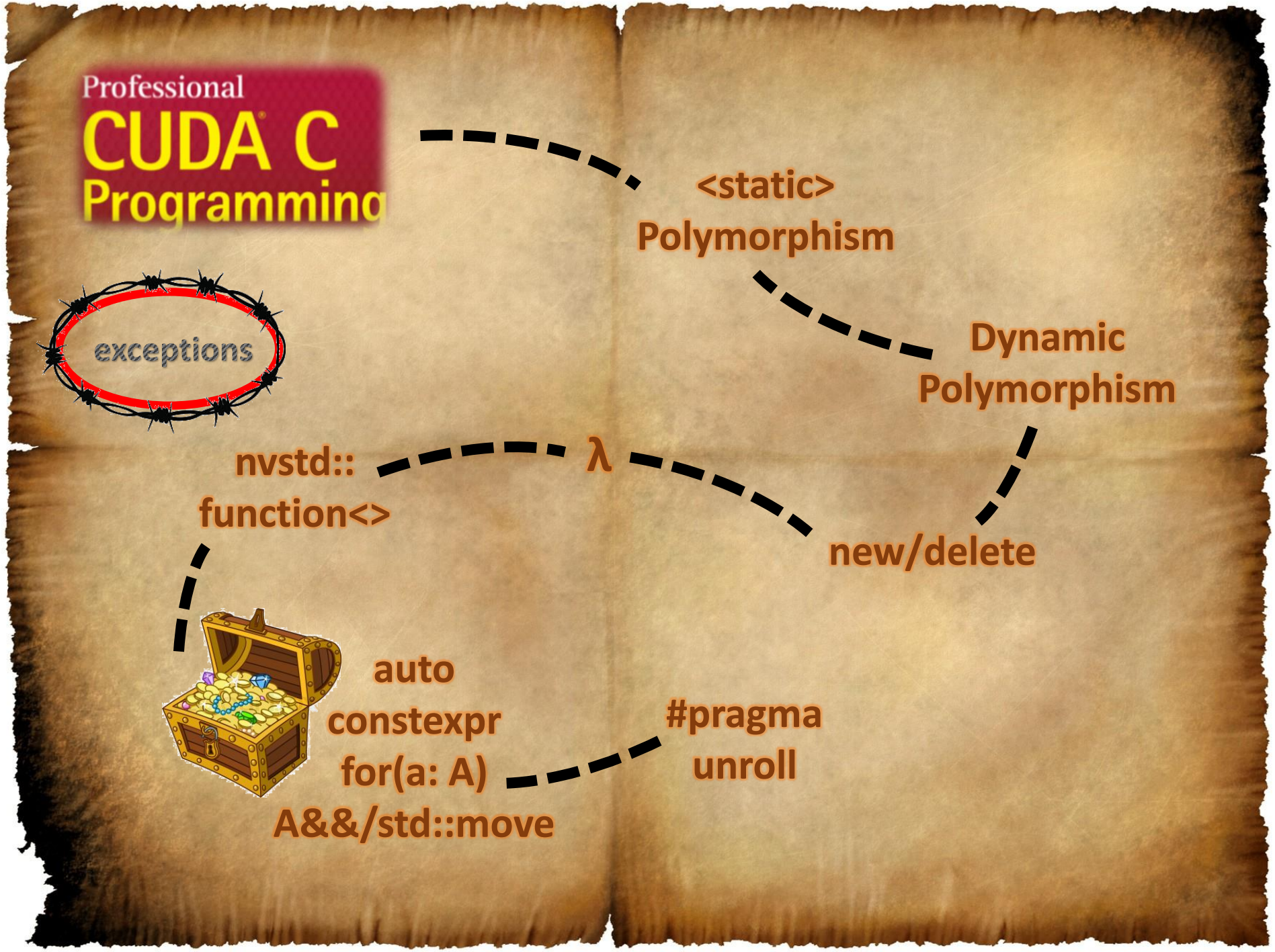
nvstd::
function<>

λ

new/delete

auto
constexpr
for(a: A)
A&&/std::move


Professional
CUDA C
Programming
exceptions
<static>
Polymorphism
Dynamic
Polymorphism
nvstd::
function<>
λ
new/delete
auto
constexpr
for(a: A)
A&&/std::move


Professional
CUDA C
Programming
exceptions
<static>
Polymorphism
Dynamic
Polymorphism
nvstd::
function<>
λ
new/delete
auto
constexpr
for(a: A)
A&&/std::move
#pragma
unroll

Device

```
template<typename T>
__device__ void myKernel(T *in, size_t length) {

  for (auto i = 0; i < length; ++i)
    // ...
}
```

Host

```
myKernel    <<<...>>>(in, length);
```

Device

```
template<typename T>
__device__ void myKernel(T *in, size_t length) {
  #pragma unroll ???
  for (auto i = 0; i < length; ++i)
    // ...
}
```

Host

```
myKernel    <<<...>>>(in, length);
```

Device

```
template<typename T>
__device__ void myKernel(T *in, size_t length) {
  #pragma unroll 10
  for (auto i = 0; i < length; ++i)
    // ...
}
```

Host

```
myKernel    <<<...>>>(in, length);
```

```
constexpr __host__ __device__ int mymax(int x, int y) {return …}

template<int unrollFactor, typename T>
__device__ void myKernel(T *in, size_t length) {
  #pragma unroll mymax(unrollFactor, 32)
  for (auto i = 0; i < length; ++i)
    // ...
}
```

Device

```
myKernel<64><<<...>>>(in, length);
```

Host

CUDA Runtime Compilation

- Runtime tuning of compilation flags (*architecture* etc.)
- **Runtime selection of template parameters**

Device

```
template<int LAYERS, typename T>
__global__ void process(T *data) {
  #pragma unroll LAYERS
  //…
}
```

Host

```
void main() {
  int layers = /*…*/
  //…
  process<????><<<…>>>(data);
}
```

```
template<int LAYERS, typename T>
__global__ void process(T *data) {/*…*/}

void doProcess(int layers, int* data) {
  if (layers == 1) process<1><<<…>>>(data);
  if (layers == 2) process<2><<<…>>>(data);
  if //…
}

void main() {
  doProcess(layers, data);
}
```

Host

All the template instantiations are being compiled

Using Dynamic Compilation

```
// load the source file and headers into buffer
nvrtcCreateProgram   // from source
```

Using Dynamic Compilation

```
// load the source file and headers into buffer
nvrtcCreateProgram        // create from source
nvrtcAddNameExpression    // register the instantiation
```

Using Dynamic Compilation

```
// load the source file and headers into buffer
nvrtcCreateProgram       // create from source
nvrtcAddNameExpression   // register the instantiation


nvrtcAddNameExpression(prog, “process<10, int>”);
```

Using Dynamic Compilation

```
// load the source file and headers into buffer
nvrtcCreateProgram       // create from source
nvrtcAddNameExpression   // register the instantiation
```

```
thrust::device_vector<int> v;
nvrtcAddNameExpression(prog, ???);
   // class thrust::device_vector<int,
   //    class thrust::device_malloc_allocator<int> >
```

Using Dynamic Compilation

```
// load the source file and headers into buffer
nvrtcCreateProgram       // create from source
nvrtcAddNameExpression   // register the instantiation
```

```
thrust::device_vector<int> v;
std::string name;
nvrtcGetTypeName<decltype(v)>(&name);
nvrtcAddNameExpression(prog, /*build the string*/);
```

Using Dynamic Compilation

```
// load the source file and headers into buffer
nvrtcCreateProgram          // create from source
nvrtcAddNameExpression      // register the instantiation
   (nvrtcGetTypeName)       // build the inst string
```

Using Dynamic Compilation

```
// load the source file and headers into buffer
nvrtcCreateProgram          // create from source
nvrtcAddNameExpression      // register the instantiation
   (nvrtcGetTypeName)       // build the inst string
nvrtcCompileProgram         // compile to PTX format
```

Using Dynamic Compilation

```
// load the source file and headers into buffer
nvrtcCreateProgram          // create from source
nvrtcAddNameExpression      // register the instantiation
   (nvrtcGetTypeName)       // build the inst string
nvrtcCompileProgram         // compile to PTX format
cuModuleLoadDataEx          // loads binary to GPU
```

Using Dynamic Compilation

```
// load the source file and headers into buffer
nvrtcCreateProgram          // create from source
nvrtcAddNameExpression      // register the instantiation
   (nvrtcGetTypeName)       // build the inst string
nvrtcCompileProgram         // compile to PTX format
cuModuleLoadDataEx          // loads binary to GPU
cuModuleGetFunction         // finds the kernel
```

Using Dynamic Compilation

```
// load the source file and headers into buffer
nvrtcCreateProgram          // create from source
nvrtcAddNameExpression      // register the instantiation
   (nvrtcGetTypeName)       // build the inst string
nvrtcCompileProgram         // compile to PTX format
cuModuleLoadDataEx          // loads binary to GPU
cuModuleGetFunction         // finds the kernel
   (nvrtcGetLoweredName)// mangle C++ names
```

Using Dynamic Compilation

```
// load the source file and headers into buffer
nvrtcCreateProgram          // create from source
nvrtcAddNameExpression      // register the instantiation
   (nvrtcGetTypeName)       // build the inst string
nvrtcCompileProgram         // compile to PTX format
cuModuleLoadDataEx          // loads binary to GPU
cuModuleGetFunction         // finds the kernel
   (nvrtcGetLoweredName)    // mangle C++ names
cuLaunchKernel              // finally, LAUNCH!
```

- Template library, *inspired* by STL
- Data is represented as vectors: `device_vector`, `host_vector` + many algorithms provided on these vectors (`transform`, `sort`, …)
- `device_ptr` is very similar to `DevicePtr` as we discussed

Professional
CUDA C
Programming

<static>
Polymorphism

Dynamic
Polymorphism

nvstd::
function<>

λ

new/delete

auto
constexpr
for(a: A)
A&&/std::move

#pragma
unroll

Professional
CUDA C
Programming

<static>
Polymorphism

Dynamic
Polymorphism

nvstd::
function<>

λ

new/delete

auto
constexpr
for(a: A)
A&&/std::move

#pragma
unroll

Using C++ !!

@michael_gop

mgopshtein/cudacpp
(code examples)

https://**corecppil**.github.io/Meetups/

- New Compiler Features in CUDA 8
  https://devblogs.nvidia.com/new-compiler-features-cuda-8/
- Kokkos: C++ Programming model for HPC
  https://github.com/kokkos/kokkos