CUDA Kernels with C++

Michael Gopshtein

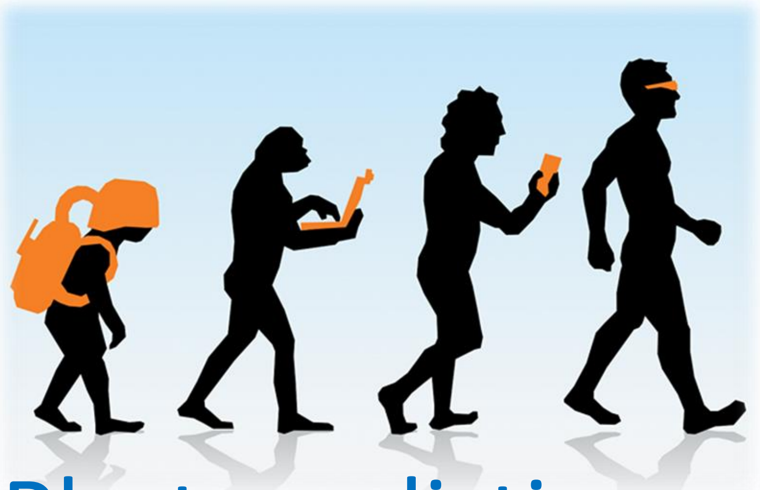
Core C++ @ TLV

Aug 2018

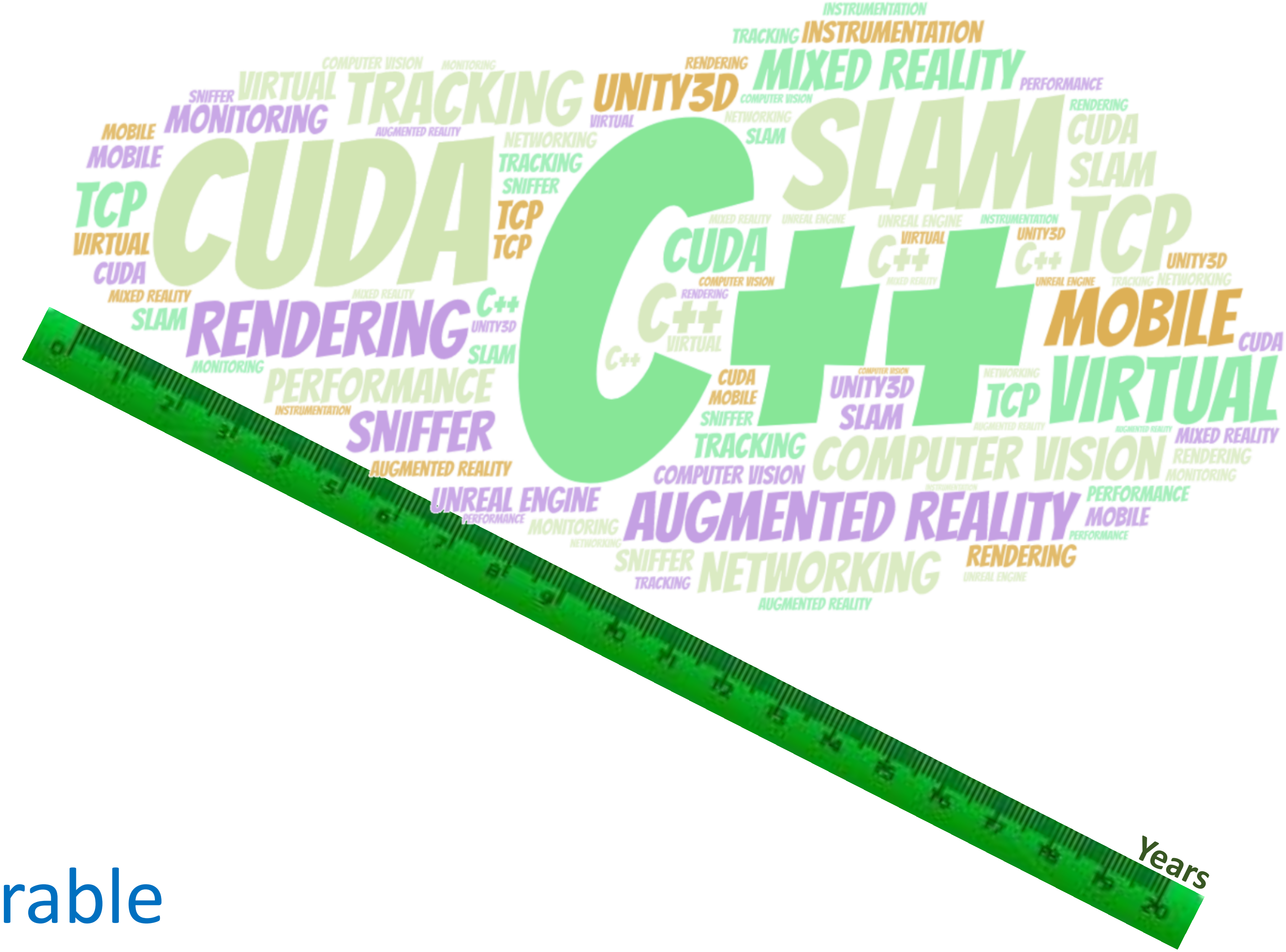
About me

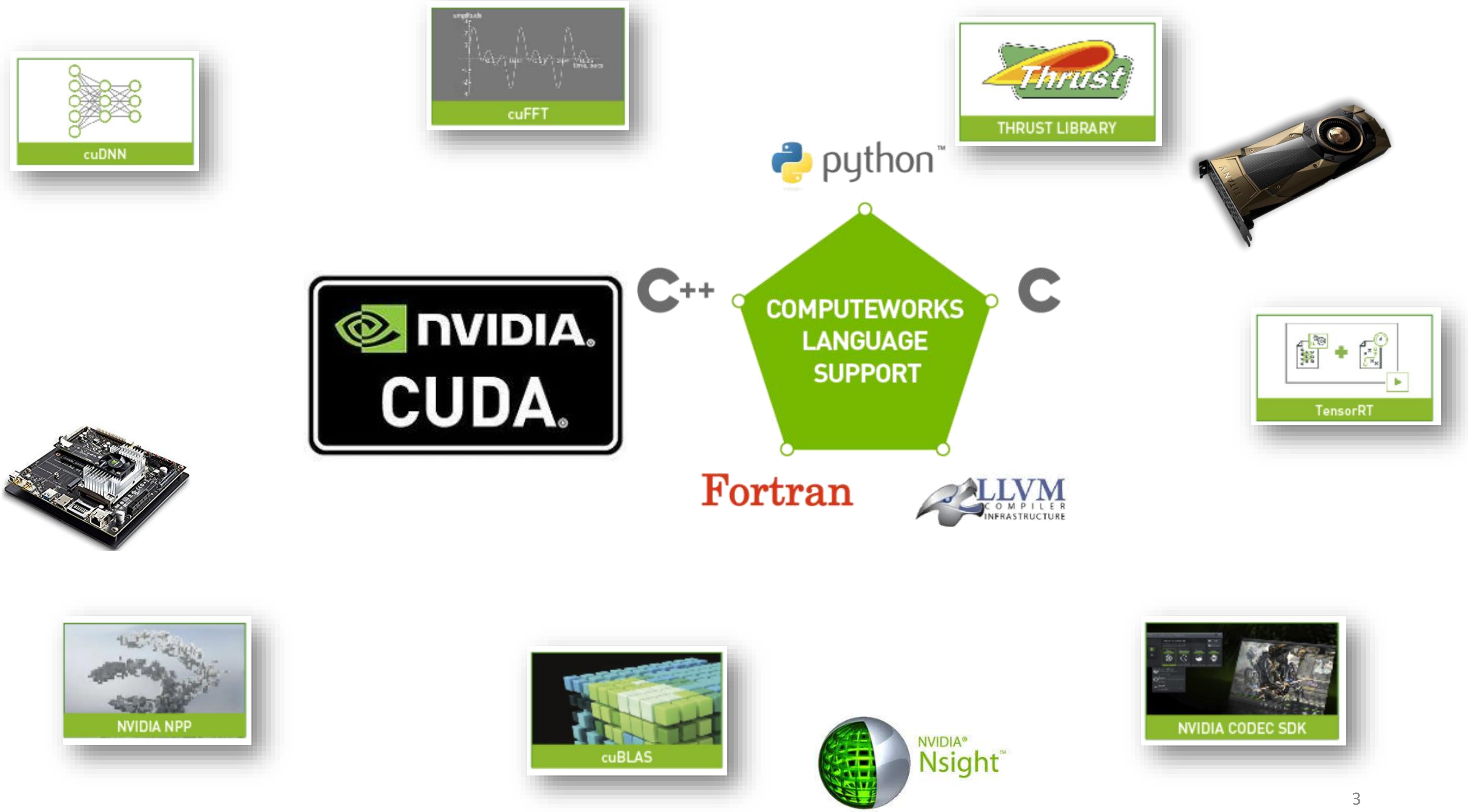
EYEWAY

ALL IS ILLUMINATED

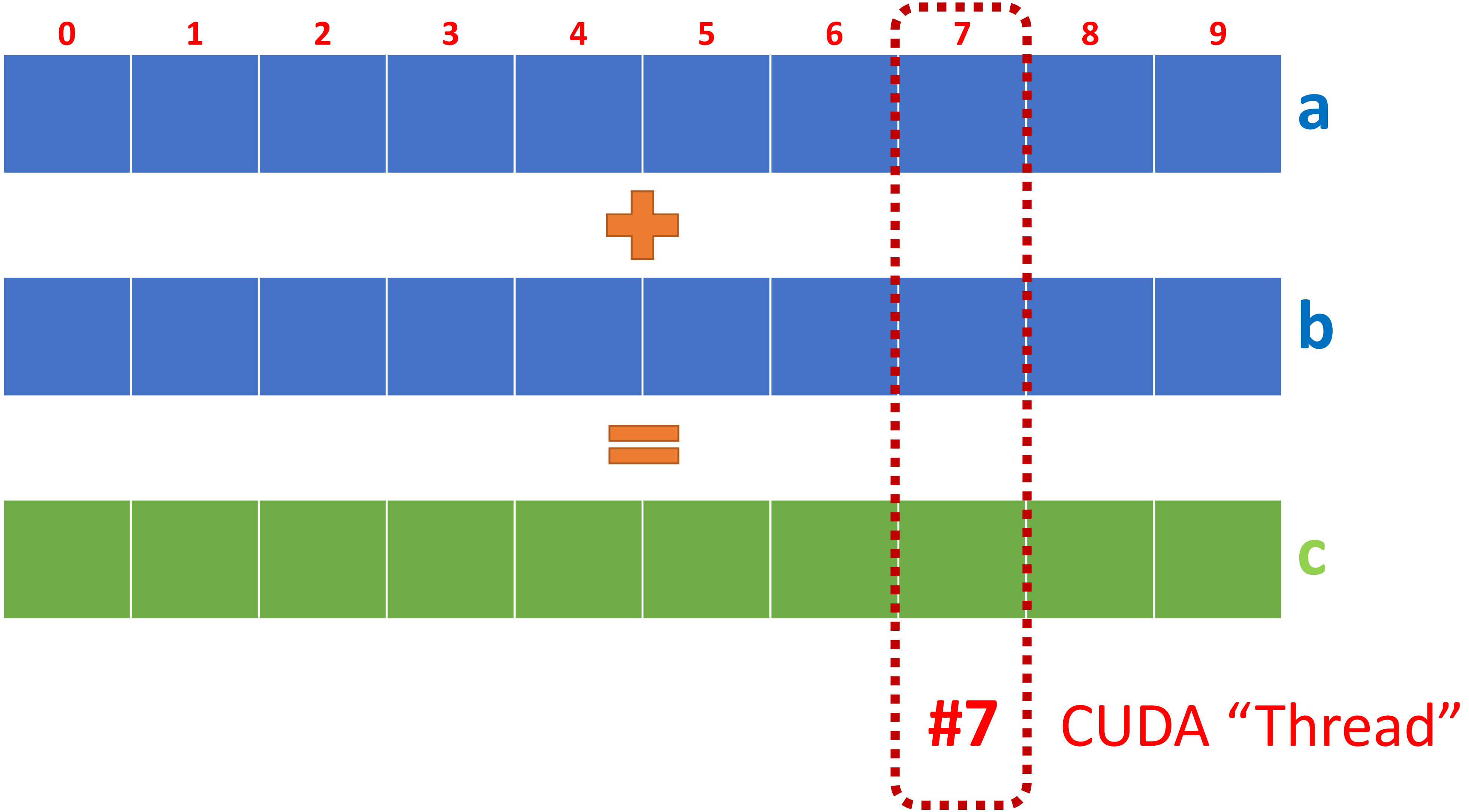


Photorealistic wearable Augmented Reality experience.




cuDNN
cuFFT
THRUST LIBRARY
Thrust
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

```
__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, dev_a, dev_b);
  //…
}
```

Kernel/Device Code

Host/CPU Code

Vector Addition in CUDA

```
__global__ void addKernel(int *c          int *a, const int *b) {
    int idx = (blockIdx.x * b       m.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?

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

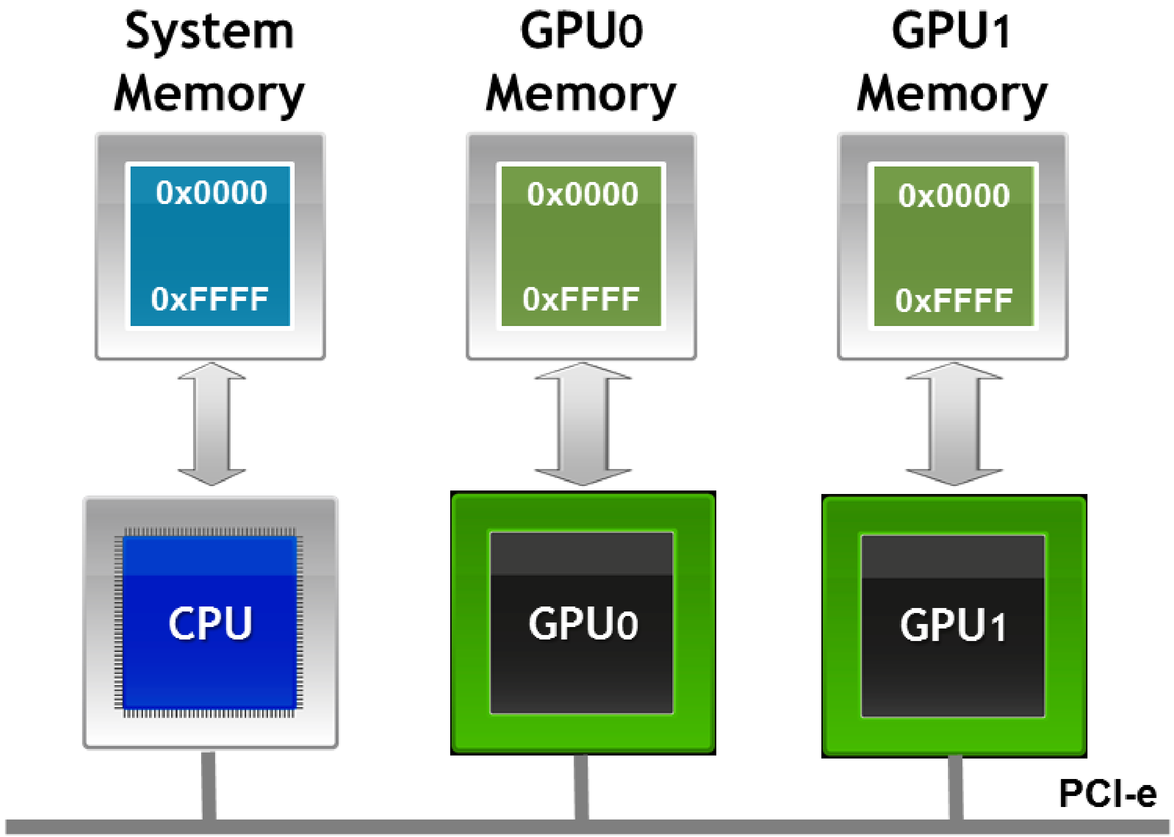
```
__global__ void addKernel(int *c, const int *a, const int *b) {
    int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
    c[idx] = a[idx] + b[idx];
}

__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!

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

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

```
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

```
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

```
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;

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

public:
  __host__ static DevicePtr FromRawDevicePtr(T *p) {
    return { p };
  }
//…
};

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

The constructor (**T***) is private

Explicit creation from raw **T***

Convenience global function

Simple usage

```
int main() {
  int *a = //… initialization of input vector
  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

```
int main() {
  unique_ptr<int[]> a = //… initialization of input vector
  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);
}
```

```
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>::FromRawDevicePtr(_p);
  }
};
```

<type_traits>

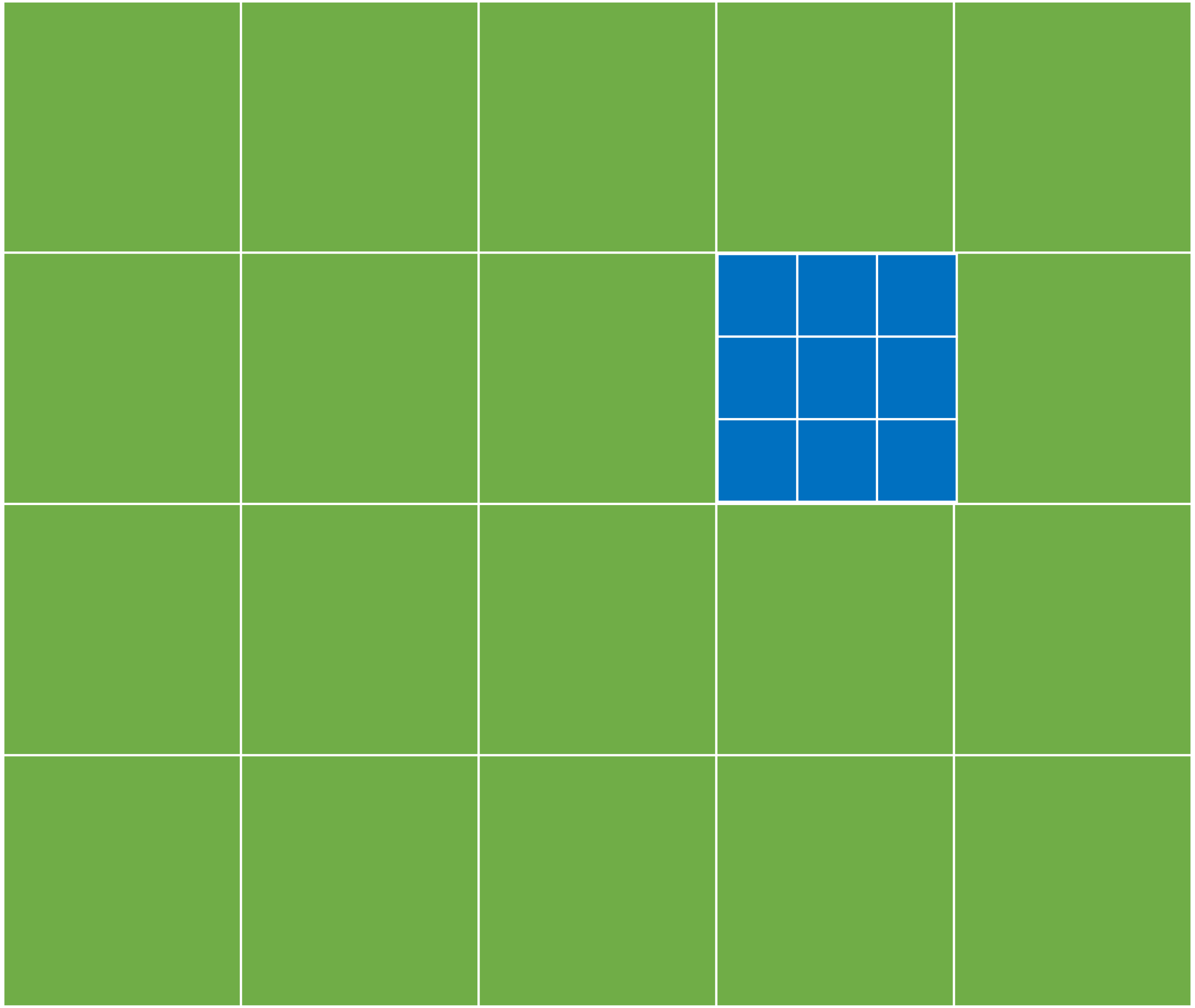
```
DevicePtr<int> a;
DevicePtr<const int> b = a; ✓
DevicePtr<int> c = b; ✗
DevicePtr<char> d = a; ✗
```

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

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

Let's look at the index

```
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;
  c[idx] = a[idx] + b[idx];
}
```

How to calculate the correct index?



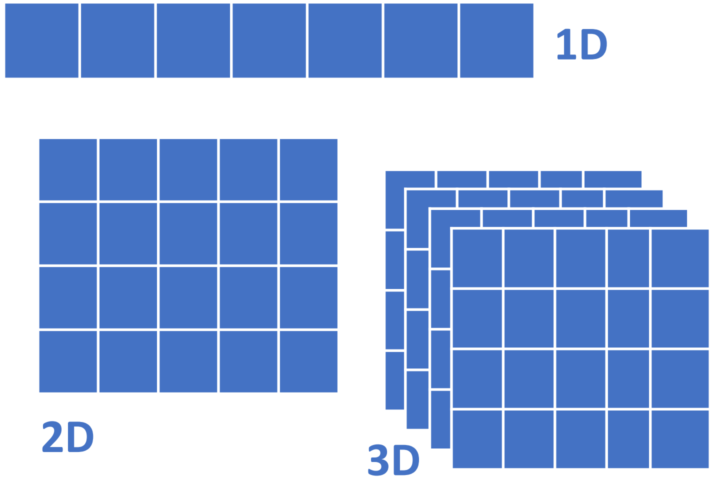
Kernel “**Threads**” are organized in **Blocks**.

Kernel is launched in a **Grid** of **Blocks**.

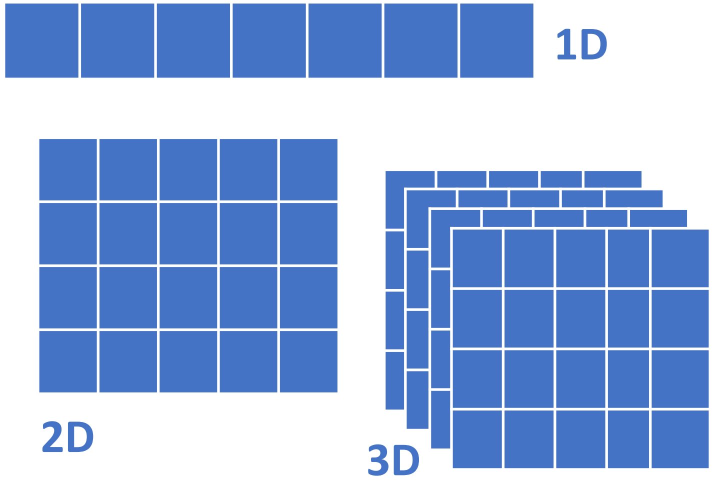
ID of a thread consists of
- Block ID
- Thread ID

Each ID can be 1/2/3-dimentional

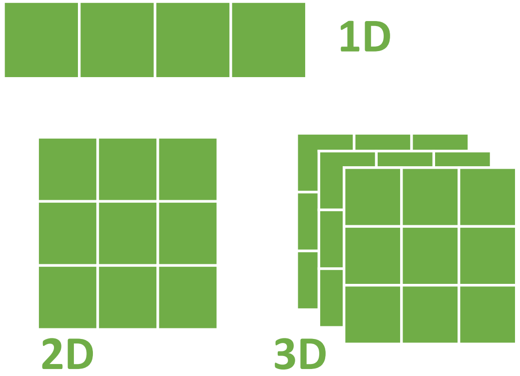
Original Data



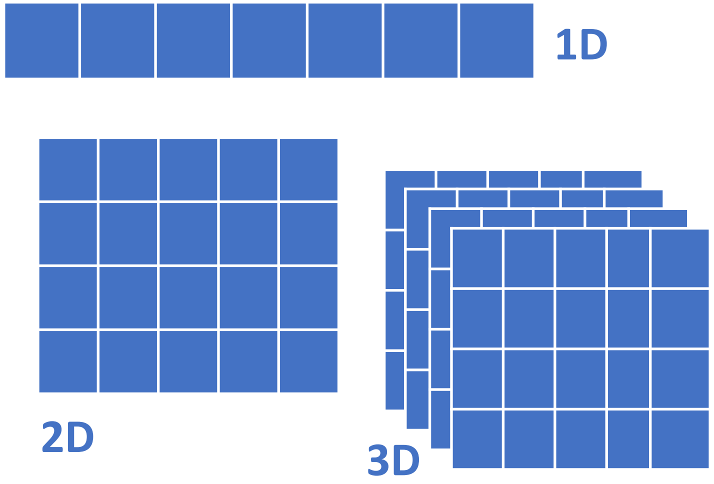
Original Data



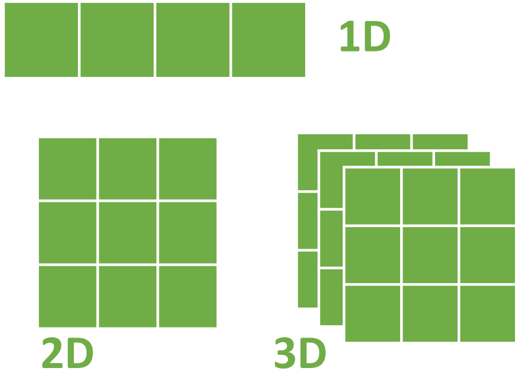
Block Size



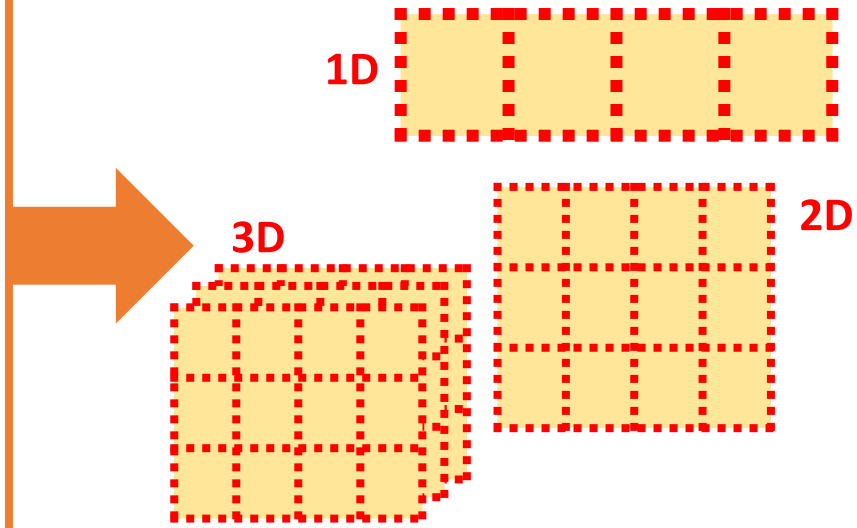
Original Data



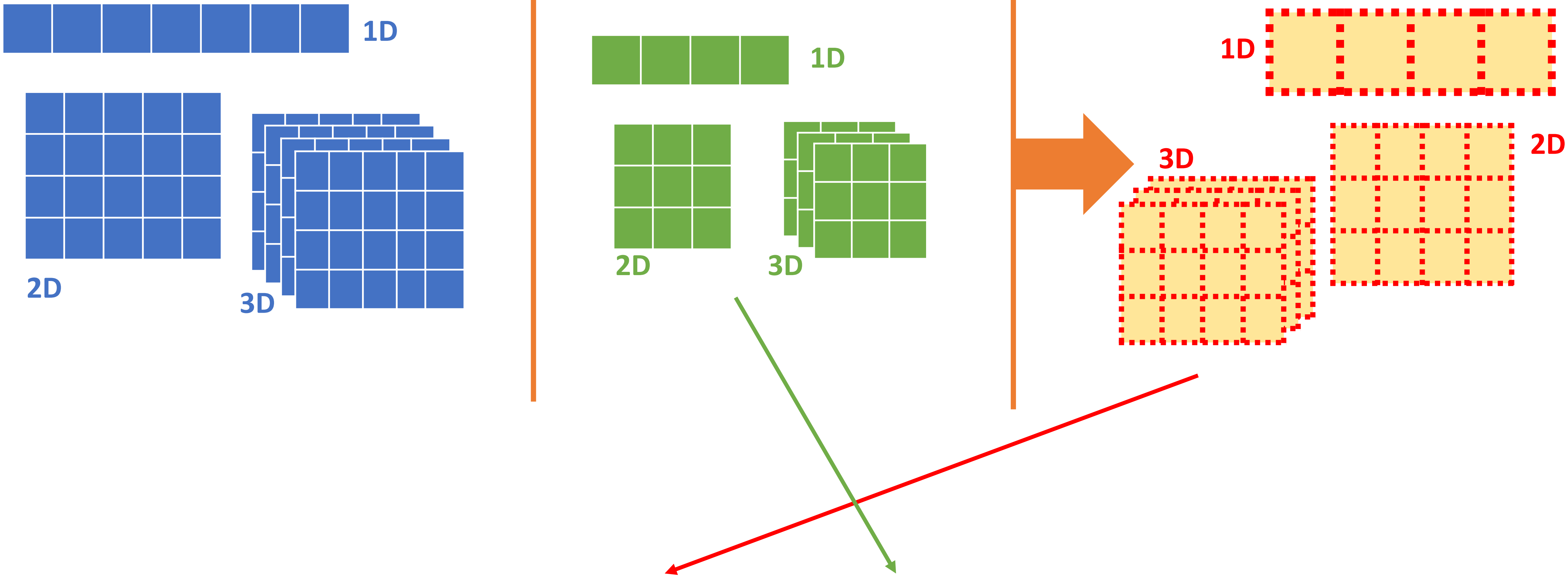
Block Size



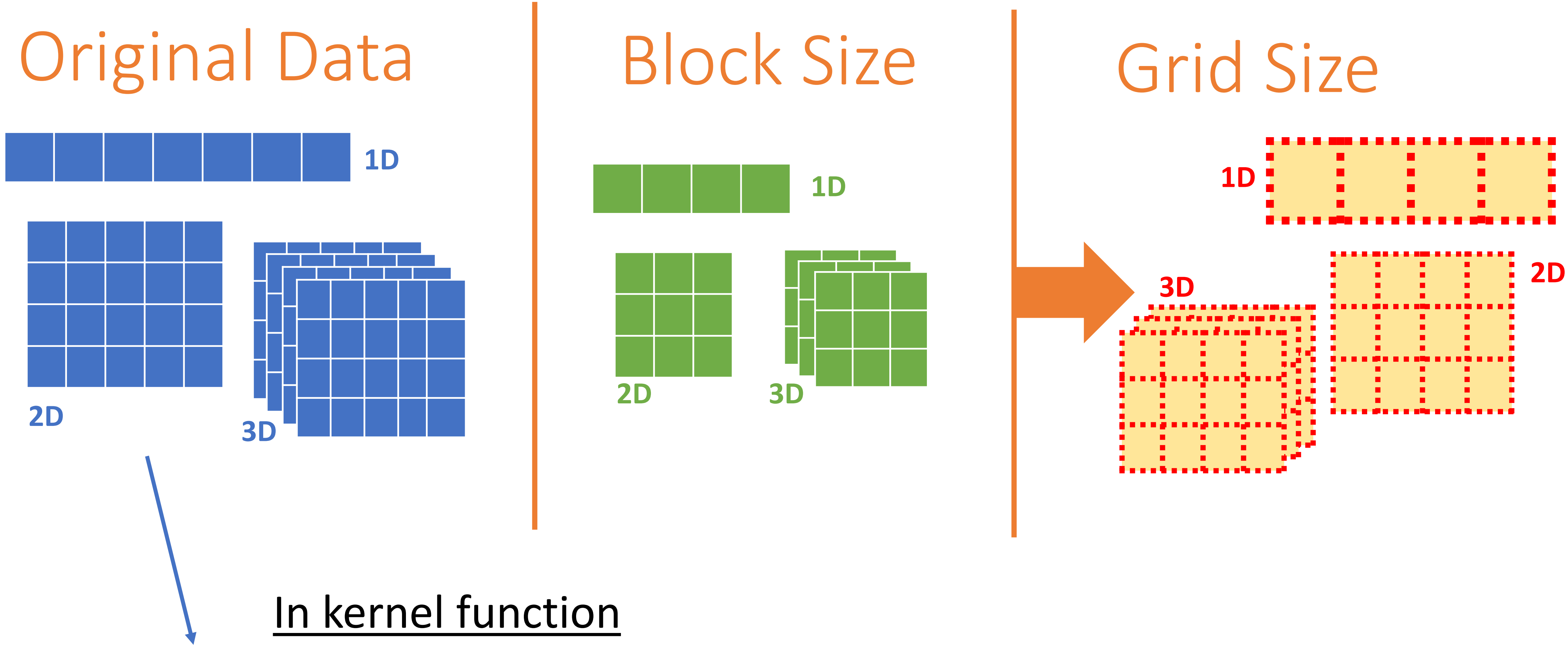
Grid Size



Original Data | Block Size | Grid Size



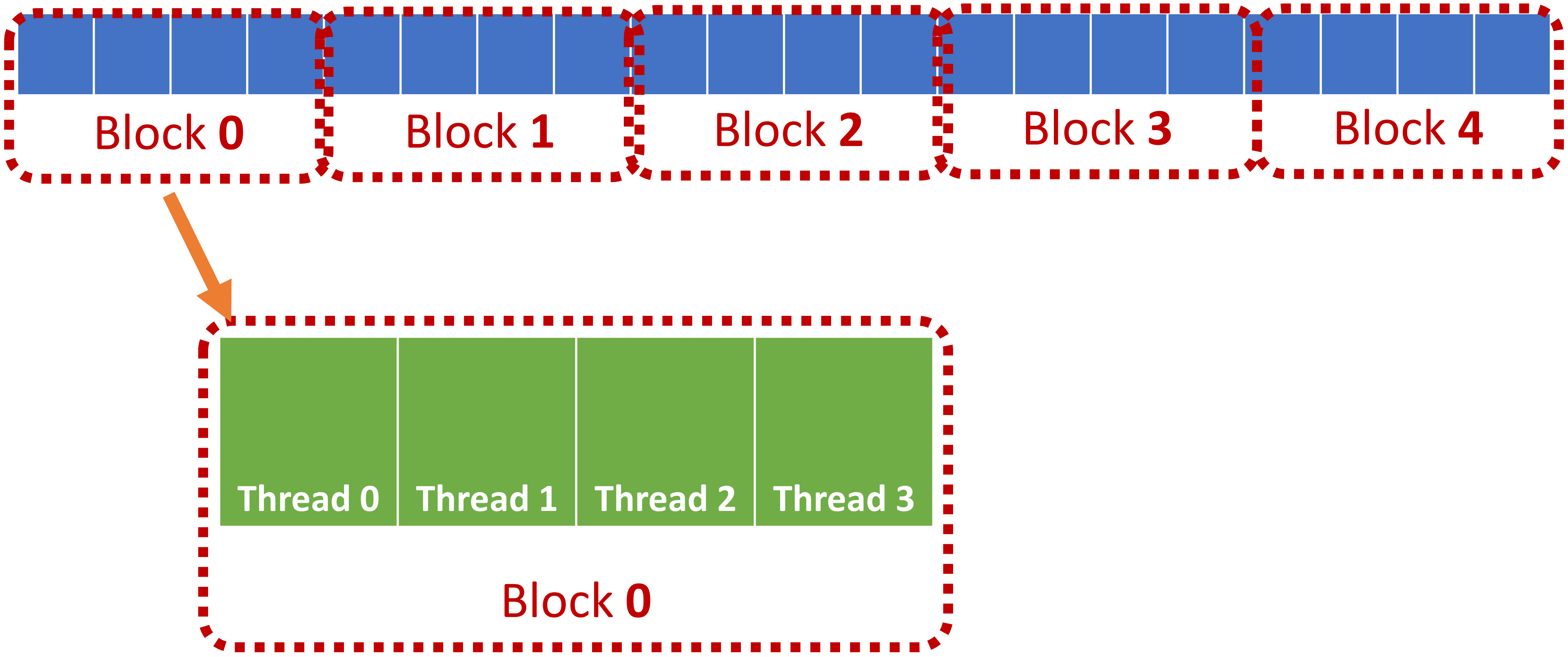

```
addKernel<<<blocks, threads>>>(cDev, aDev, bDev);
```



In kernel function

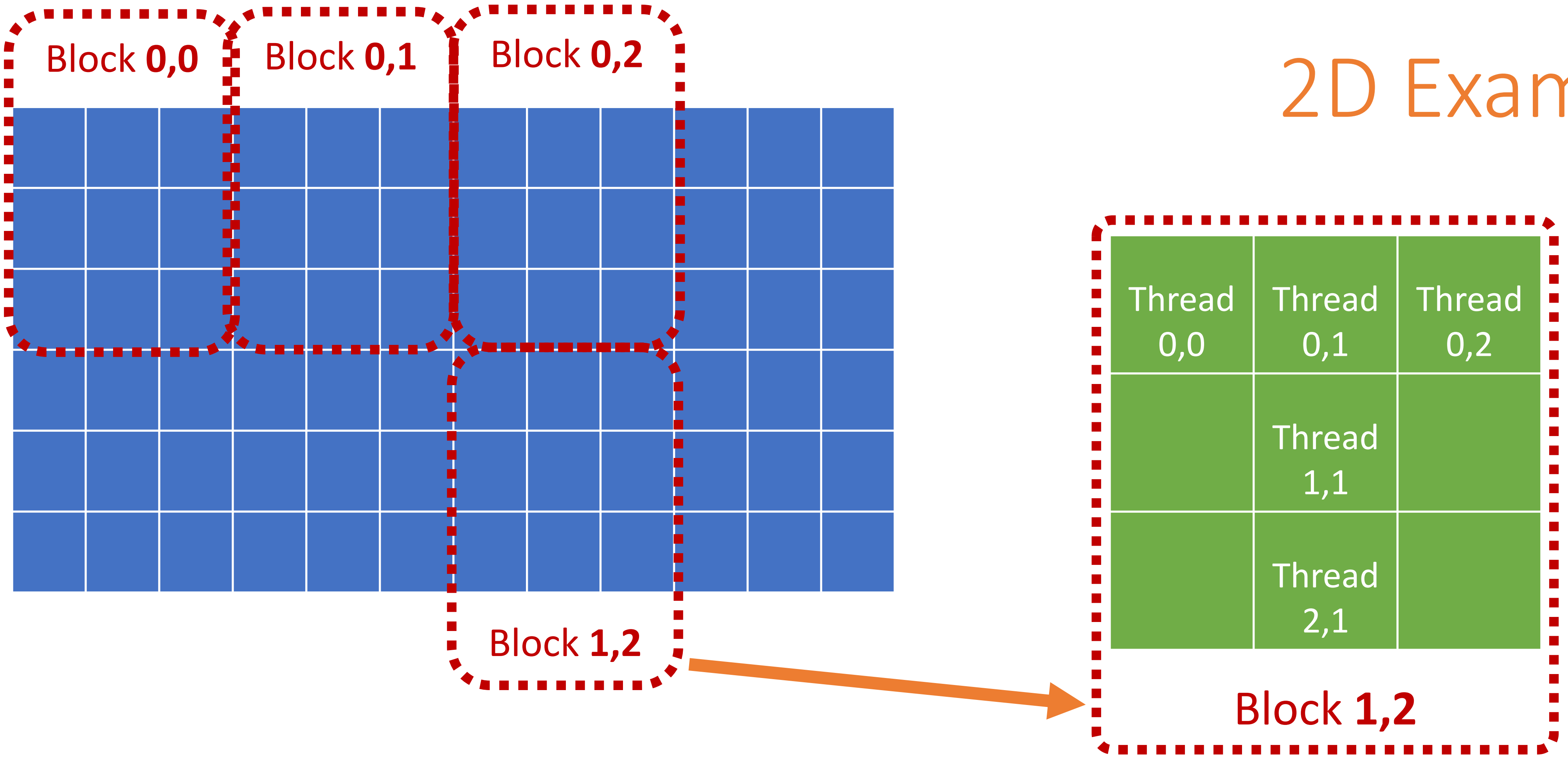
- Number of dimentions in the index
- Building the index based on `threadIdx`/`blockIdx`/`gridDim`/…

1D Example



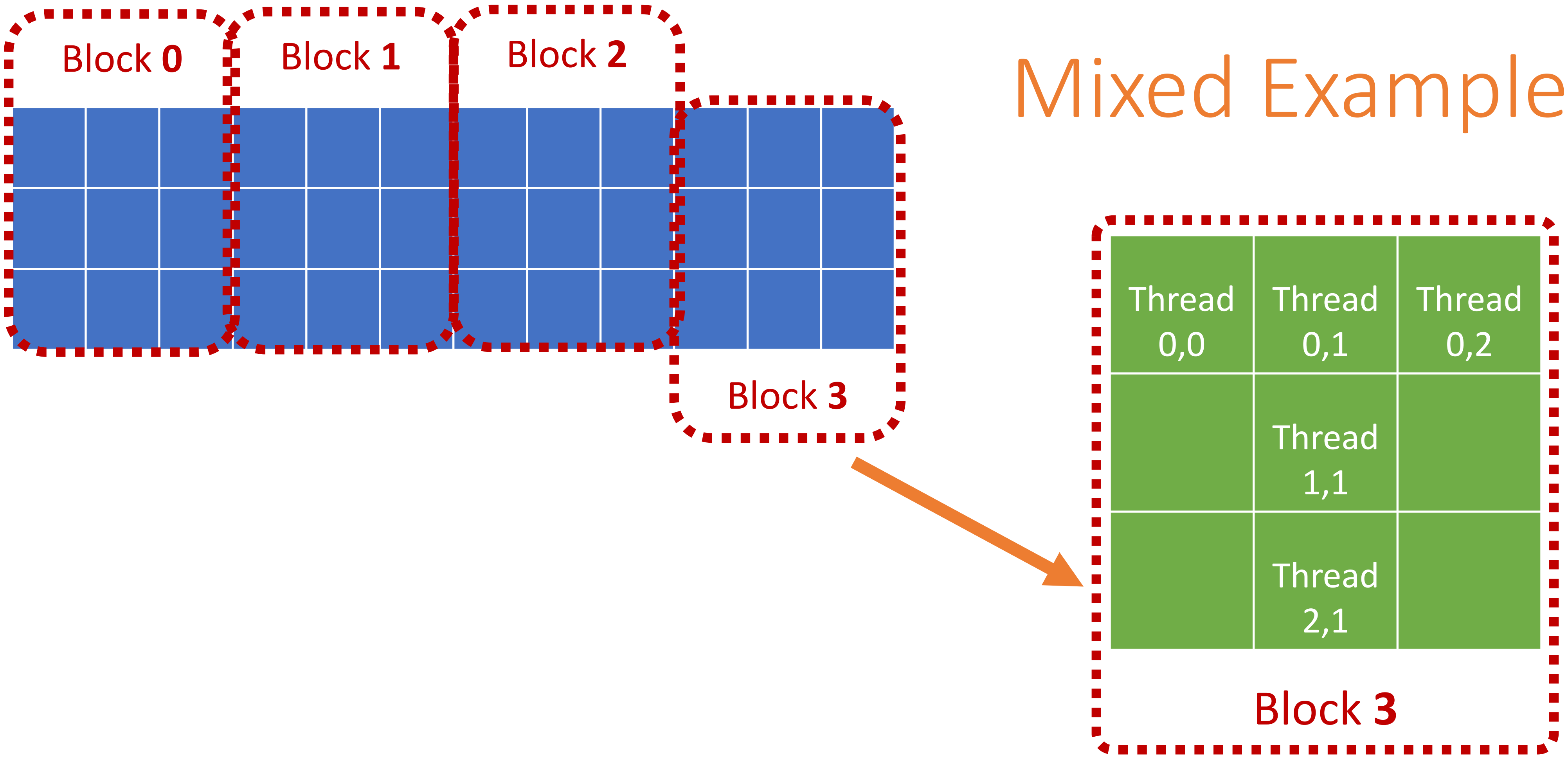

```
int myX = (blockIdx.x * blockDim.x) + threadIdx.x;
```

2D Example




```
int myX = (blockIdx.x * blockDim.x) + threadIdx.x;
int myY = (blockIdx.y * blockDim.y) + threadIdx.y;
```

Mixed Example




```
int myX = (blockIdx.x * blockDim.x) + threadIdx.x;
int myY = threadIdx.y;
```

No compile-time validation for dimensions

```
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;
  c[idx] = a[idx] + b[idx];
}

addKernel<<<dim3(10,10,1), 32>>>(cDev, aDev, bDev);
```

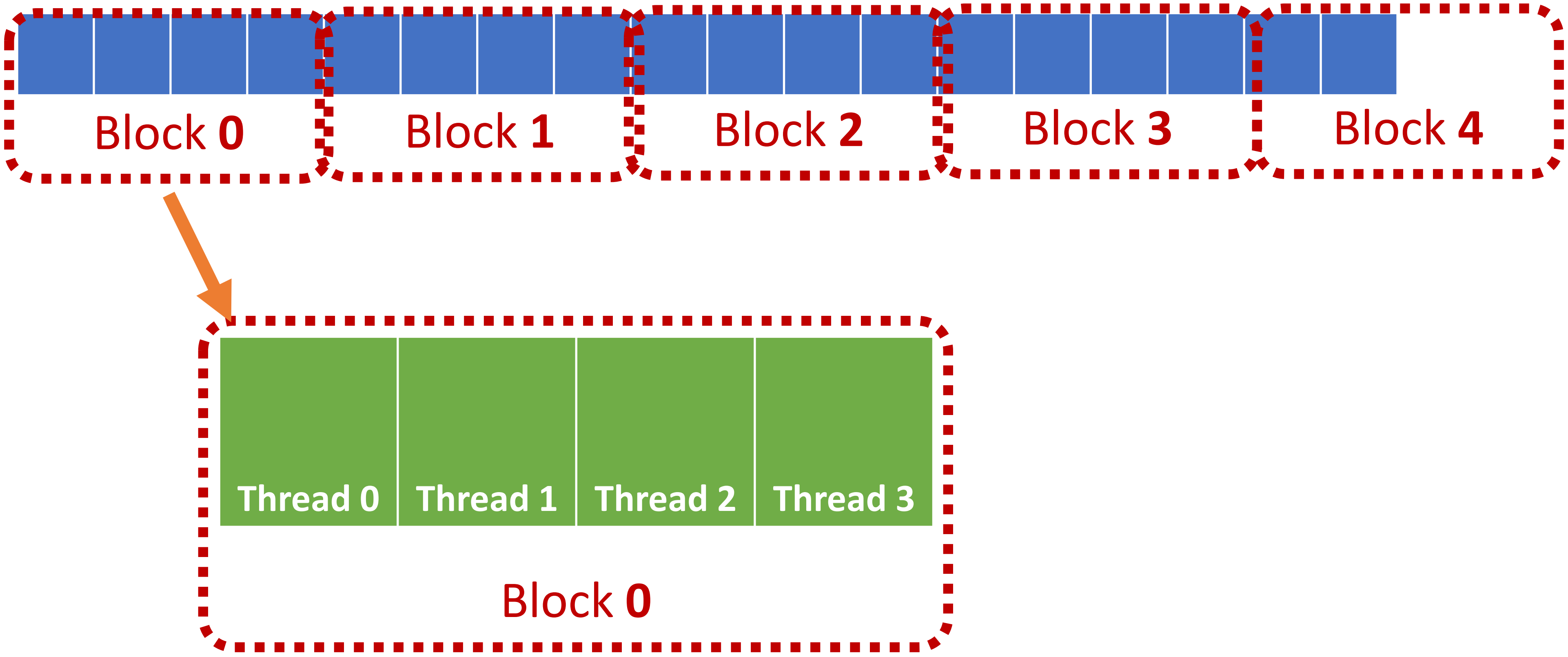
Assumes 1D grid

Uses 2D grid

correct index

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

addKernel<<<dim3(10,10,1), 32>>>(cDev, aDev, bDev);
```

1D Example – Out Of Bounds




```
int myX = (blockIdx.x * blockDim.x) + threadIdx.x;
```

```
template<int DIM_GRID, int DIM_BLOCK, int DIM_DATA>
struct GridInfo {
  Size<DIM_DATA> dataSz;

  __device__ Index<DIM_DATA> index() const;
  __device__ bool inRange() const;
};
```

Used in Kernel code

```
template<int DIM_GRID, int DIM_BLOCK, int DIM_DATA>
struct Grid {
  Size<DIM_GRID> blocks;
  Size<DIM_BLOCK> blockSz;
  Size<DIM_DATA> dataSz;

  auto info() const {
    return GridInfo<DIM_GRID, DIM_BLOCK, DIM_DATA>{ dataSz };
  }
};

template<int DIM_GRID, int DIM_BLOCK, int DIM_DATA>
static auto CreateGrid(const Size<DIM_BLOCK> &szBlock,
                       const Size<DIM_DATA> &szData);
```

Used in CPU code

Grid Info as template parameter

```
template<typename T, typename GRID_INFO>
__global__ void addKernel(DevPtr<T> c, DevPtr<const T> a,
                          DevPtr<const T> b, GRID_INFO info) {
  auto idx = info.index();
  if (info.inRange())
    c[idx] = a[idx] + b[idx];
}

Size<1> dataSz{ SIZE };
Size<1> blockSz{ 128 };
auto grid = CreateGrid<1>(dataSz, blockSz);
addKernel<<<grid.blocks, grid.blockSz>>>(cDev, aDev, bDev, grid.info());
```

GRID_INFO type *"knows"* all the dimensions

calculates the index,

can validate the range of the index

The **Grid** will calculate the number of blocks needed

<static>
Polymorphism

<static>
Polymorphism

Dynamic
Polymorphism

```cpp
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; }
};
```

```cpp
template<typename T>
__device__ void addKernelDo(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]);
}
template<typename T>
__global__ void addKernel(DevPtr<T> c, DevPtr<const T> a, DevPtr<const T> b) {
  addKernelVirtualDo(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…*

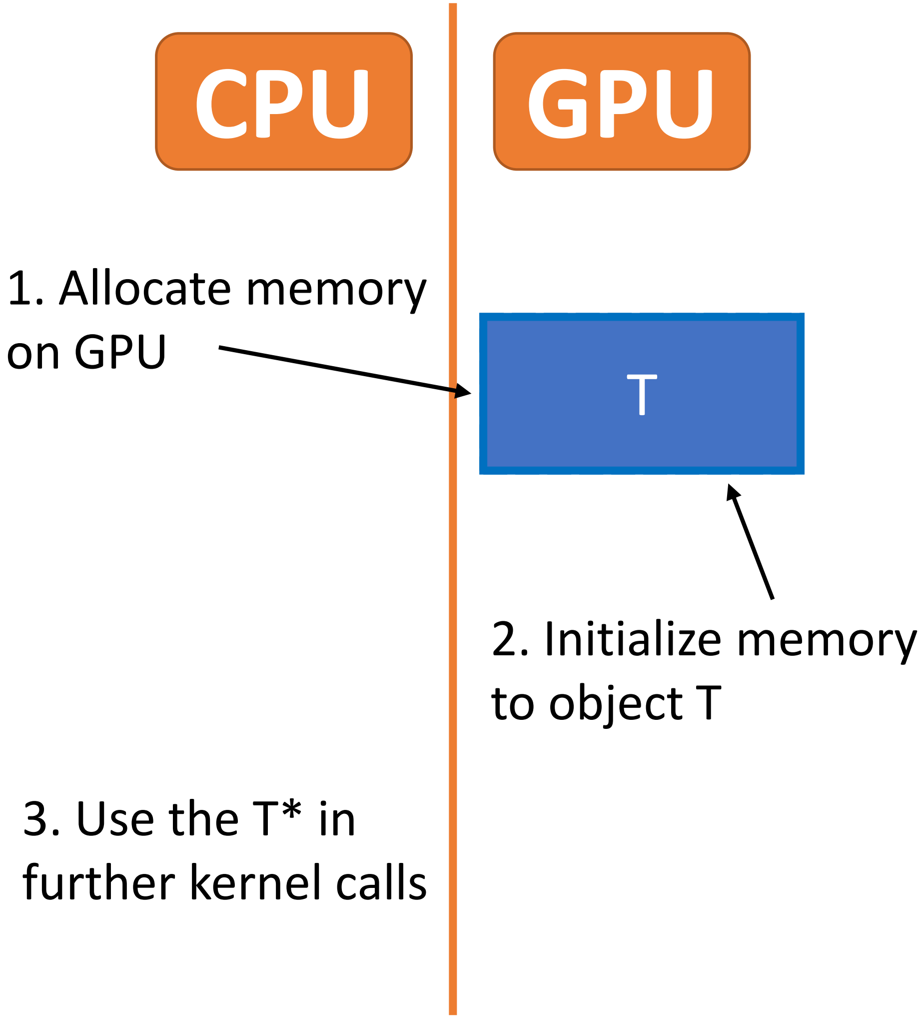
Solution 1: from template to virtual

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

template<typename T> __device__
void addKernelDo(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) {
  addKernelDo(c, a, b, OP<T>{});
}

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

Solution 2: dynamic allocation

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

template<template<typename> typename OP, 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]);
}

DevObject<BinaryOpPlus<int>> op;
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) 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();
  }
};
```

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

```
DevObject<BinaryOpPlus<int>> op;
addKernel<<<...>>>(cDev, aDev, bDev, op);
```

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

4. Release

```
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();
    }
  }
};
```

<static>
Polymorphism

Dynamic
Polymorphism

new/delete

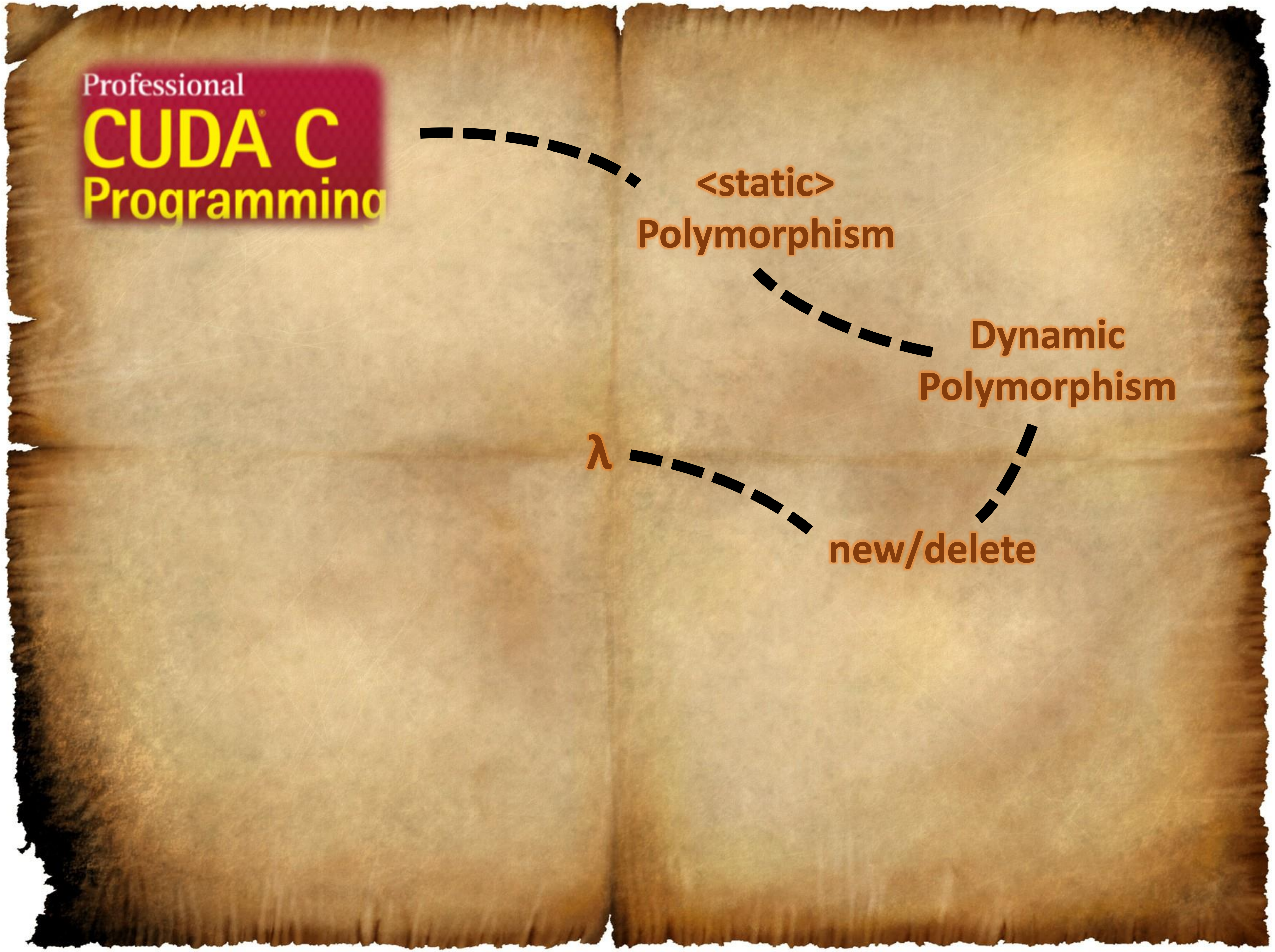
You can just use

malloc/free

and

new/delete

in the kernel code


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

Simplest lambda

```
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

```
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();
}
```

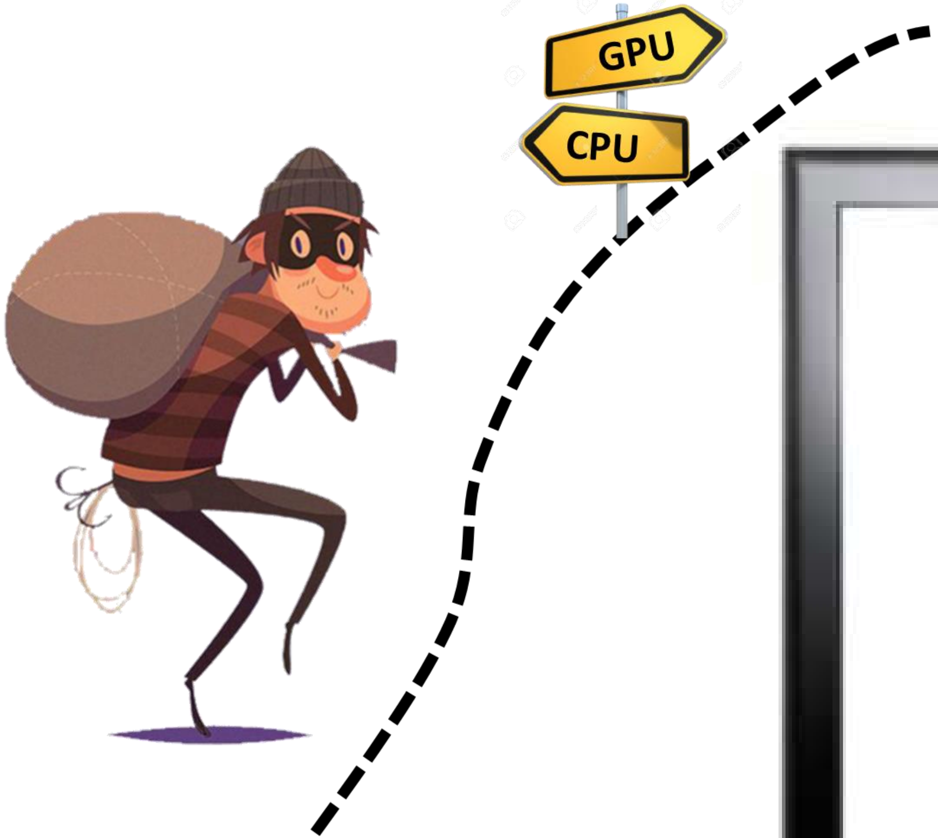
Lambda parameters!!

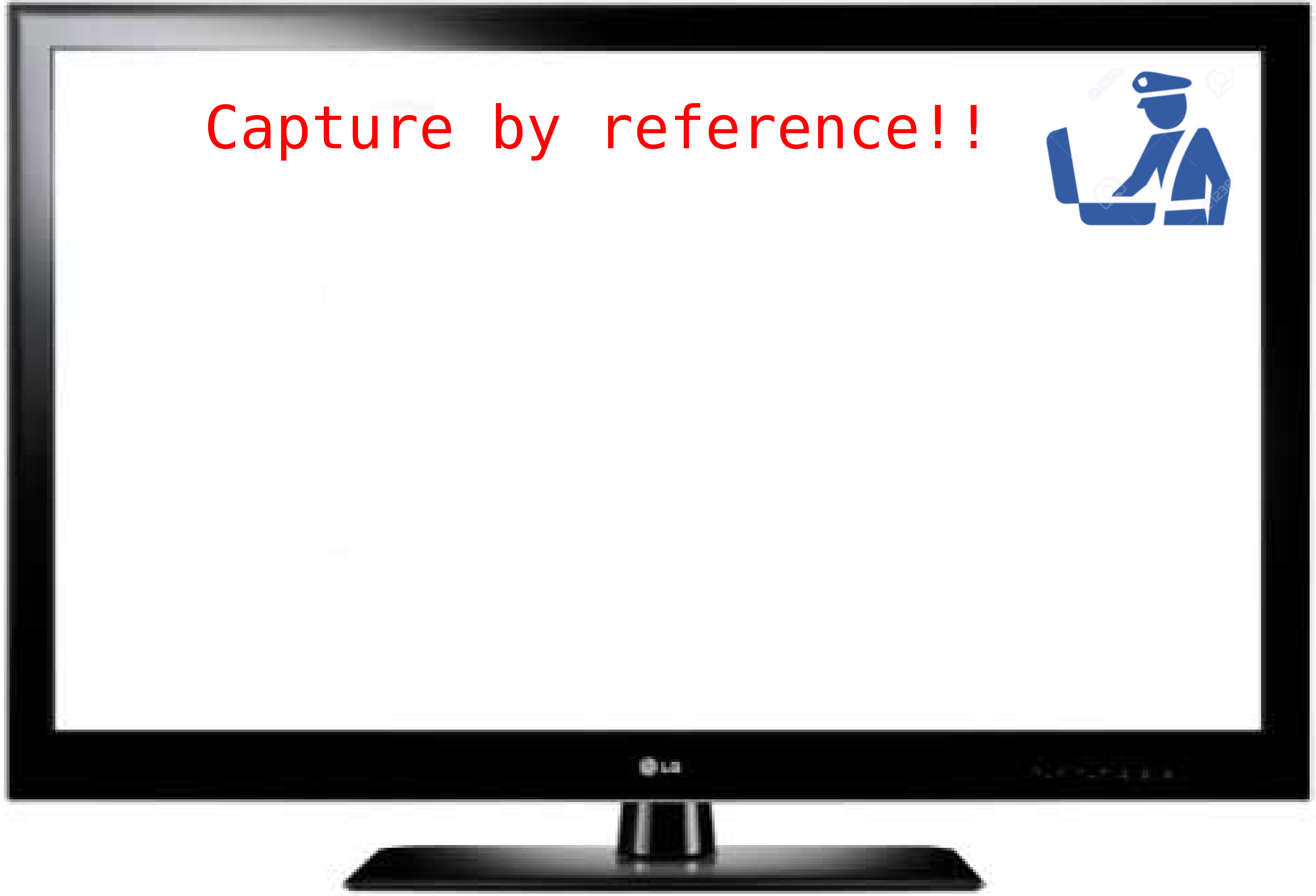
```
template<typename T, typename OP>
__global__ void addKernel(DevPtr<T> c, DevPtr<const T> a, DevPtr<const T> b,
                          OP op) {
  int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
  c[idx] = op(a[idx], b[idx]);
}

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

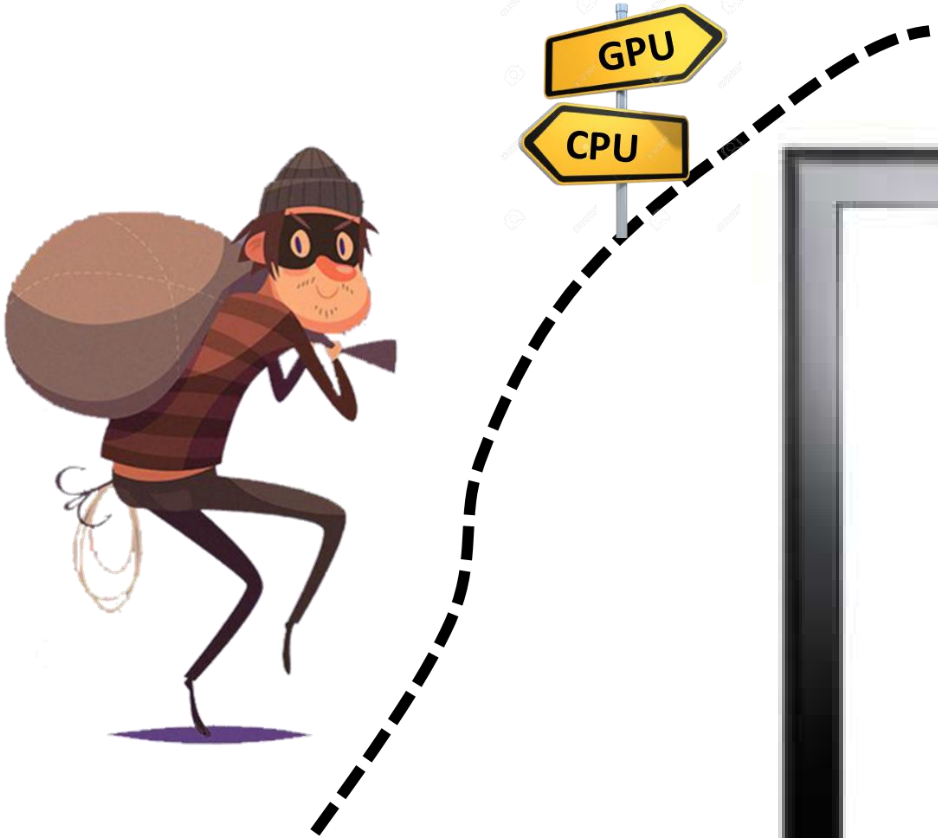
Note the *__device__* keyword

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

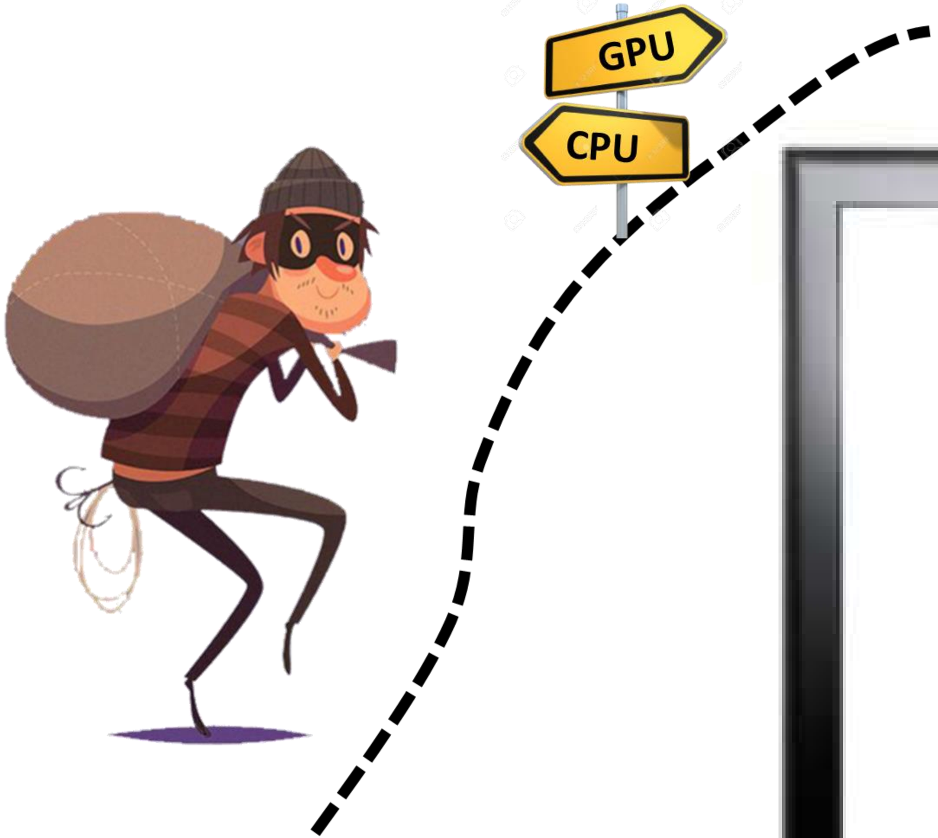

λ
GPU
CPU


GPU
CPU


Capture by reference!!



Capture by reference!!


GPU
CPU


this pointer!!

```cpp
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);
  //…
}
```

```
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);
  //…
}
```

```
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);
  //…
}
```

```
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};
  ?????
  //…
}
```

```
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};
  addKernel<<<blocks, 32>>>(cDev, aDev, bDev, op.make_op());
  //…
}
```

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());
  //…
}
```

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());
  //…
}
```

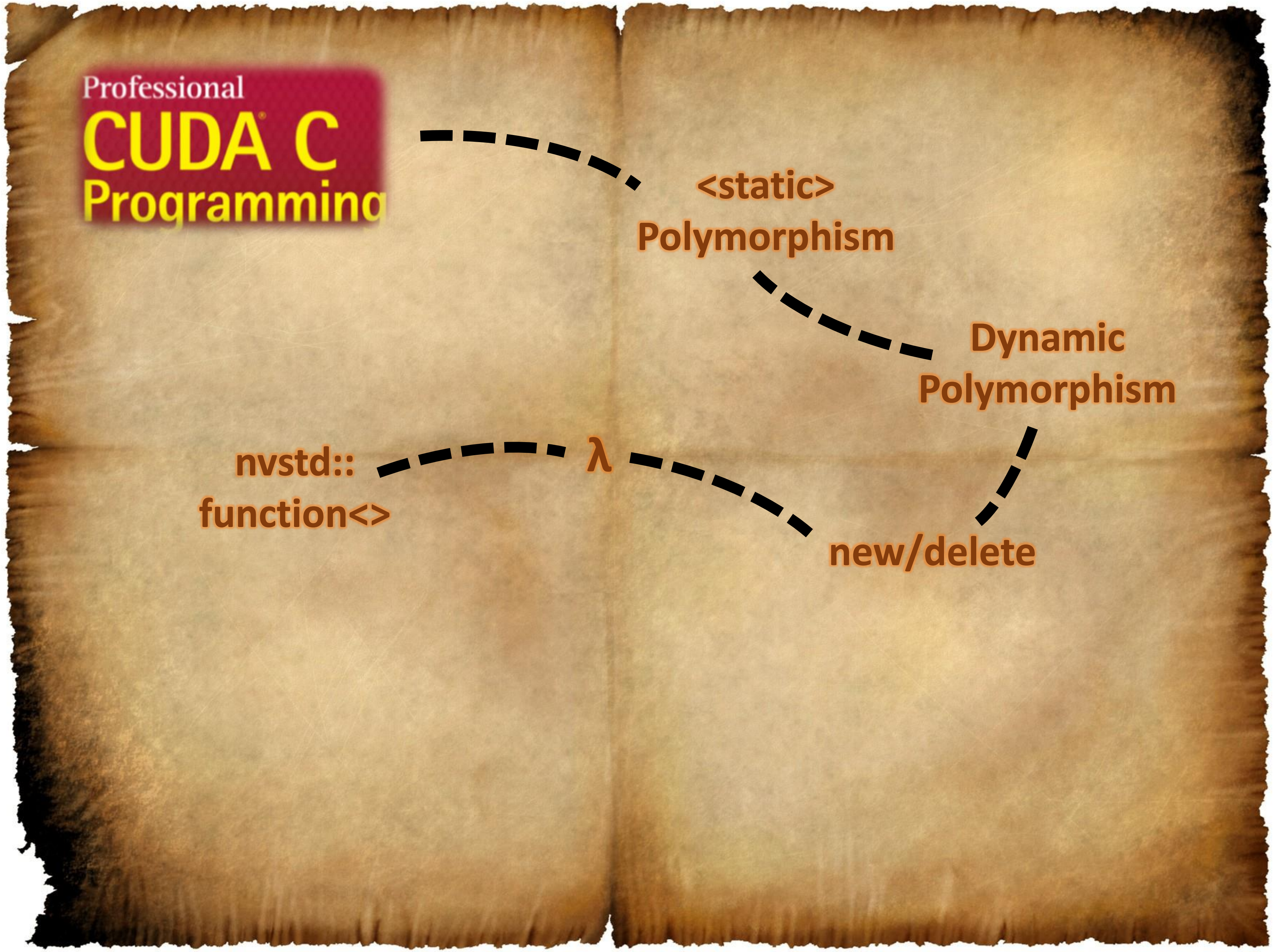
Attempt 3

Compiles but fails at runtime

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

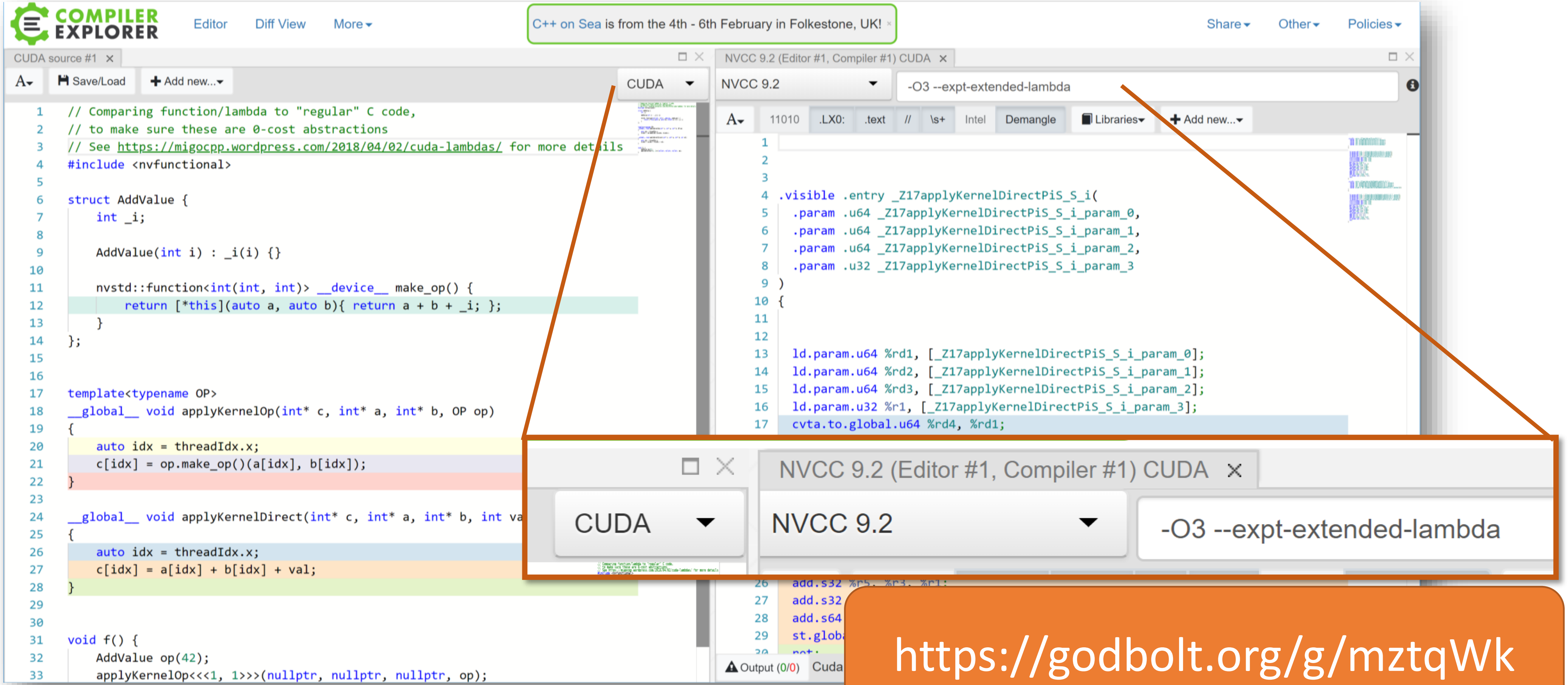
```
struct OP {
  int _i;
  explicit OP(int i) : _i(i) {}

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

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

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


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

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

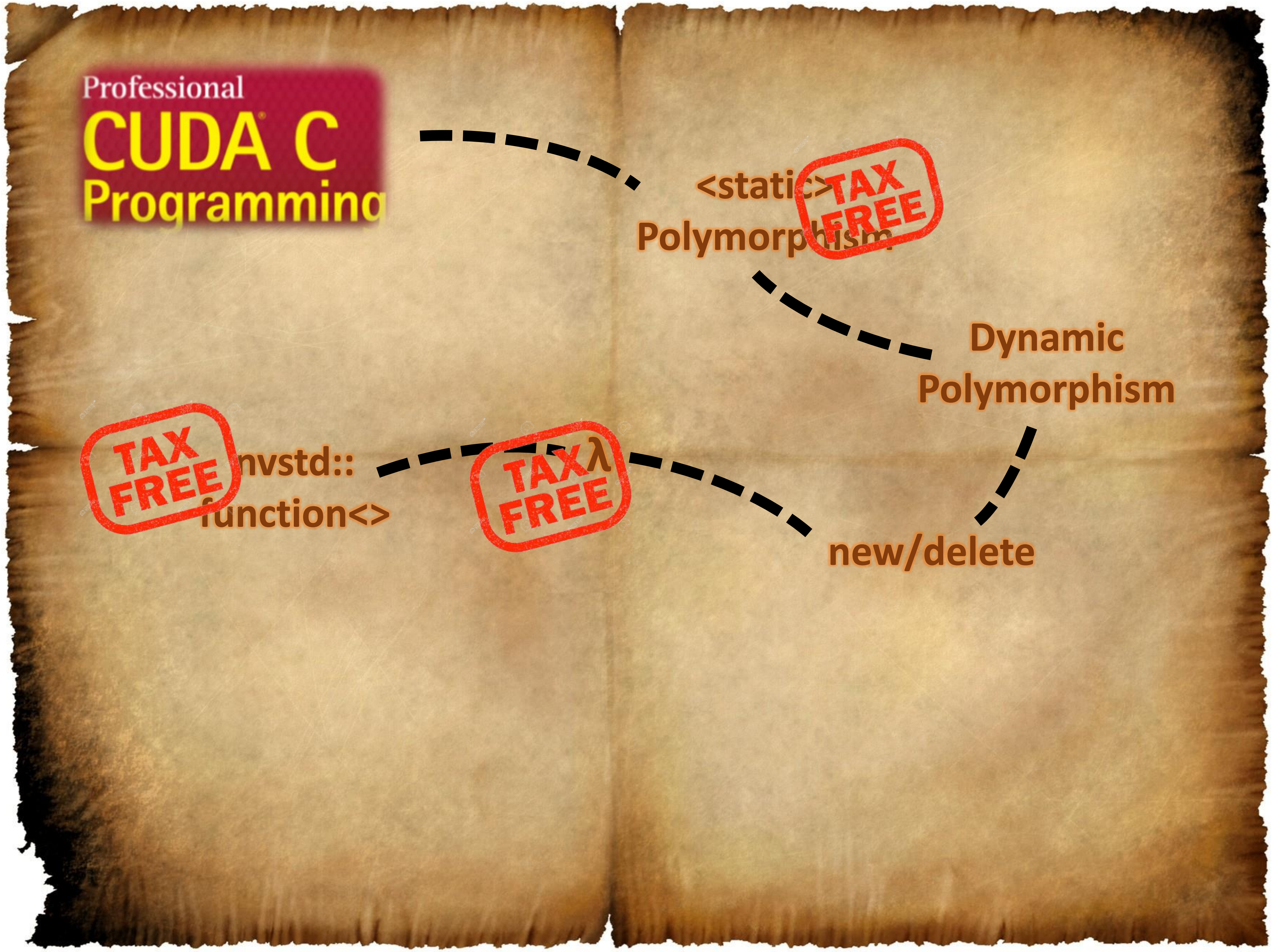
Godbolting CUDA

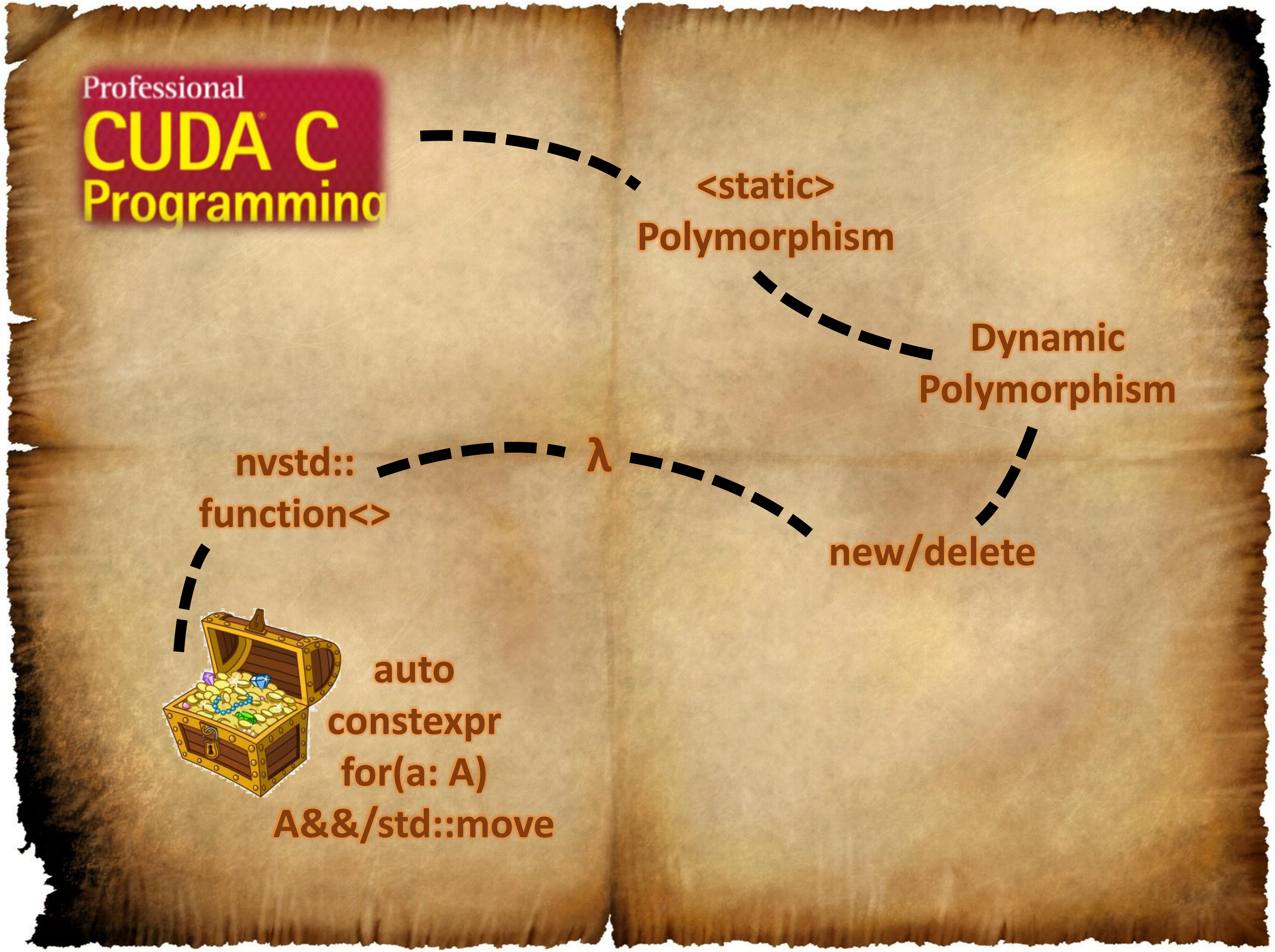


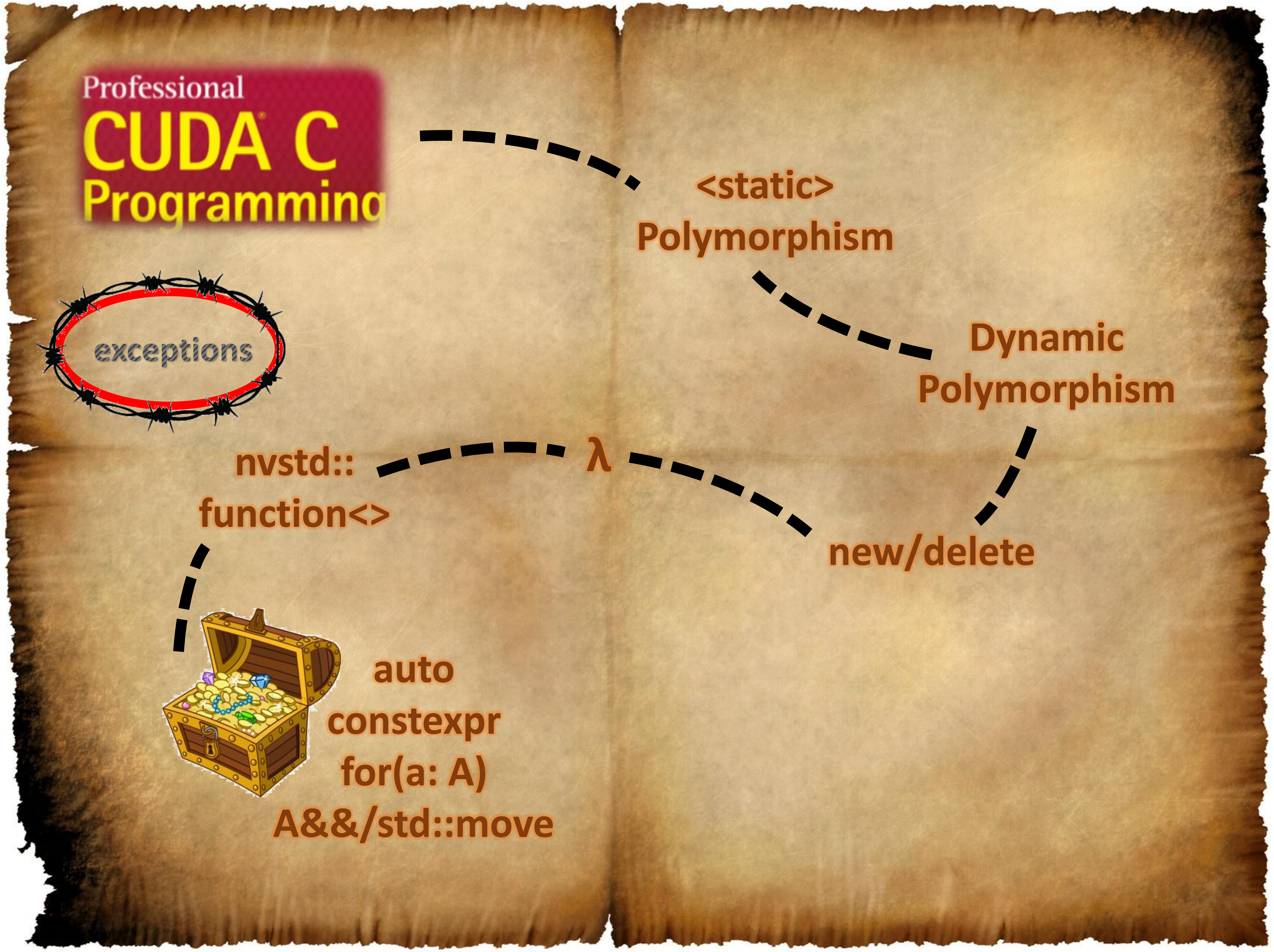
https://godbolt.org/g/mztqWk

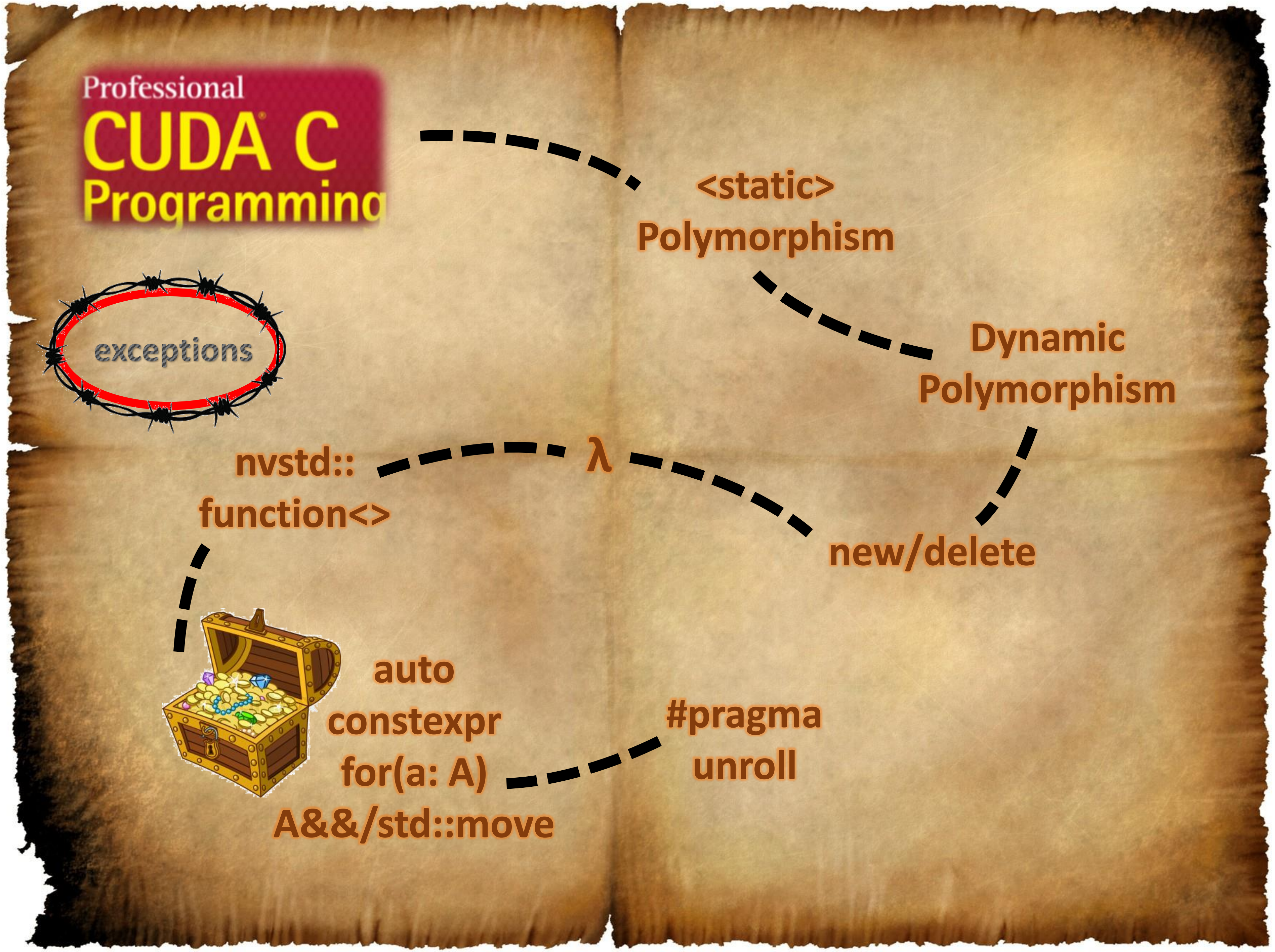
cuobjdump

```
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
nvstd::
function<>
TAX FREE
λ
TAX FREE
new/delete


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


Professional
CUDA C
Programming
<static>
Polymorphism
exceptions
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

```
template<typename T, typename F>
__device__ void apply_function(T *in, T *out, F f, size_t length) {

  for (auto i = 0; i < length; ++i)
    out[i] += f(in[i]);
}

__device__ void dowork(int *in, int *out, size_t length) {
  auto work = [] (int in) { /*few lines of code*/ ;
  apply_function    (in, out, work, length);
}
```

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

template<int unrollFactor, typename T, typename F>
__device__ void apply_function(T *in, T *out, F f, size_t length) {
  #pragma unroll mymax(unrollFactor, 32)
  for (auto i = 0; i < length; ++i)
    out[i] += f(in[i]);
}

__device__ void dowork(int *in, int *out, size_t length) {
  auto work = [] (int in) { /*few lines of code*/ ;
  apply_function<64>(in, out, work, length);
}
```

Runtime Templates

Why use runtime CUDA compilation?

- No need for NVCC compiler – the code is plain C++
- Runtime tuning of compilation flags (*architecture* etc.)
- **Runtime selection of template parameters**

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


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

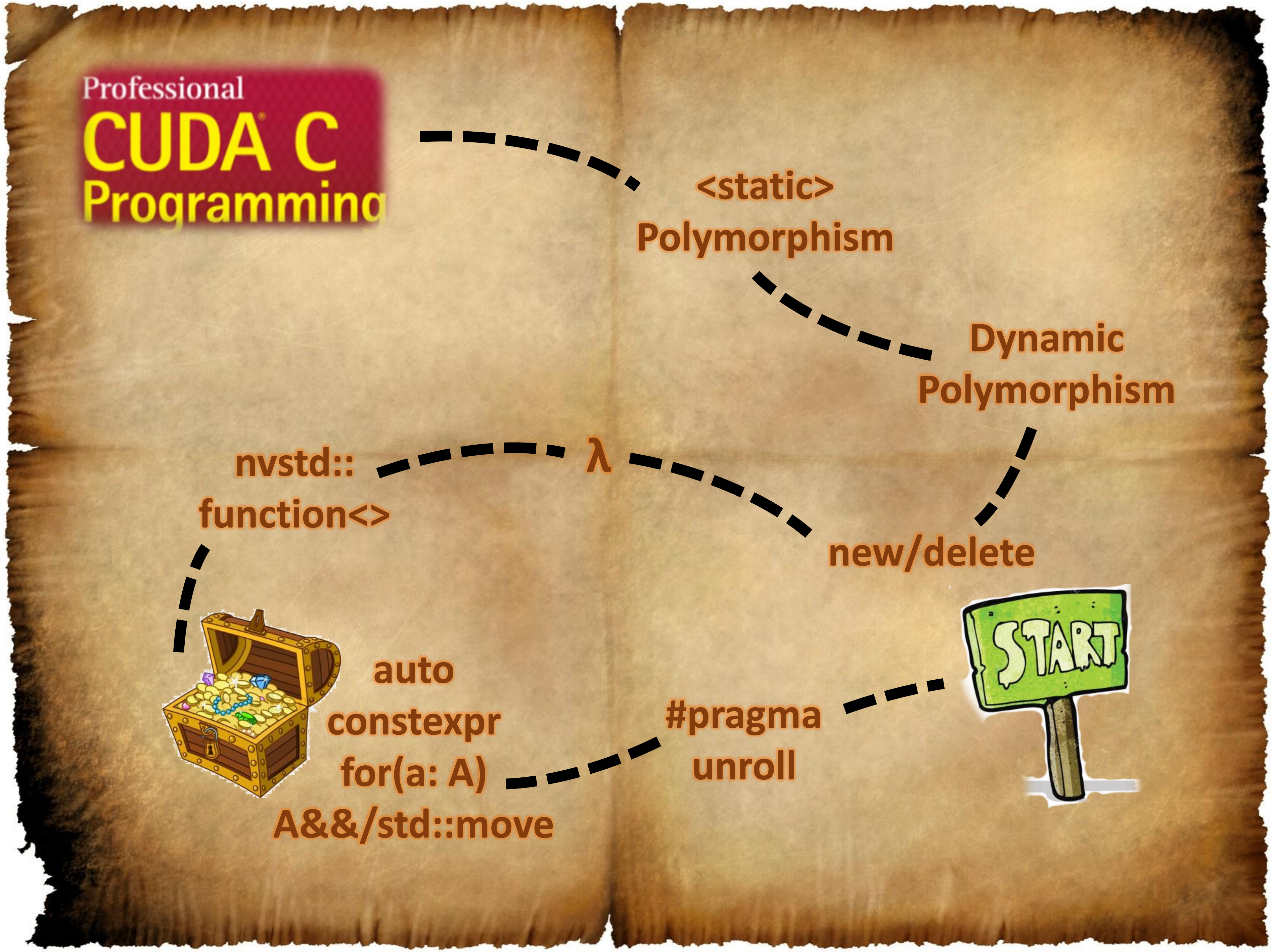
```
template<int LAYERS, typename T>
__global__ void process(T *data) {/*…*/}

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

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

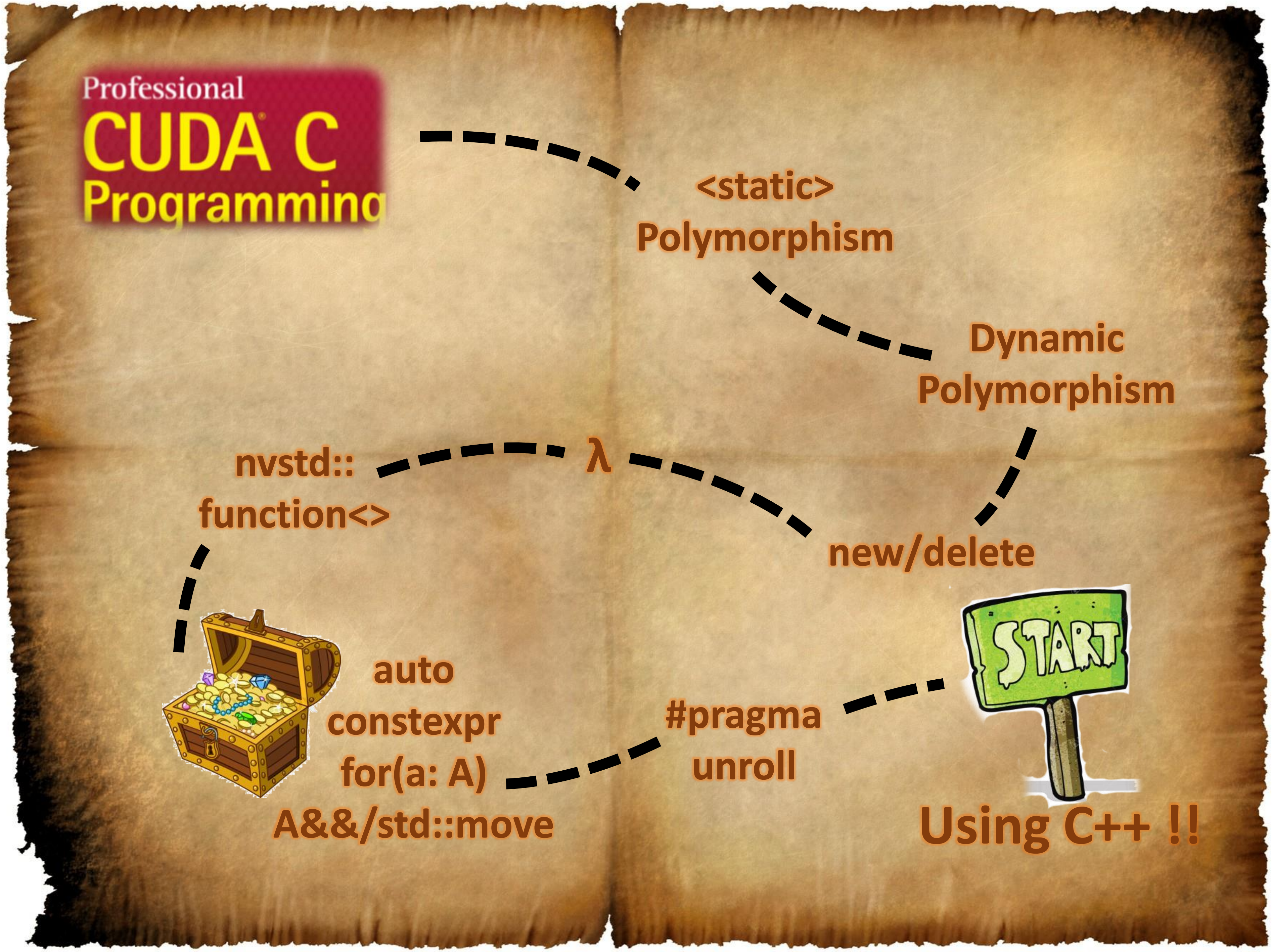
All the template instantiations are being compiled

Another option – compile CUDA at runtime

- Need to use *"**CUDA Driver API**"*

  `nvrtcGetTypeName<T>`
  `nvrtcAddNameExpression`
  `nvrtcGetLoweredName`
  etc.
- Examples – documentation, my blog post
- Be extra careful, the kernel is invoked using `cuLaunchKernel`, no compiler validation for parameters.


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


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

https://**migocpp**.wordpress.com/

@michael_gop

mgopshtein/cudacpp
(code examples)

- 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

NOP

EXTRA SLIDES

Solution 1: always use max-dim index

```
__device__ __inline__ int my1DimIndex() {
   int blockId = blockIdx.x
               + blockIdx.y * gridDim.x
               + blockIdx.z * gridDim.x * gridDim.y;
   int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
                + threadIdx.x
                + threadIdx.y * blockDim.x
                + threadIdx.z * blockDim.x * blockDim.y;
   return threadId;
}

template<typename T>
__global__ void addKernel(DevPtr<T> c, DevPtr<const T> a, DevPtr<const T> b) {
   int idx = my1DimIndex();
   c[idx] = a[idx] + b[idx];
}
```

OpenCV Integration

OpenCV provides **`cuda::GpuMat`** class which takes care for memory allocation and copying.