THE CUDA C++ DEVELOPER'S TOOLBOX

Bryce Adelstein Lelbach

Principal Architect

brycelelbach@gmail.com @blelbach

NVIDIA

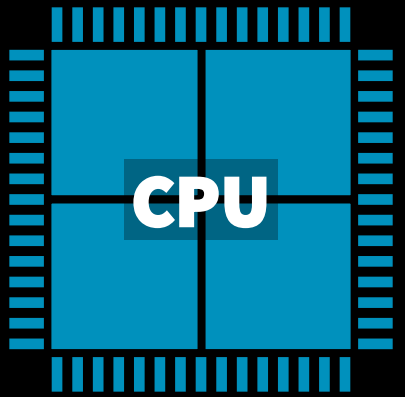
THE CUDA C++ DEVELOPER'S TOOLBOX

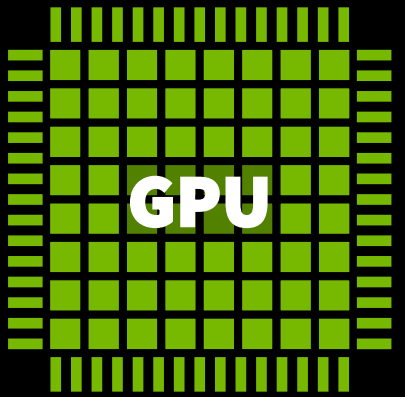
Bryce Adelstein Lelbach

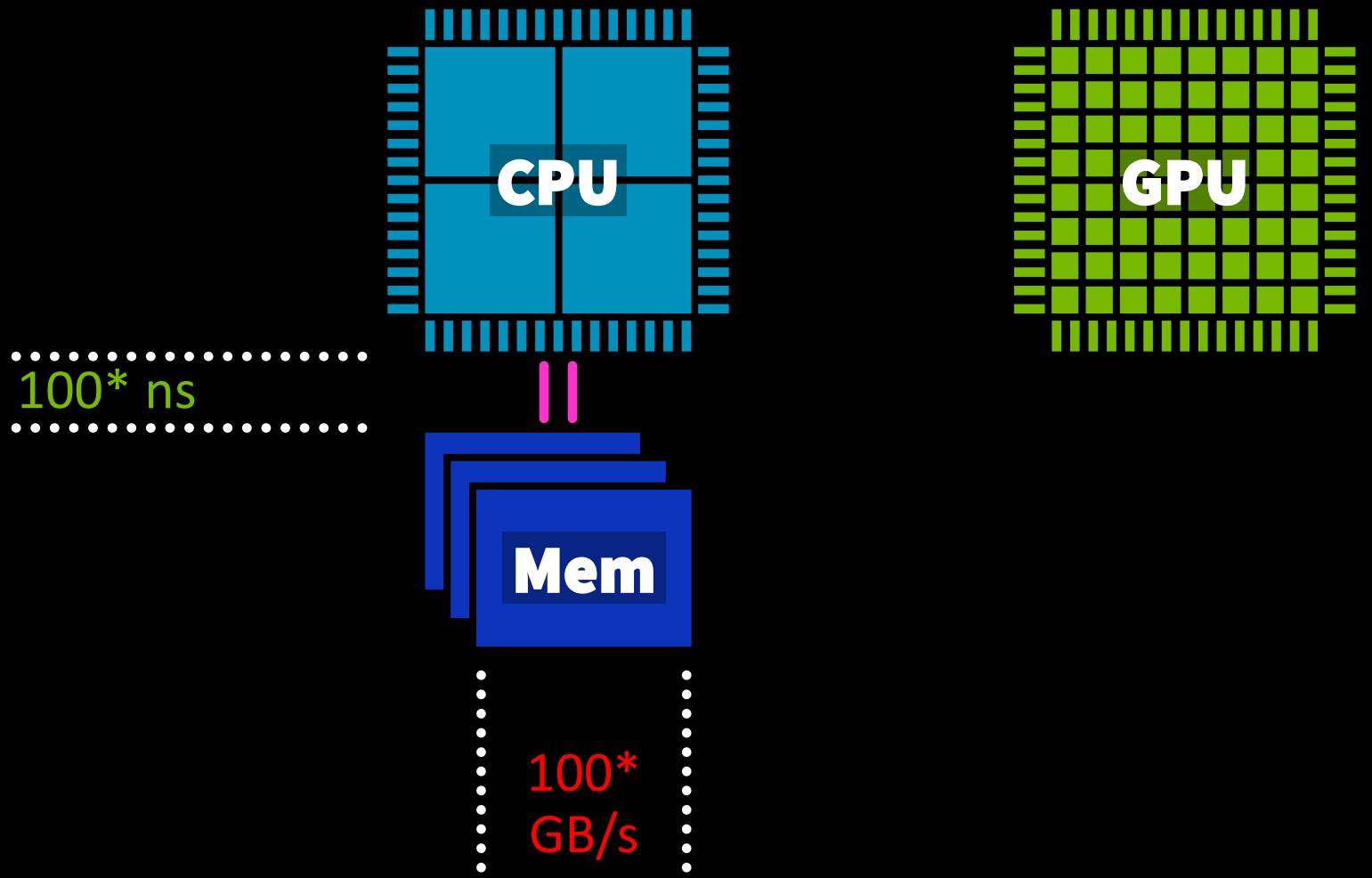
Principal Architect

brycelelbach@gmail.com @blelbach

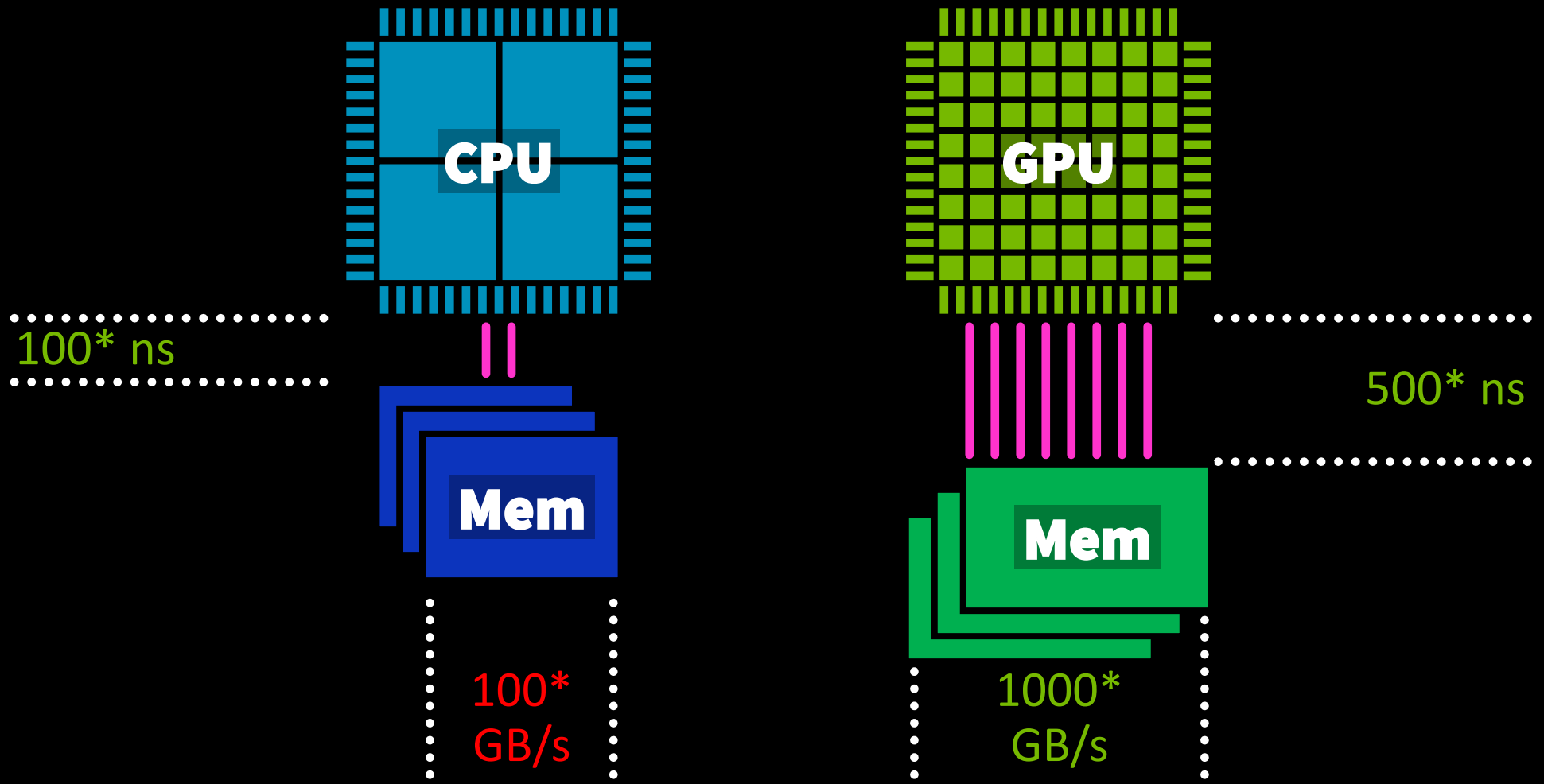
NVIDIA







* Numbers are made up and for expository purposes only.



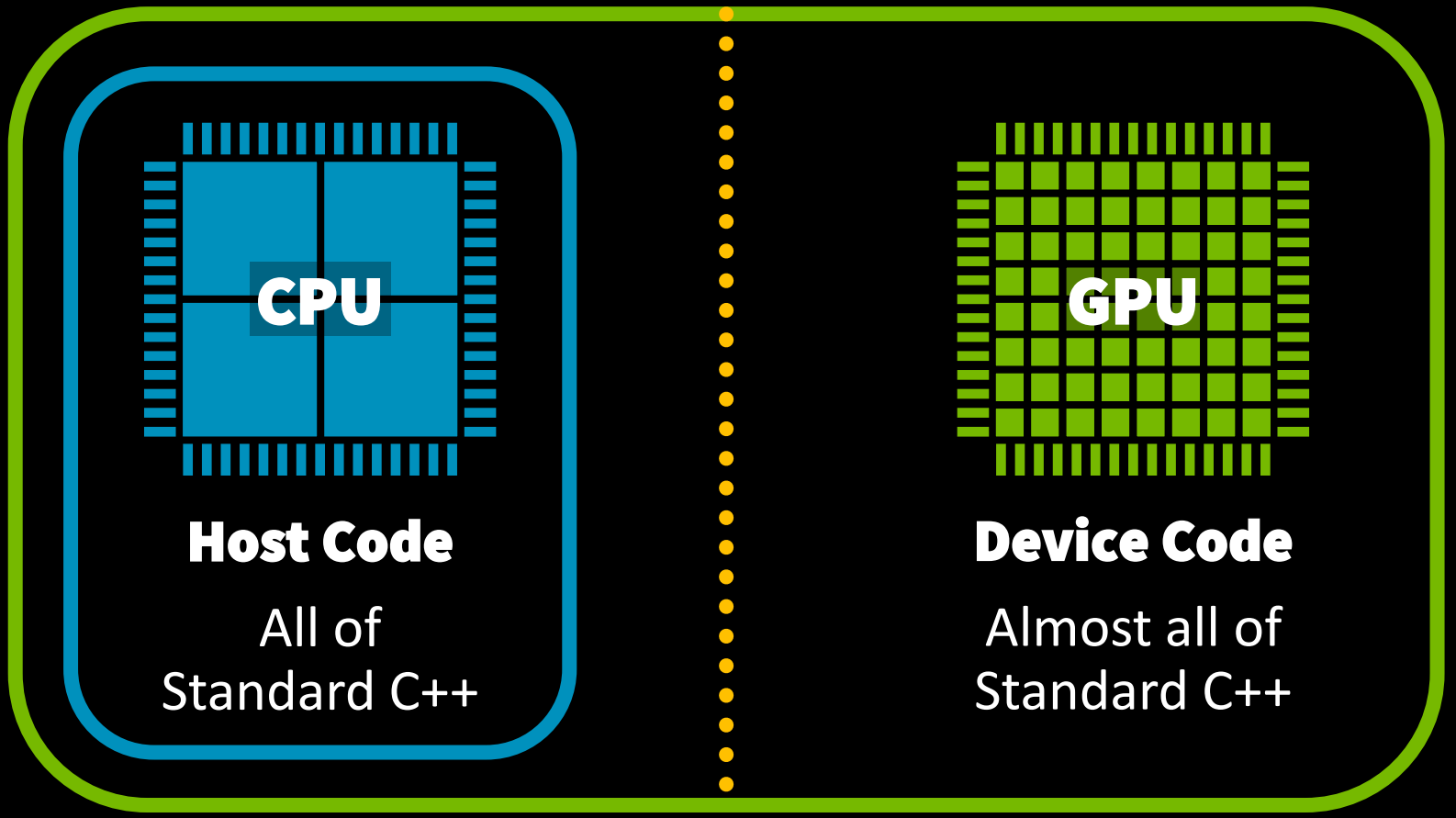
* Numbers are made up and for expository purposes only.

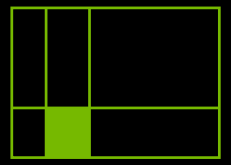
How do we program a GPU in C++?

How do we program a GPU in C++?

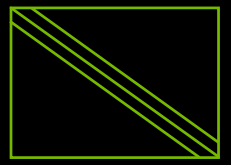
CUDA C++ and Accelerated Libraries

CUDA C++ is an extension of Standard C++ for writing programs that simultaneously run on CPUs (hosts) & GPUs (devices).



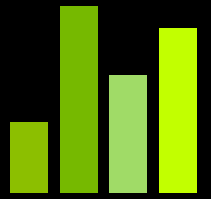


cuBLAS

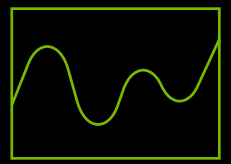


cuSPARSE

cuFile (GDS)



NVBench



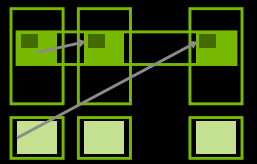
cuFFT

cuDSS

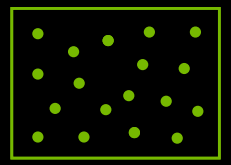
CV CUDA

cuLitho

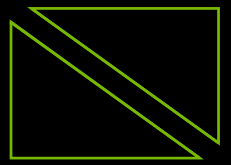
Cooperative Groups



NVSHMEM



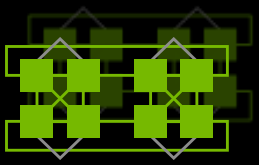
cuRAND



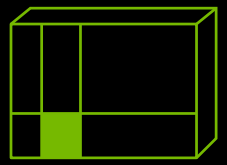
cuSOLVER

CUDA-Q

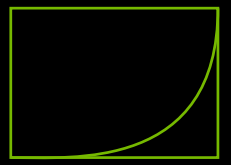
CUTLASS



NCCL

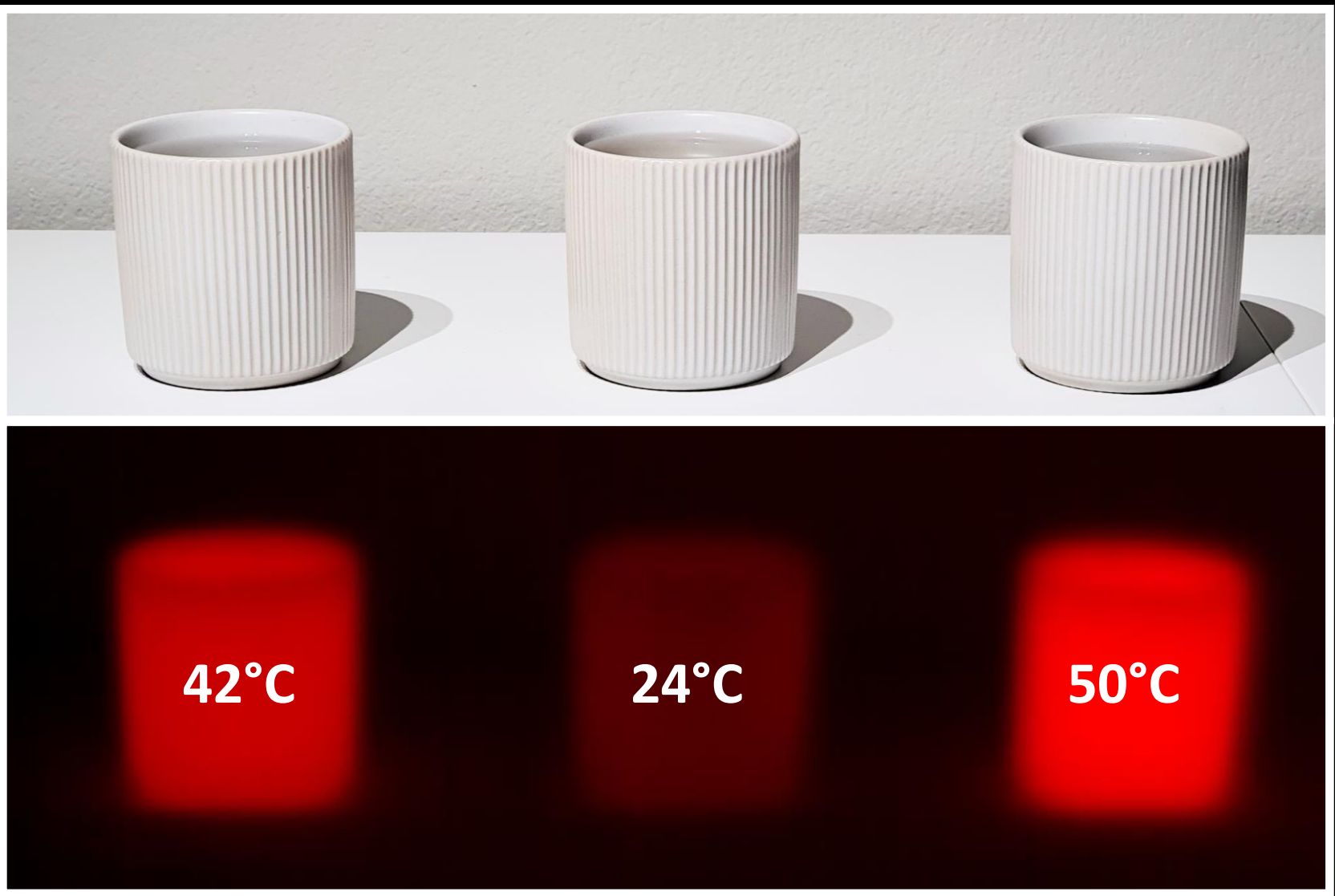


cuTENSOR



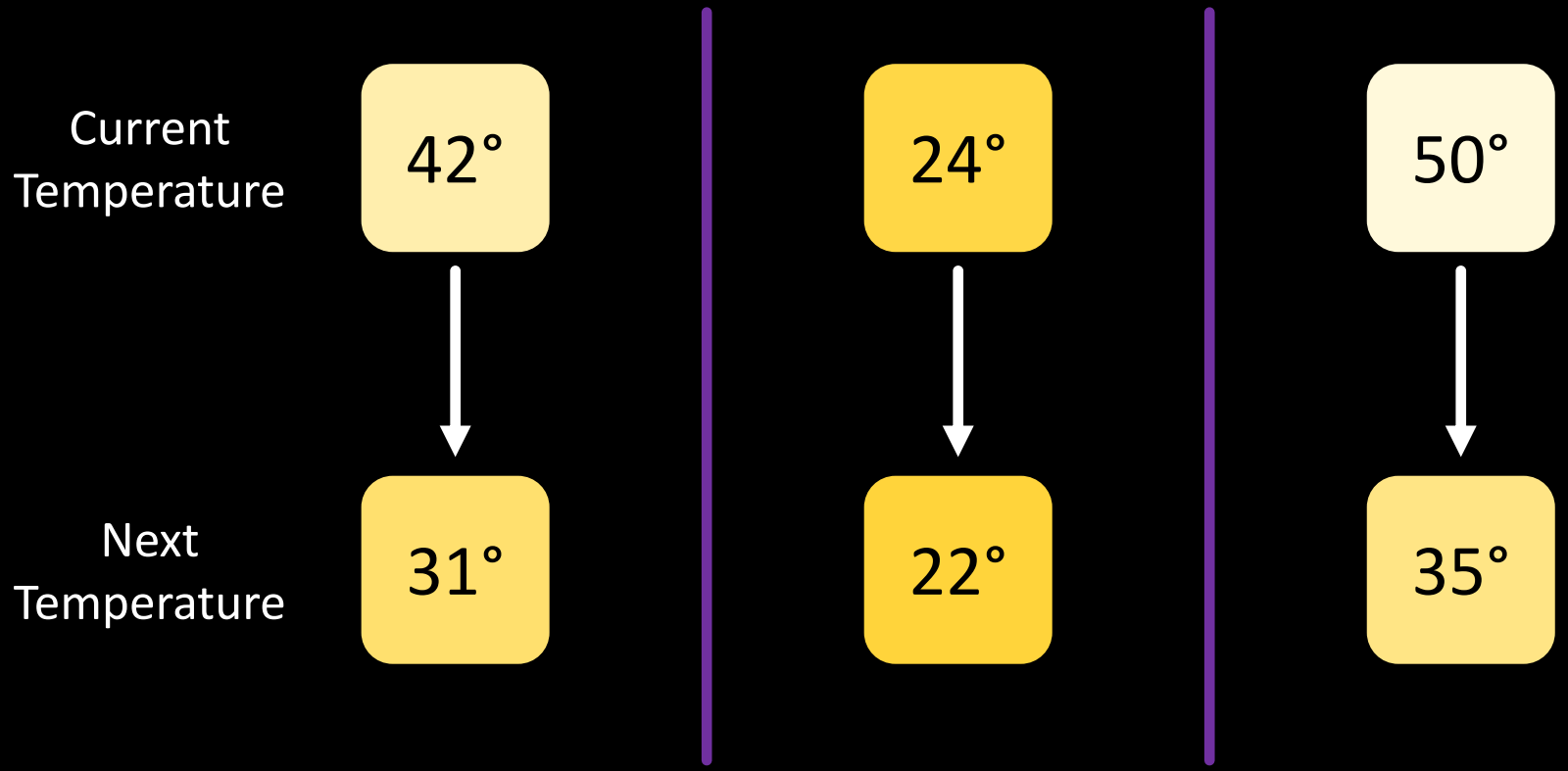
CUDA Math

CUDA Runtime



Ambient Temperature = 20°C

Heat Transfer Coefficient (k) = 0.5



Next Temp = Current Temp + Heat Transfer Coefficient * (Ambient Temp – Current Temp)

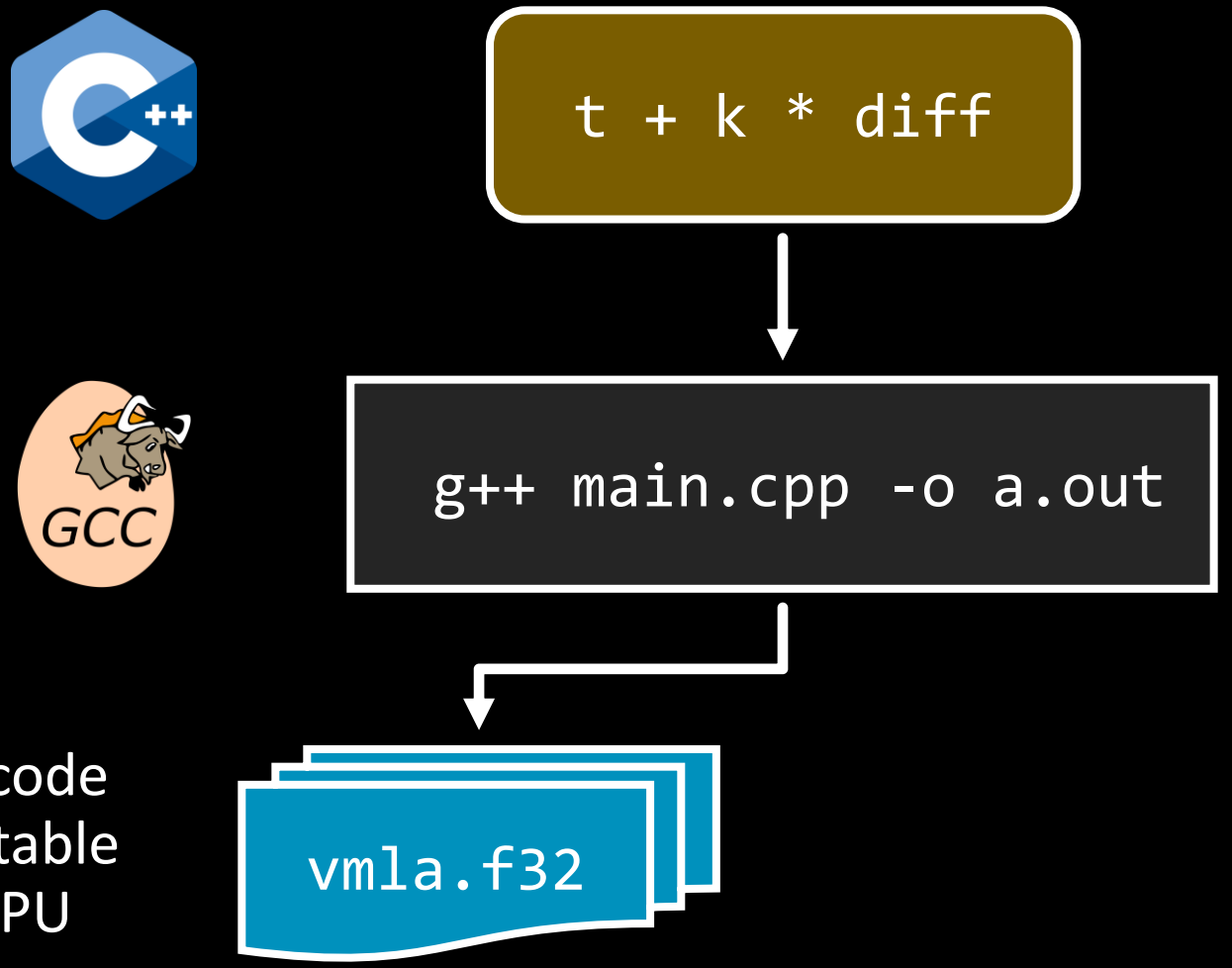
```
int steps = 3;
float k = 0.5;
float ambient_temp = 20;
std::vector<float> cups{42, 24, 50};
```

```
int steps = 3;
float k = 0.5;
float ambient_temp = 20;
std::vector<float> cups{42, 24, 50};

auto op = [=] (float t) {
  float diff = ambient_temp - t;
  return t + k * diff;
};
```

```
int steps = 3;
float k = 0.5;
float ambient_temp = 20;
std::vector<float> cups{42, 24, 50};

auto op = [=] (float t) {
  float diff = ambient_temp - t;
  return t + k * diff;
};

for (int step : std::views::iota(0, steps))
{
  …
}
```

```
int steps = 3;
float k = 0.5;
float ambient_temp = 20;
std::vector<float> cups{42, 24, 50};

auto op = [=] (float t) {
  float diff = ambient_temp - t;
  return t + k * diff;
};

for (int step : std::views::iota(0, steps))
{
  std::println("{} {}", step, cups);
  std::transform(cups.begin(), cups.end(),
                 cups.begin(), op);
}
```

```
std::transform(first, last,
               result, op);
```



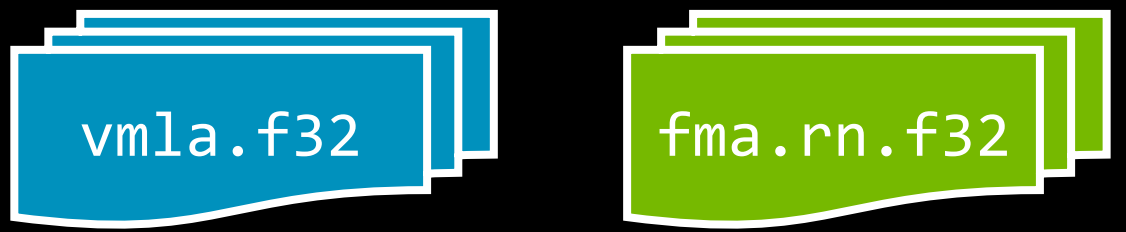
Host code executable by CPU

CUDA C++

```
t + k * diff
```

NVCC

```
nvcc main.cpp -o a.out
```



Host code executable by CPU

Device code executable by GPU

```
int steps = 3;
float k = 0.5;
float ambient_temp = 20;
std::vector<float> cups{42, 24, 50};

auto op = [=] (float t) {
  float diff = ambient_temp - t;
  return t + k * diff;
};

for (int step : std::views::iota(0, steps))
{
  std::println("{} {}", step, cups);
  std::transform(cups.begin(), cups.end(),
                 cups.begin(), op);
}
```

```
int steps = 3;
float k = 0.5;
float ambient_temp = 20;
std::vector<float> cups{42, 24, 50};

auto op = [=] __host__ __device__ (float t) {
  float diff = ambient_temp - t;
  return t + k * diff;
};

for (int step : std::views::iota(0, steps))
{
  std::println("{} {}", step, cups);
  std::transform(cups.begin(), cups.end(),
                 cups.begin(), op);
}
```

```
int steps = 3;
float k = 0.5;
float ambient_temp = 20;
std::vector<float> cups{42, 24, 50};

auto op = [=] __host__ __device__ (float t) {
  float diff = ambient_temp - t;
  return t + k * diff;
};

for (int step : std::views::iota(0, steps))
{
  std::println("{} {}", step, cups);
  std::transform(cups.begin(), cups.end(),
                 cups.begin(), op);
}
```

`void a();`
 	
`__host__`
`void b();`
	Only executable by the CPU. Includes all unannotated functions.

```
int steps = 3;
float k = 0.5;
float ambient_temp = 20;
std::vector<float> cups{42, 24, 50};

auto op = [=] __host__ __device__ (float t) {
  float diff = ambient_temp - t;
  return t + k * diff;
};

for (int step : std::views::iota(0, steps))
{
  std::println("{} {}", step, cups);
  std::transform(cups.begin(), cups.end(),
                 cups.begin(), op);
}
```

`void a();`
 	
`__host__`
`void b();`
	Only executable by the CPU. Includes all unannotated functions.

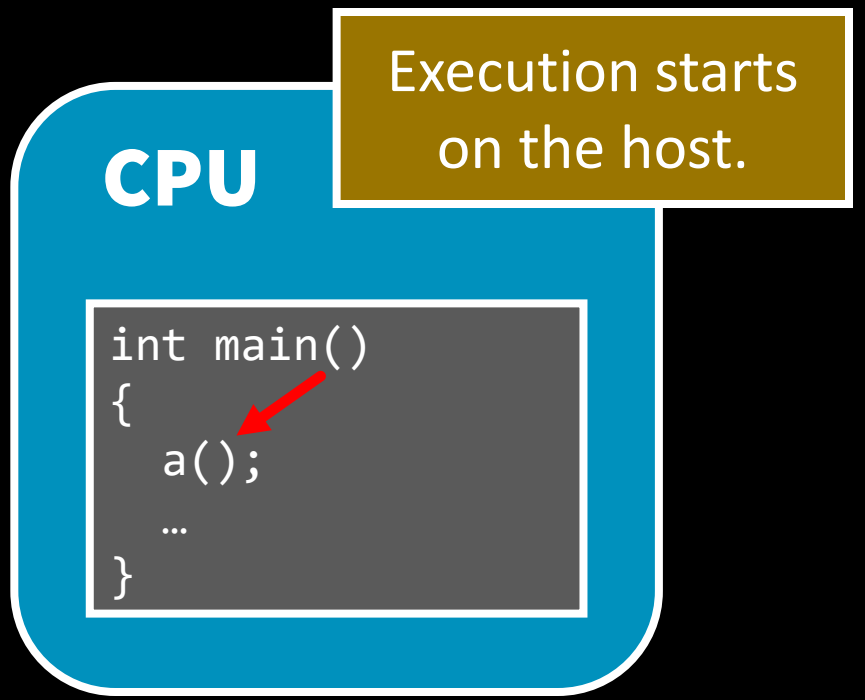
`__device__` `void c();`	Only executable by the GPU.

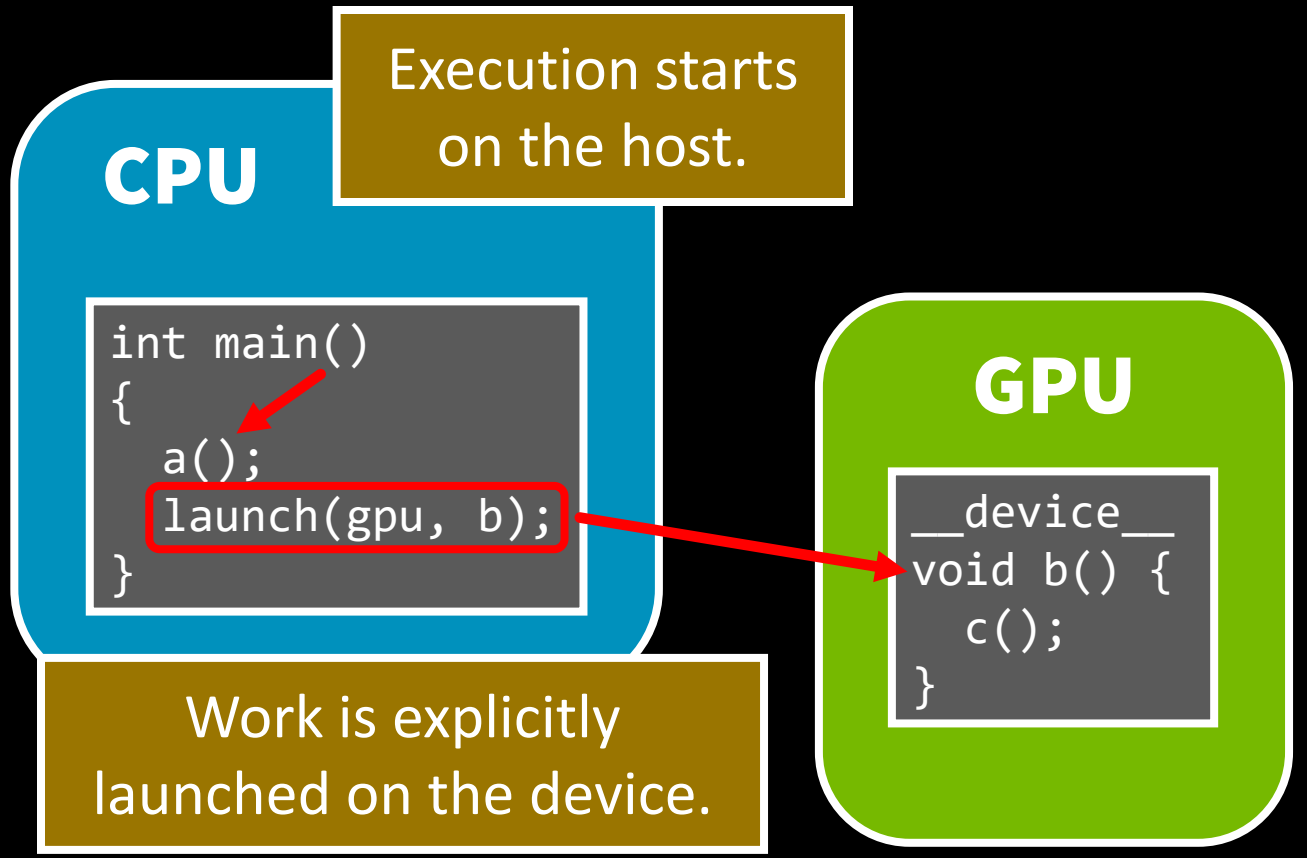
```
int steps = 3;
float k = 0.5;
float ambient_temp = 20;
std::vector<float> cups{42, 24, 50};

auto op = [=] __host__ __device__ (float t) {
  float diff = ambient_temp - t;
  return t + k * diff;
};

for (int step : std::views::iota(0, steps))
{
  std::println("{} {}", step, cups);
  std::transform(cups.begin(), cups.end(),
                 cups.begin(), op);
}
```

`void a();`
 	
`__host__`
`void b();`
	Only executable by the CPU. Includes all unannotated functions.

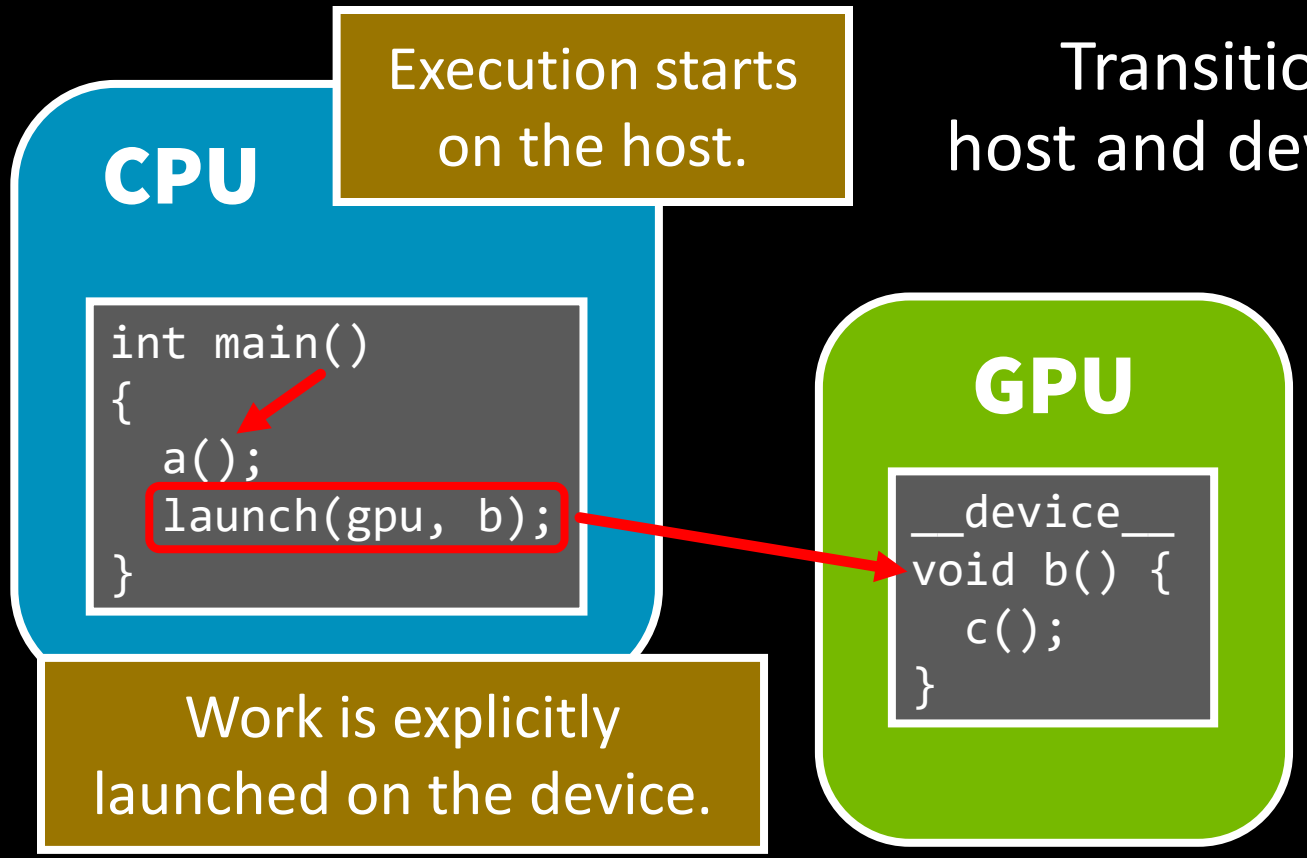
`__device__` `void c();`	Only executable by the GPU.

`__host__`
`__device__`
`void d();`
	Executable by CPU and GPU.

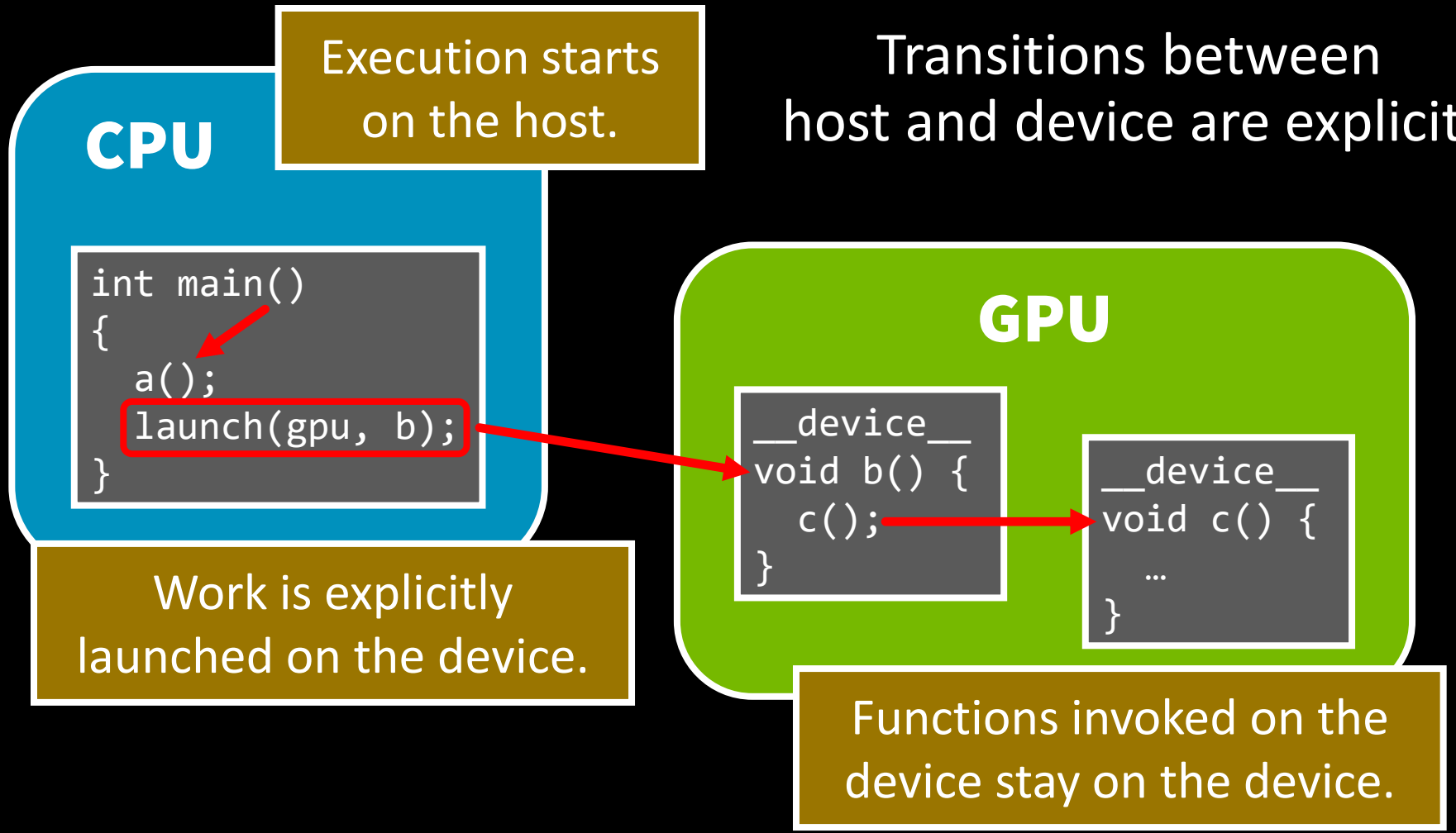




Transitions between host and device are explicit.



Transitions between host and device are explicit.



```
int steps = 3;
float k = 0.5;
float ambient_temp = 20;
std::vector<float> cups{42, 24, 50};

auto op = [=] __host__ __device__ (float t) {
  float diff = ambient_temp - t;
  return t + k * diff;
};

for (int step : std::views::iota(0, steps))
{
  std::println("{} {}", step, cups);
  std::transform(cups.begin(), cups.end(),
                 cups.begin(), op);
}
```

```
int steps = 3;
float k = 0.5;
float ambient_temp = 20;
std::vector<float> cups{42, 24, 50};

auto op = [=] __host__ __device__ (float t) {
  float diff = ambient_temp - t;
  return t + k * diff;
};

for (int step : std::views::iota(0, steps))
{
  std::println("{} {}", step, cups);
  thrust::transform(thrust::cuda::par, cups.begin(), cups.end(),
                    cups.begin(), op);
}
```

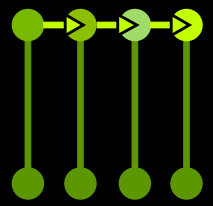
```
int steps = 3;
float k = 0.5;
float ambient_temp = 20;
thrust::universal_vector<float> cups{42, 24, 50};

auto op = [=] __host__ __device__ (float t) {
  float diff = ambient_temp - t;
  return t + k * diff;
};

for (int step : std::views::iota(0, steps))
{
  std::println("{} {}", step, cups);
  thrust::transform(thrust::cuda::par, cups.begin(), cups.end(),
                    cups.begin(), op);
}
```

The C++ Parallel Algorithms Library

https://nvidia.github.io/cccl/thrust

The C++ Parallel Algorithms Library

https://nvidia.github.io/cccl/thrust

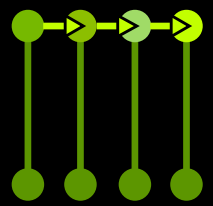


Standard Algorithms

- `thrust::transform_reduce`
- `thrust::inclusive_scan`
- `thrust::sort`
- `thrust::copy`
- `...`

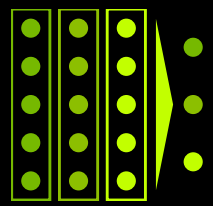
The C++ Parallel Algorithms Library

https://nvidia.github.io/cccl/thrust



Standard Algorithms

- `thrust::transform_reduce`
- `thrust::inclusive_scan`
- `thrust::sort`
- `thrust::copy`
- `...`

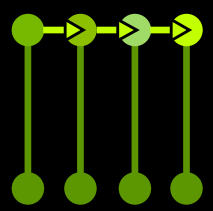


Extended Algorithms

- `thrust::reduce_by_key`
- `thrust::sort_by_key`
- `thrust::tabulate`
- `thrust::gather`
- `...`

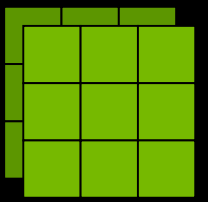
The C++ Parallel Algorithms Library

https://nvidia.github.io/cccl/thrust



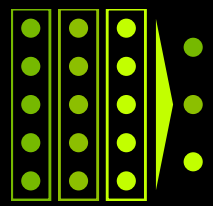
Standard Algorithms

- `thrust::transform_reduce`
- `thrust::inclusive_scan`
- `thrust::sort`
- `thrust::copy`
- `...`



Containers

- `thrust::device_vector`
- `thrust::host_vector`
- `thrust::universal_vector`
- `thrust::allocate_unique`
- `...`



Extended Algorithms

- `thrust::reduce_by_key`
- `thrust::sort_by_key`
- `thrust::tabulate`
- `thrust::gather`
- `...`

```
std::vector<T> h0(1 << 30);
std::vector<T> h1(1 << 30);

h0 = h1;
```

`std::vector`'s contents might only be accessible in *host code*.

Construction and assignment is *serial*.

```
std::vector<T> h0(1 << 30);
std::vector<T> h1(1 << 30);

h0 = h1;
```

`std::vector`'s contents might only be accessible in *host code*.

Construction and assignment is *serial*.

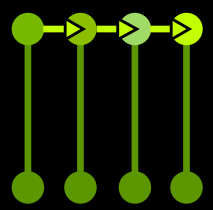
```
thrust::universal_vector<T> u0(1 << 30);
thrust::universal_vector<T> u1(1 << 30);

u0 = u1;
```

`thrust::universal_vector`'s contents are accessible in host and device code.

Construction and assignment is parallel.

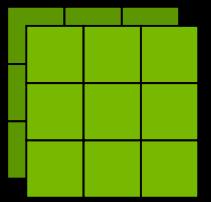
The C++ Parallel Algorithms Library

https://nvidia.github.io/cccl/thrust



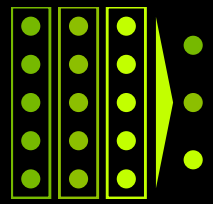
Standard Algorithms

- thrust::transform_reduce
- thrust::inclusive_scan
- thrust::sort
- thrust::copy
- ...



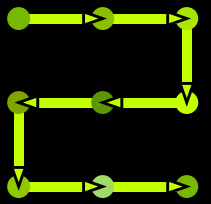
Containers

- thrust::device_vector
- thrust::host_vector
- thrust::universal_vector
- thrust::allocate_unique
- ...



Extended Algorithms

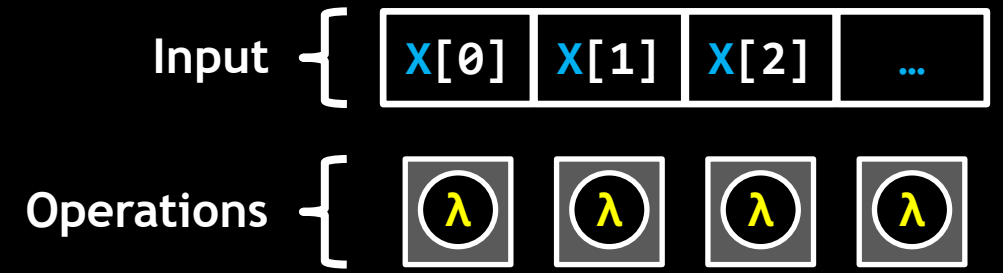
- thrust::reduce_by_key
- thrust::sort_by_key
- thrust::tabulate
- thrust::gather
- ...



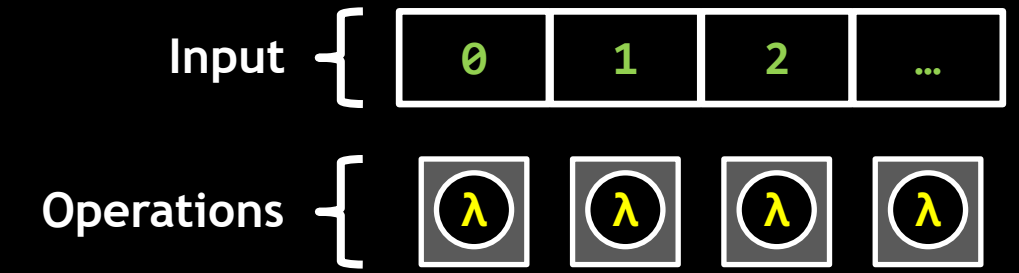
Iterators

- thrust::counting_iterator
- thrust::constant_iterator
- thrust::transform_iterator
- thrust::zip_iterator
- ...

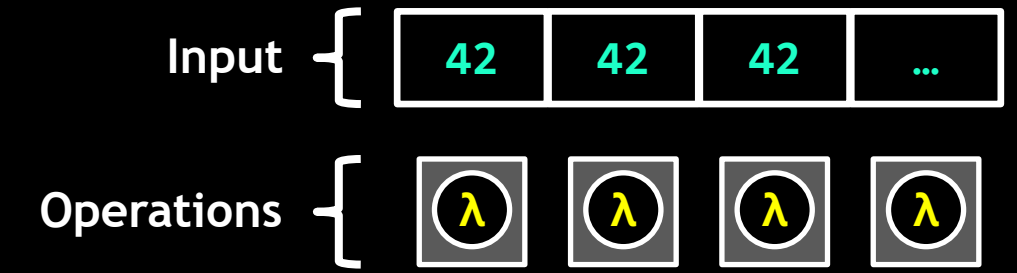
```
thrust::universal_vector X(N);

thrust::for_each_n(
  thrust::cuda::par, X.begin(), N,
  […] __host__ __device__ (auto& obj) { … });
```



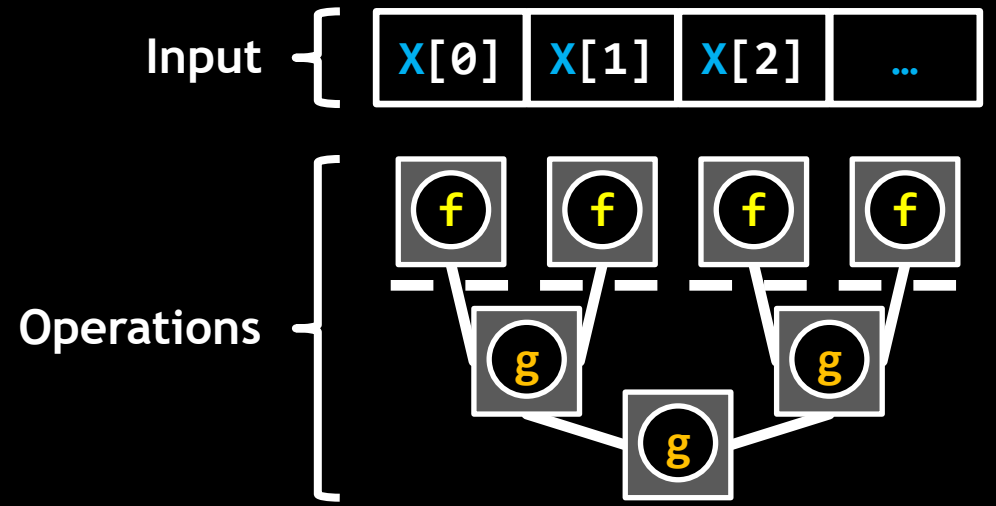
```
auto i = thrust::make_counting_iterator(0);

thrust::for_each_n(
  thrust::cuda::par, i, N,
  […] __host__ __device__ (auto idx) { … });
```



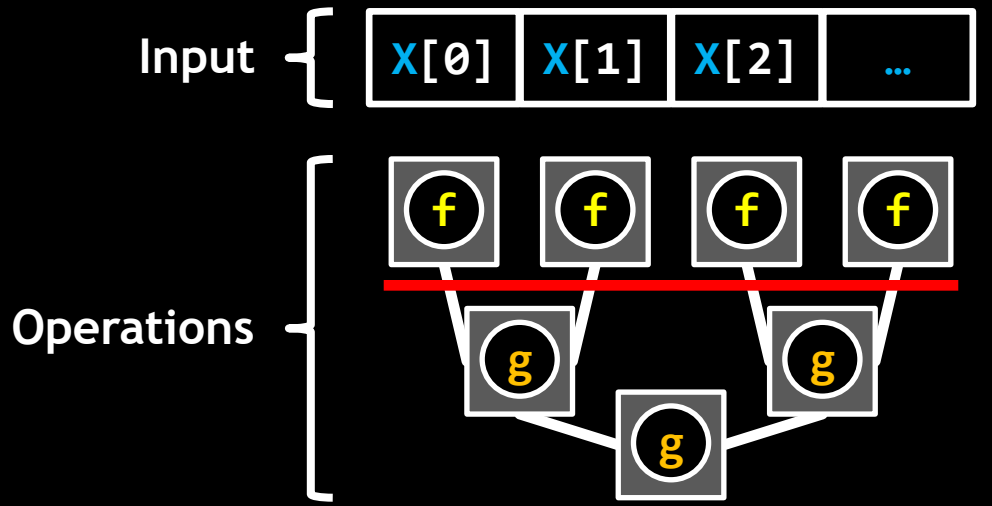
```
auto i = thrust::make_constant_iterator(42);

thrust::for_each_n(
  thrust::cuda::par, i, N,
  […] __host__ __device__ (auto idx) { … });
```

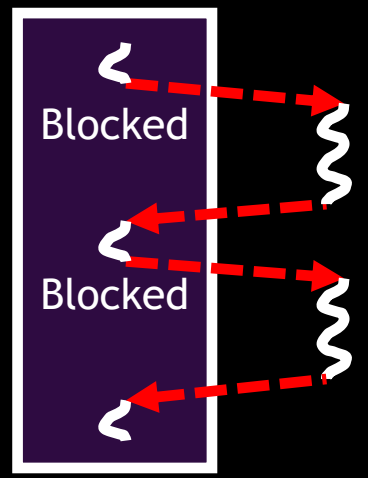


```
thrust::universal_vector X(N), tmp(N);

thrust::transform(thrust::cuda::par,
  X.begin(), X.end(), tmp.begin(), f);

auto r = thrust::reduce(thrust::cuda::par,
  tmp.begin(), tmp.end(), T{}, g);
```



```
thrust::universal_vector X(N), tmp(N);

thrust::transform(thrust::cuda::par,
  X.begin(), X.end(), tmp.begin(), f);

auto r = thrust::reduce(thrust::cuda::par,
  tmp.begin(), tmp.end(), T{}, g);
```

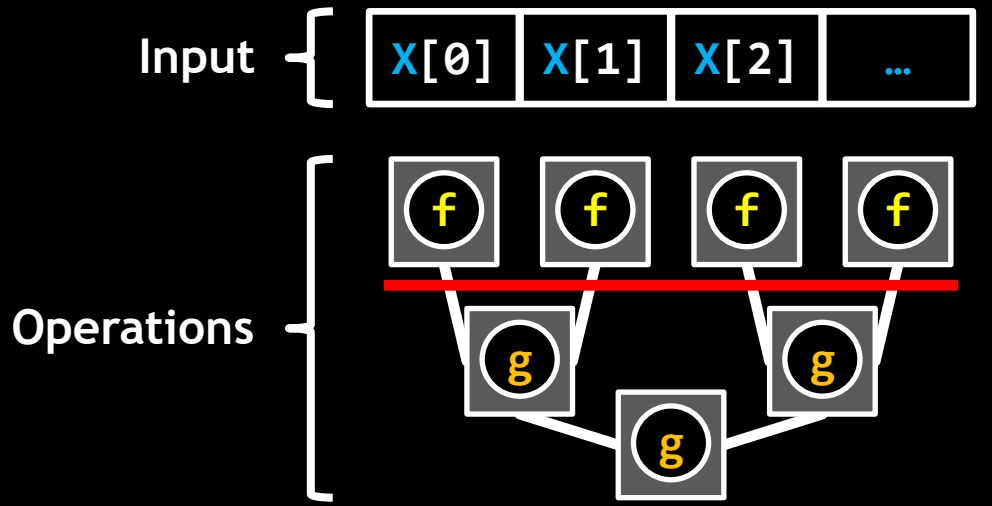


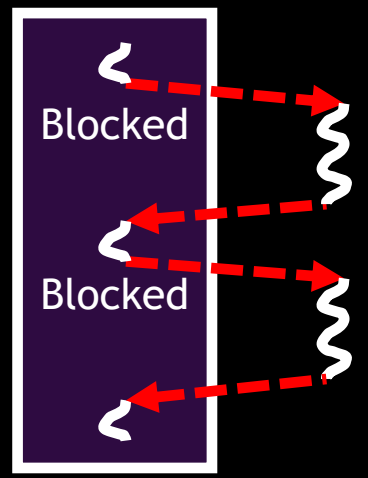


NVIDIA.

```
thrust::universal_vector X(N), tmp(N);

thrust::transform(thrust::cuda::par,
  X.begin(), X.end(), tmp.begin(), f);

auto r = thrust::reduce(thrust::cuda::par,
  tmp.begin(), tmp.end(), T{}, g);
```

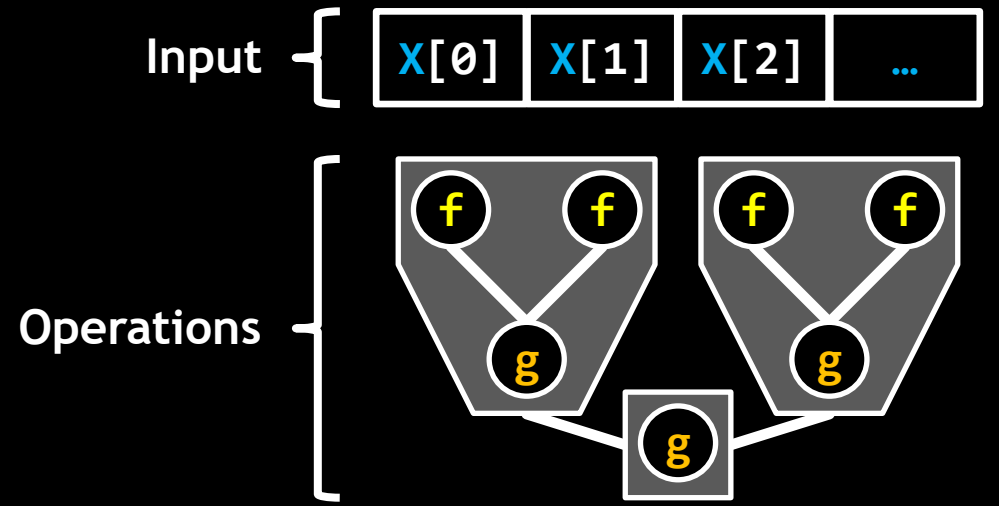
Materializes O(N) temporary storage in precious device memory.



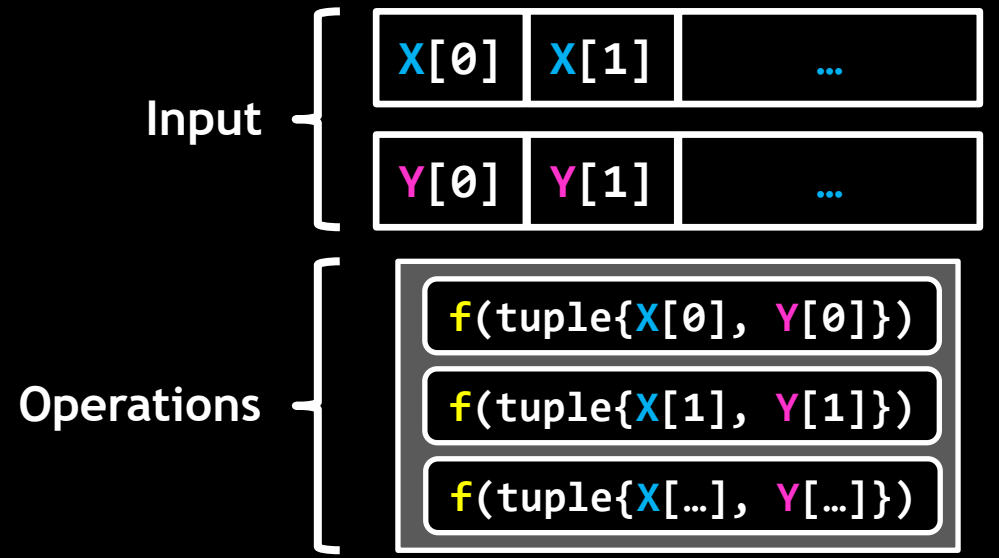


NVIDIA.

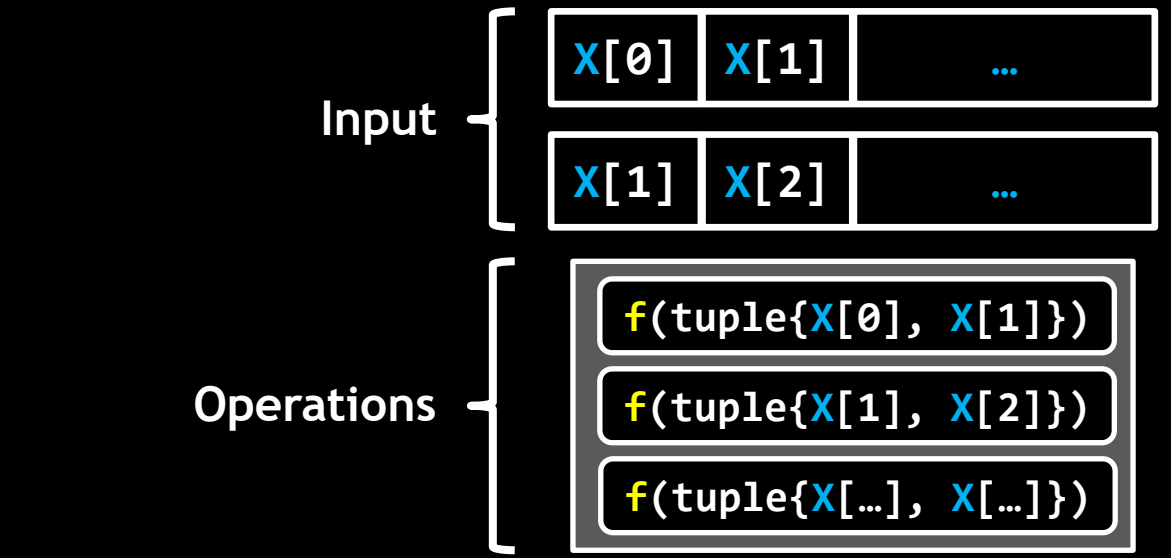
```
thrust::universal_vector X(N);

auto tmp = thrust::make_transform_iterator(
  X.begin(), f);

auto r = thrust::reduce(thrust::cuda::par,
  tmp, tmp + N, T{}, g);
```



```
thrust::universal_vector X(N), Y(N);

auto XY = thrust::make_zip_iterator(
  thrust::cuda::par, X.begin(), Y.begin());

thrust::for_each_n(
  thrust::cuda::par, XY, N, f);
```



```
thrust::universal_vector X(N);

auto adj = thrust::make_zip_iterator(
  thrust::cuda::par, X.begin(), X.begin() + 1);

thrust::for_each_n(
  thrust::cuda::par, adj, N - 1, f);
```

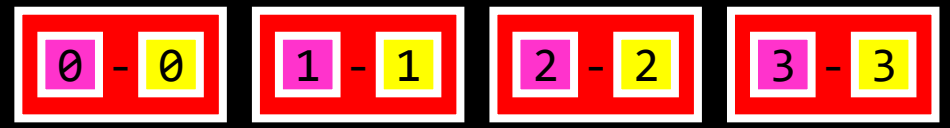


42	31		42	-	31		11	
24	22	→	24	-	22	→	2	→ 15
50	35		50	-	35		15	
A	B		A	-	B		max	

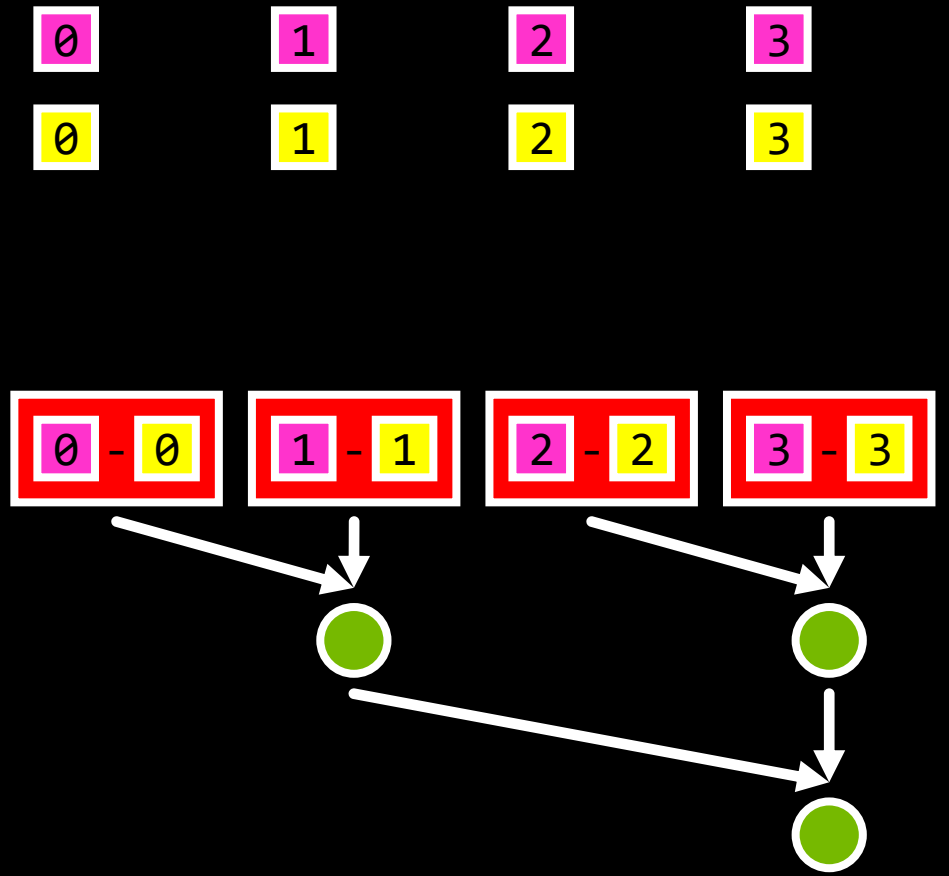
```
thrust::universal_vector<int> A(…);
thrust::universal_vector<int> B(…);
```

```
thrust::universal_vector<int> A(…);
thrust::universal_vector<int> B(…);
```

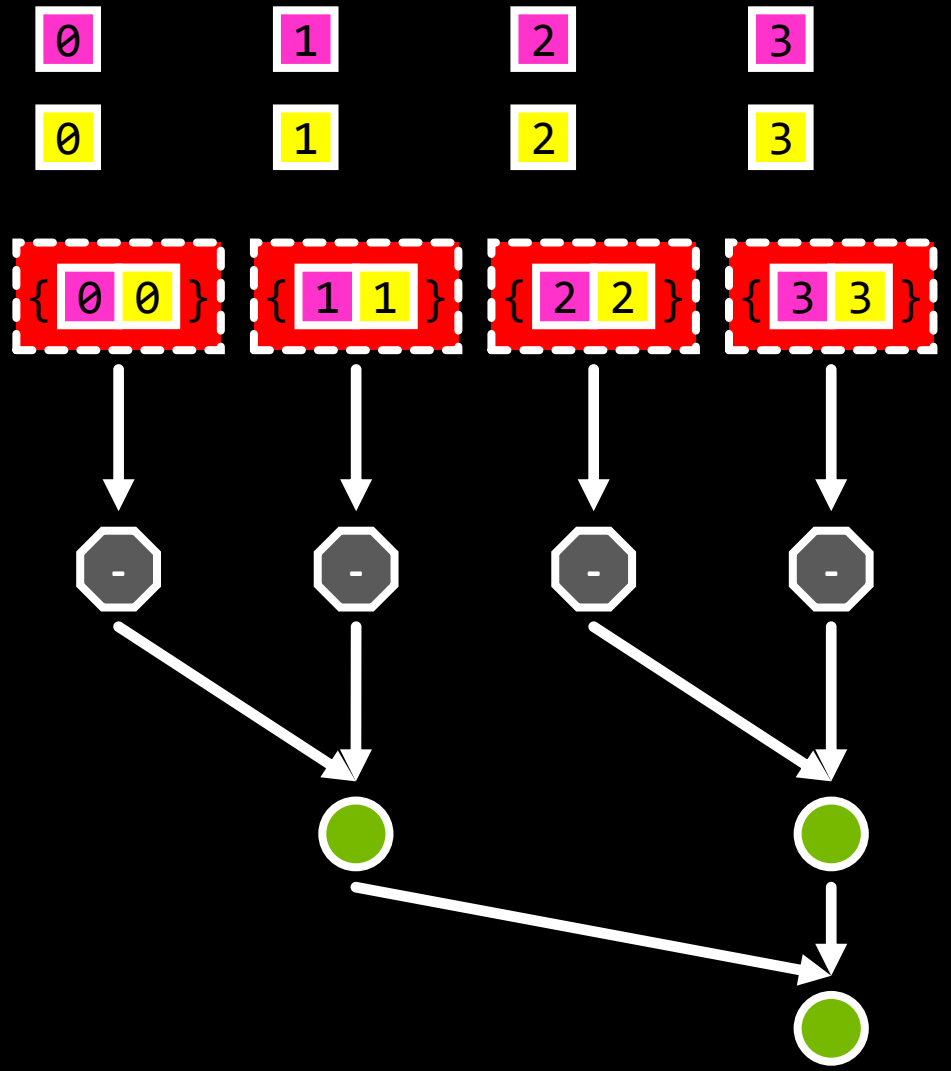

```
thrust::universal_vector<int> diffs(A.size());

thrust::transform(thrust::cuda::par,
  A.begin(), A.end(), B.begin(), diffs.begin(),
  [] __host__ __device__ (int a, int b)
  { return abs(a - b); });
```



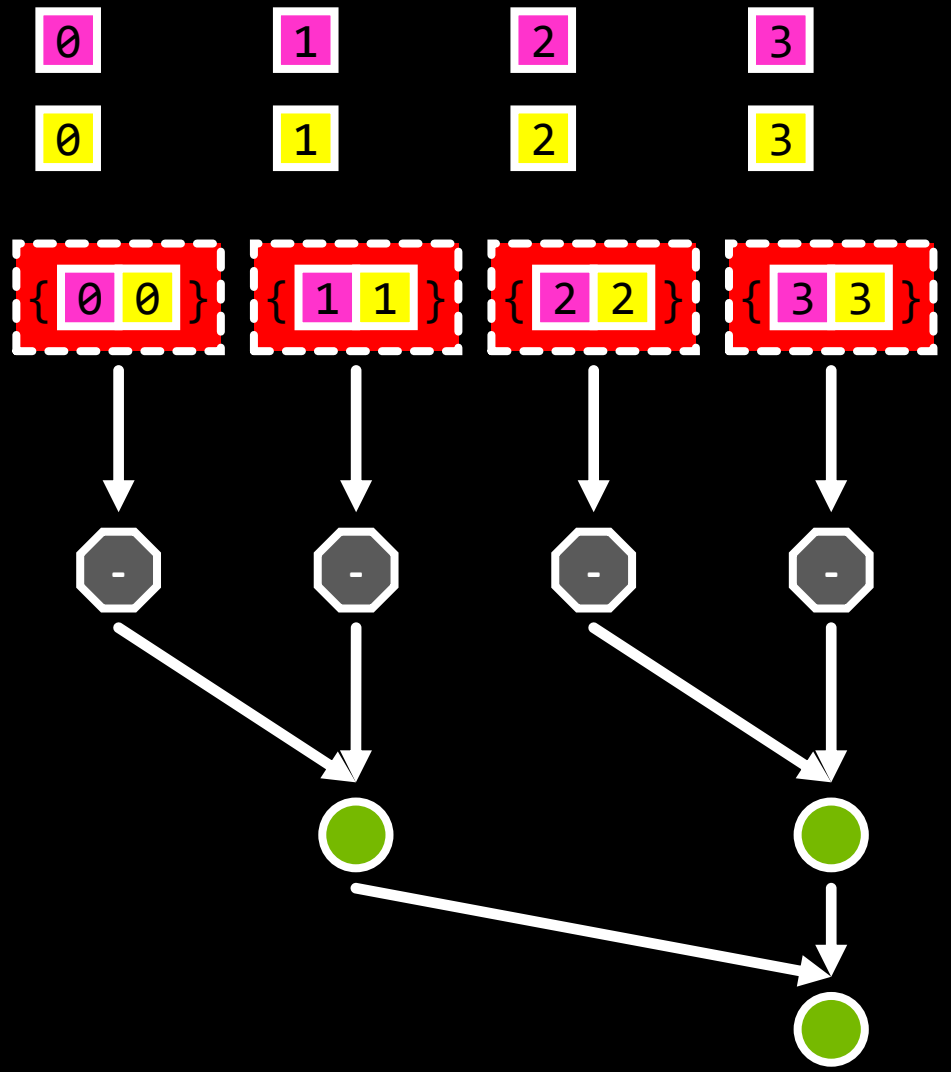
```
thrust::universal_vector<int> A(…);
thrust::universal_vector<int> B(…);


thrust::universal_vector<int> diffs(A.size());

thrust::transform(thrust::cuda::par,
  A.begin(), A.end(), B.begin(), diffs.begin(),
  [] __host__ __device__ (int a, int b)
  { return abs(a - b); });

auto max_diff = thrust::reduce(
  thrust::cuda::par,
  diffs.begin(), diffs.end(),
  0,
  cuda::maximum{});
```



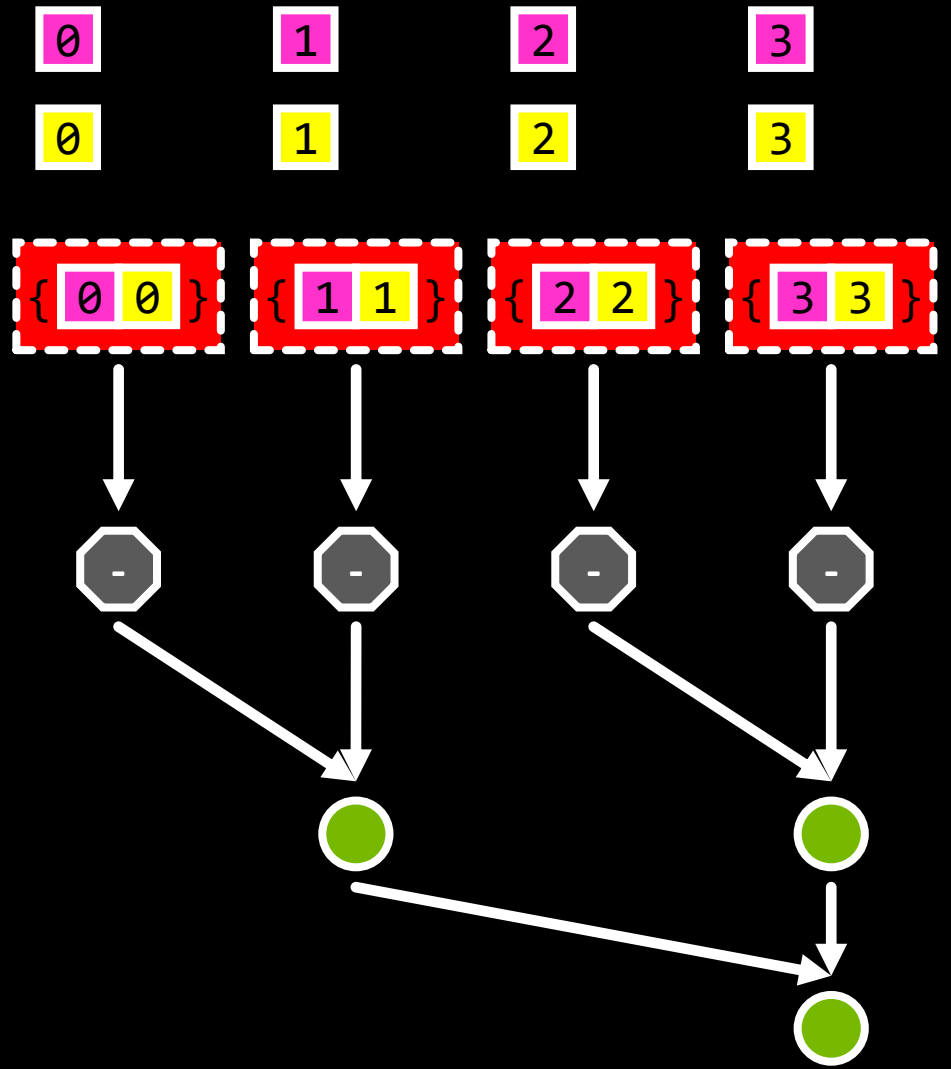
```
thrust::universal_vector<int> A(…);
thrust::universal_vector<int> B(…);


auto AB = thrust::make_zip_iterator(
  A.begin(), B.begin());

auto diffs = thrust::make_transform_iterator(
  AB.begin(),
  [] __host__ __device__
  (cuda::std::tuple<int, int> ab) {
    auto [a, b] = ab;
    return abs(a - b)
  });

auto max_diff = thrust::reduce(
  thrust::cuda::par,
  diffs, diffs + A.size(),
  0,
  cuda::maximum{});
```



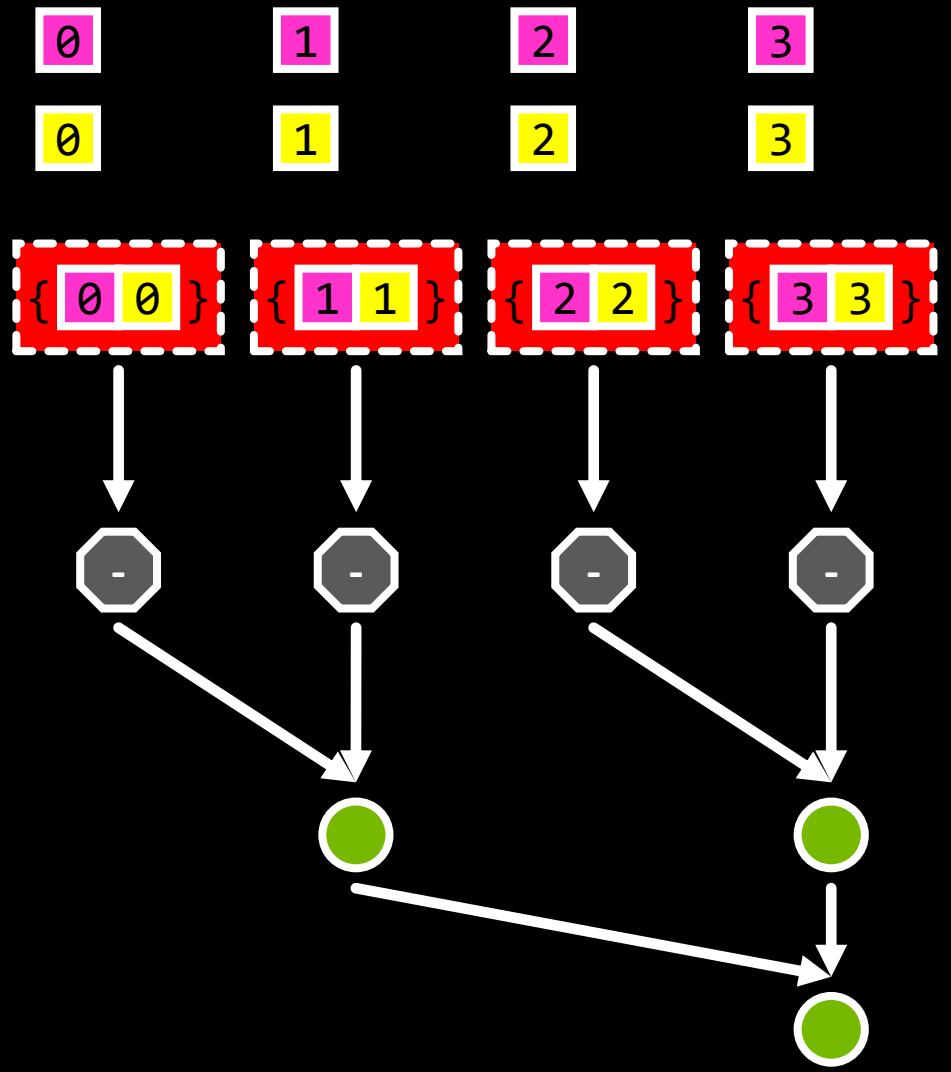
```
thrust::universal_vector<int> A(…);
thrust::universal_vector<int> B(…);


auto AB = thrust::make_zip_iterator(
  A.begin(), B.begin());

auto diffs = thrust::make_transform_iterator(
  AB.begin(),
  [] __host__ __device__
  (cuda::std::tuple<int, int> ab) {
    auto [a, b] = ab;
    return abs(a - b)
  });

auto max_diff = thrust::reduce(
  thrust::cuda::par,
  diffs, diffs + A.size(),
  0,
  cuda::maximum{});
```



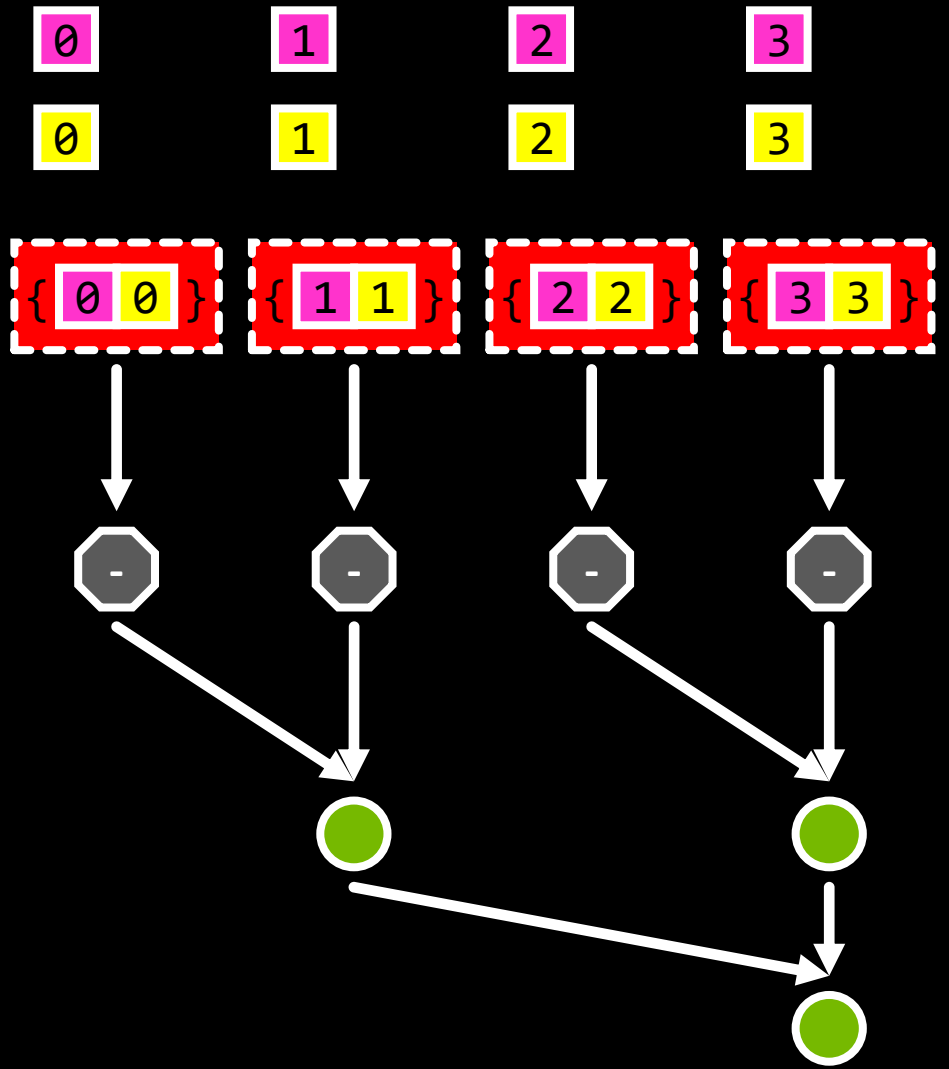
```
thrust::universal_vector<int> A(…);
thrust::universal_vector<int> B(…);


auto AB = thrust::make_zip_iterator(
  A.begin(), B.begin());

auto diffs = thrust::make_transform_iterator(
  AB.begin(),
  [] __host__ __device__
  (cuda::std::tuple<int, int> ab) {
    auto [a, b] = ab;
    return abs(a - b)
  });

auto max_diff = thrust::reduce(
  thrust::cuda::par,
  diffs, diffs + A.size(),
  0,
  cuda::maximum{});
```

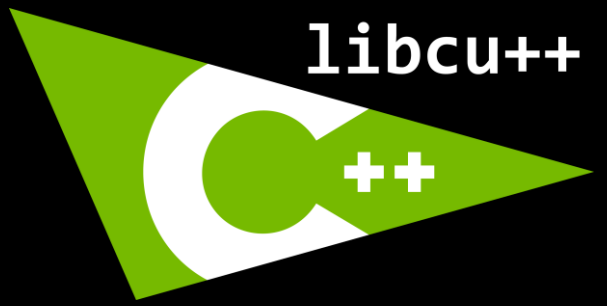


```
thrust::universal_vector<int> A(…);
thrust::universal_vector<int> B(…);


auto AB = thrust::make_zip_iterator(
  A.begin(), B.begin());

auto max_diff = thrust::transform_reduce(
  thrust::cuda::par,
  AB, AB + A.size(),
  [] __host__ __device__
  (cuda::std::tuple<int, int> ab) {
    auto [a, b] = ab;
    return abs(a - b)
  },
  0,
  cuda::maximum{});
```



```
thrust::universal_vector<int> A(…);
thrust::universal_vector<int> B(…);


auto AB = thrust::make_zip_iterator(
  A.begin(), B.begin());

auto max_diff = thrust::transform_reduce(
  thrust::cuda::par,
  AB, AB + A.size(),
  [] __host__ __device__
  (cuda::std::tuple<int, int> ab) {
    auto [a, b] = ab;
    return abs(a - b)
  },
  0,
  cuda::maximum{});
```

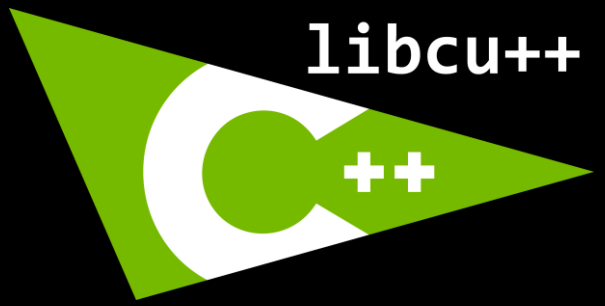




The CUDA C++ Foundational Library

https://nvidia.github.io/cccl/libcudacxx

Learn More: S72575 How You Should Write a CUDA C++ Kernel

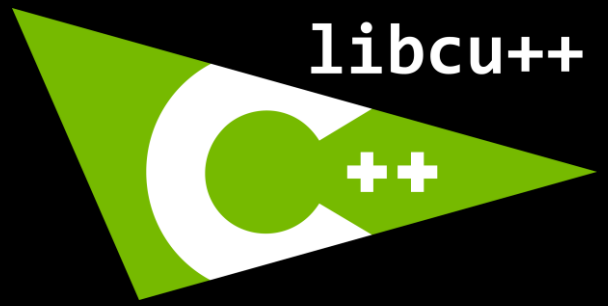


The CUDA C++ Foundational Library

https://nvidia.github.io/cccl/libcudacxx

Learn More: S72575 How You Should Write a CUDA C++ Kernel

Host Compiler's Standard Library (GCC, LLVM, MSVC, etc)	
`#include <…>`	Standard C++, `__host__` only.
`std::`	Complete, strictly conforming to Standard C++.

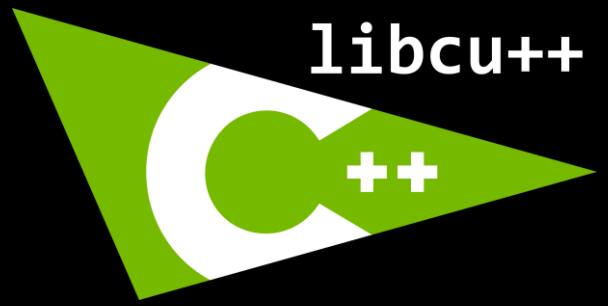


The CUDA C++ Foundational Library

https://nvidia.github.io/cccl/libcudacxx

Learn More: S72575 How You Should Write a CUDA C++ Kernel

Host Compiler's Standard Library (GCC, LLVM, MSVC, etc)	
`#include <…>`	Standard C++, `__host__` only.
`std::`	Complete, strictly conforming to Standard C++.
`#include <cuda/std/…>`	CUDA C++, `__host__ __device__`.
`cuda::std::`	Subset, strictly conforming to Standard C++.
`libcu++`	



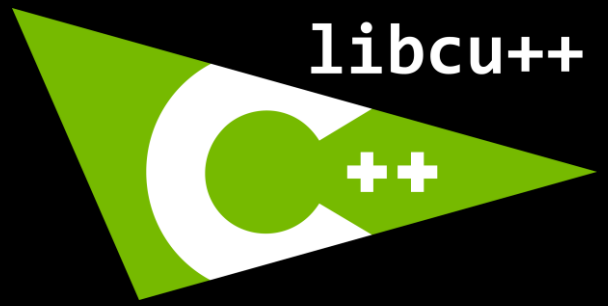
The CUDA C++ Foundational Library

https://nvidia.github.io/cccl/libcudacxx

Learn More: S72575 How You Should Write a CUDA C++ Kernel

Host Compiler's Standard Library (GCC, LLVM, MSVC, etc)	
`#include <…>`	Standard C++, `__host__` only.
`std::`	Complete, strictly conforming to Standard C++.
`#include <cuda/std/…>`	CUDA C++, `__host__ __device__`.
`cuda::std::`	Subset, strictly conforming to Standard C++.
`#include <cuda/…>`	CUDA C++, `__host__` and/or `__device__`.
`cuda::`	Modern C++ API for CUDA & Standard C++ extensions.
`libcu++`	

NVIDIA



The CUDA C++ Foundational Library

https://nvidia.github.io/cccl/libcudacxx

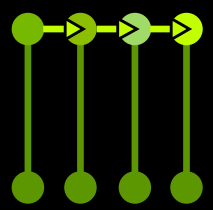
Learn More: S72575 How You Should Write a CUDA C++ Kernel

Host Compiler's Standard Library (GCC, LLVM, MSVC, etc)	
`#include <…>`	Standard C++, `__host__` only.
`std::`	Complete, strictly conforming to Standard C++.
`#include <cuda/std/…>`	CUDA C++, `__host__ __device__`.
`cuda::std::`	Subset, strictly conforming to Standard C++.
`#include <cuda/…>`	CUDA C++, `__host__` and/or `__device__`.
`cuda::`	Modern C++ API for CUDA & Standard C++ extensions.
`#include <cuda/experimental/…>`	CUDA C++, `__host__` and/or `__device__`.
`cuda::experimental:: (cudax::)`	Beta features.
libcu++	

NVIDIA.

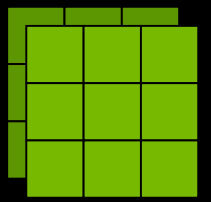
The C++ Parallel Algorithms Library

https://nvidia.github.io/cccl/thrust



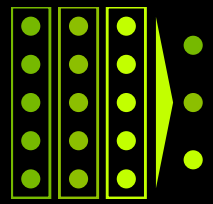
Standard Algorithms

- thrust::transform_reduce
- thrust::inclusive_scan
- thrust::sort
- thrust::copy
- ...



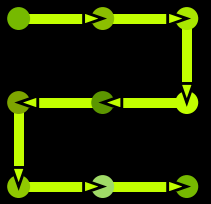
Containers

- thrust::device_vector
- thrust::host_vector
- thrust::universal_vector
- thrust::allocate_unique
- ...



Extended Algorithms

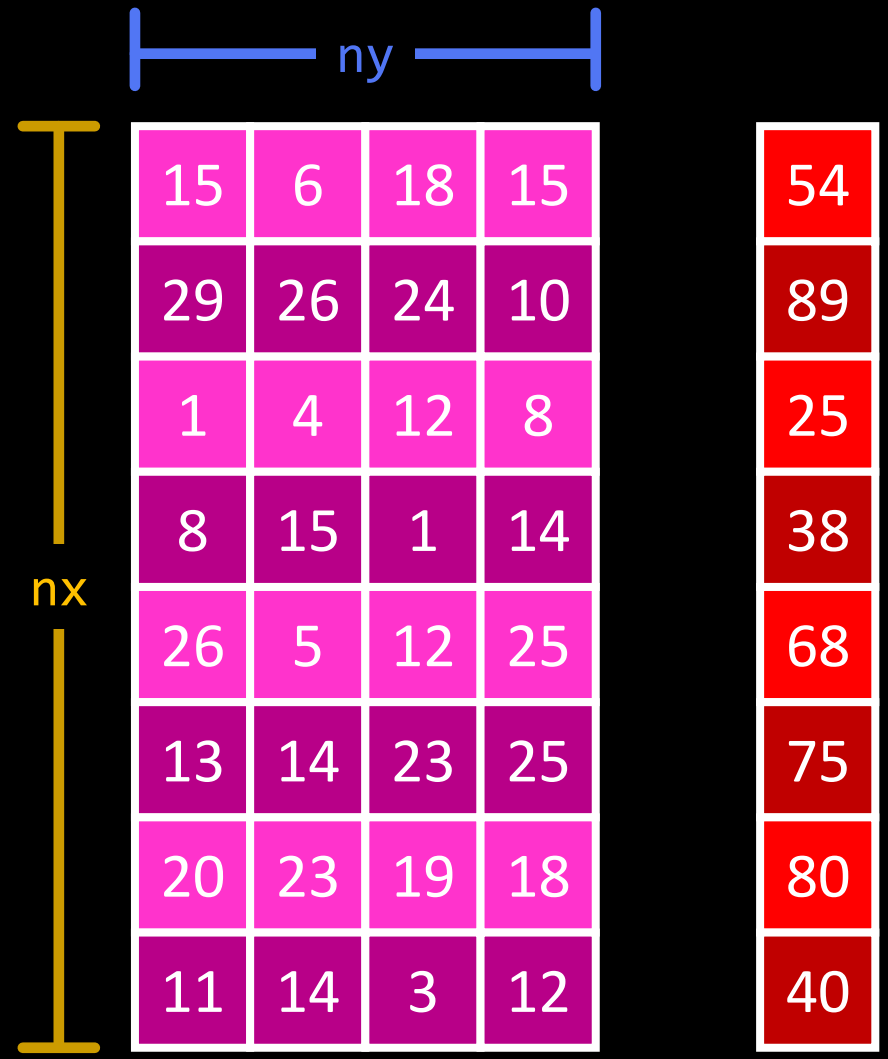
- thrust::reduce_by_key
- thrust::sort_by_key
- thrust::tabulate
- thrust::gather
- ...



Iterators

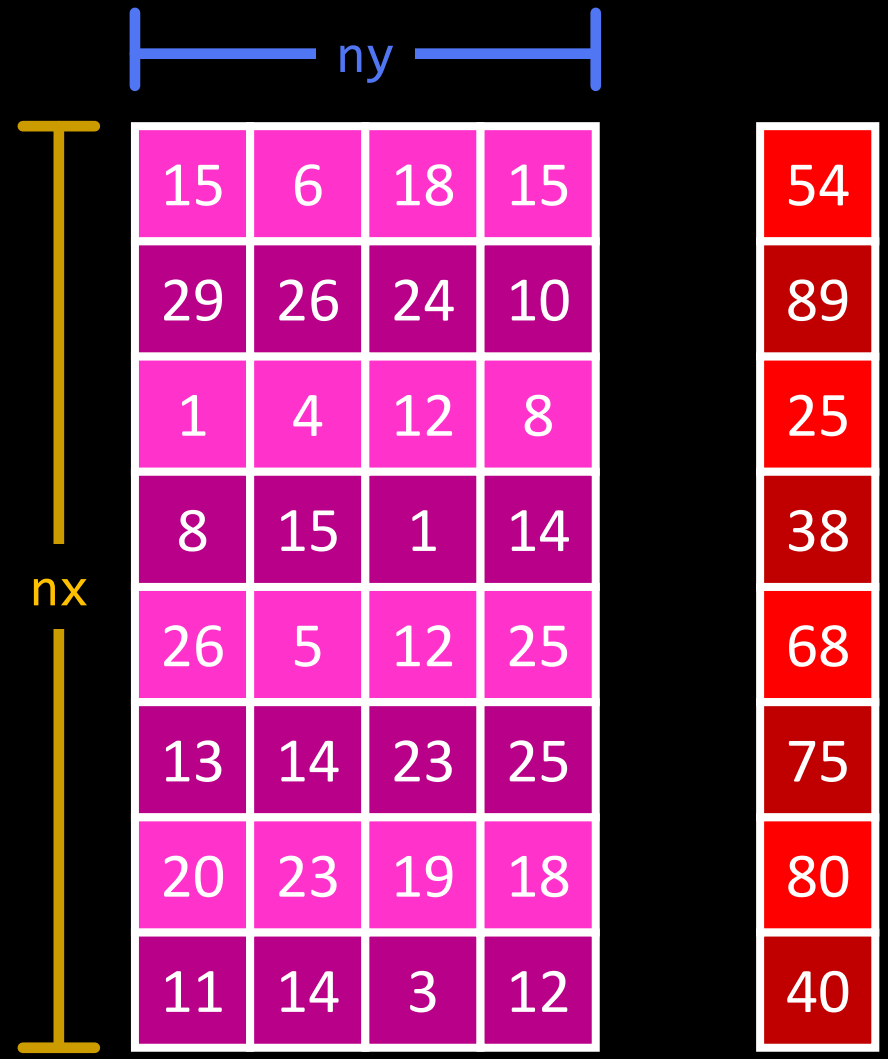
- thrust::counting_iterator
- thrust::constant_iterator
- thrust::transform_iterator
- thrust::zip_iterator
- ...

```
int nx(…), ny(…);
thrust::universal_vector<float> M(nx * ny);
```



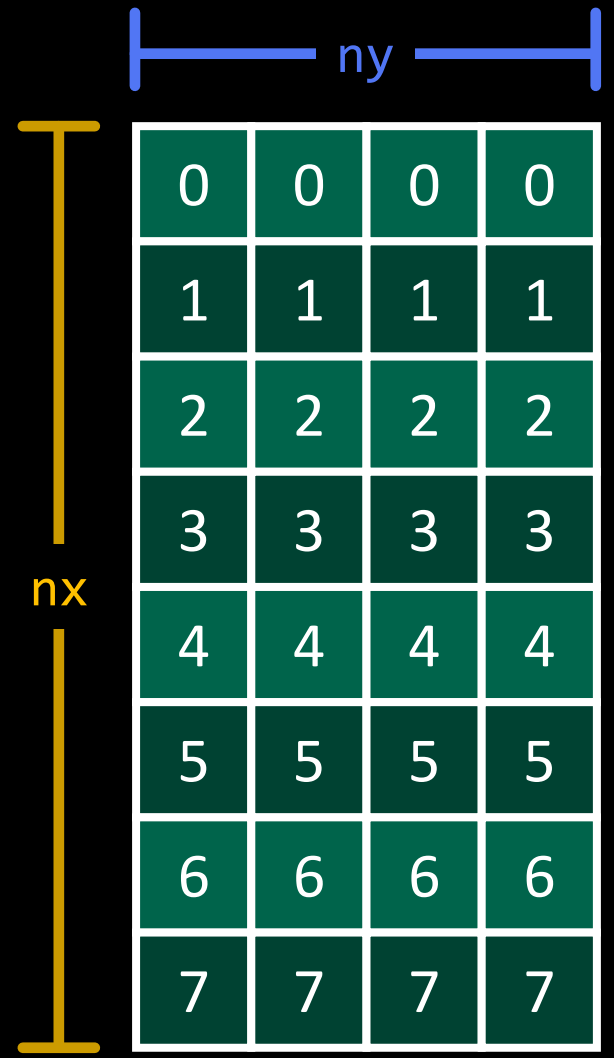
```
int nx(…), ny(…);
thrust::universal_vector<float> M(nx * ny);
```

Each set of consecutive keys that compare equal defines a group.



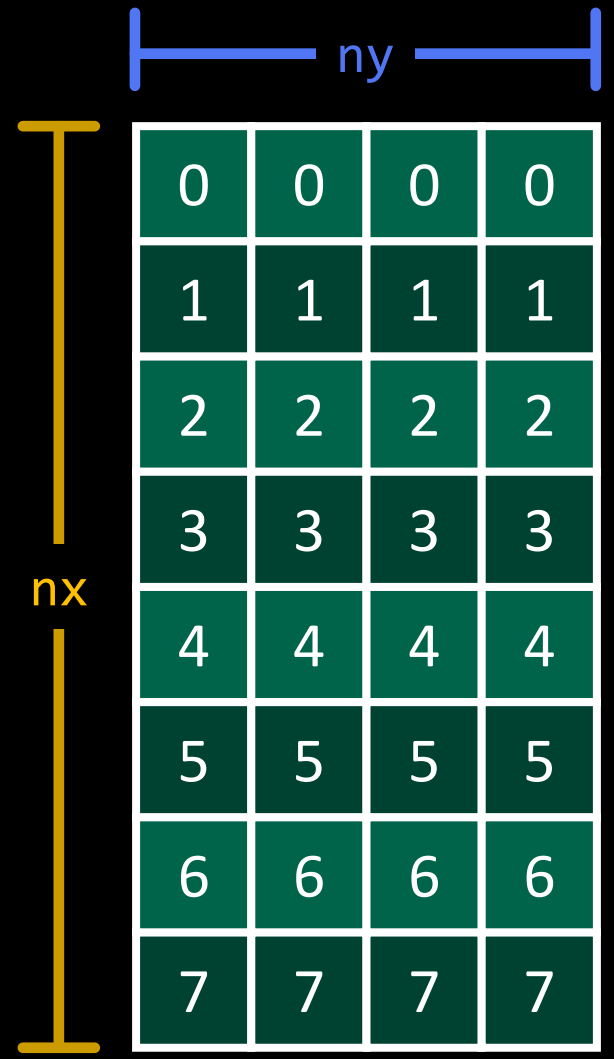
```
int nx(…), ny(…);
thrust::universal_vector<float> M(nx * ny);
```

Each set of consecutive keys that compare equal defines a group.

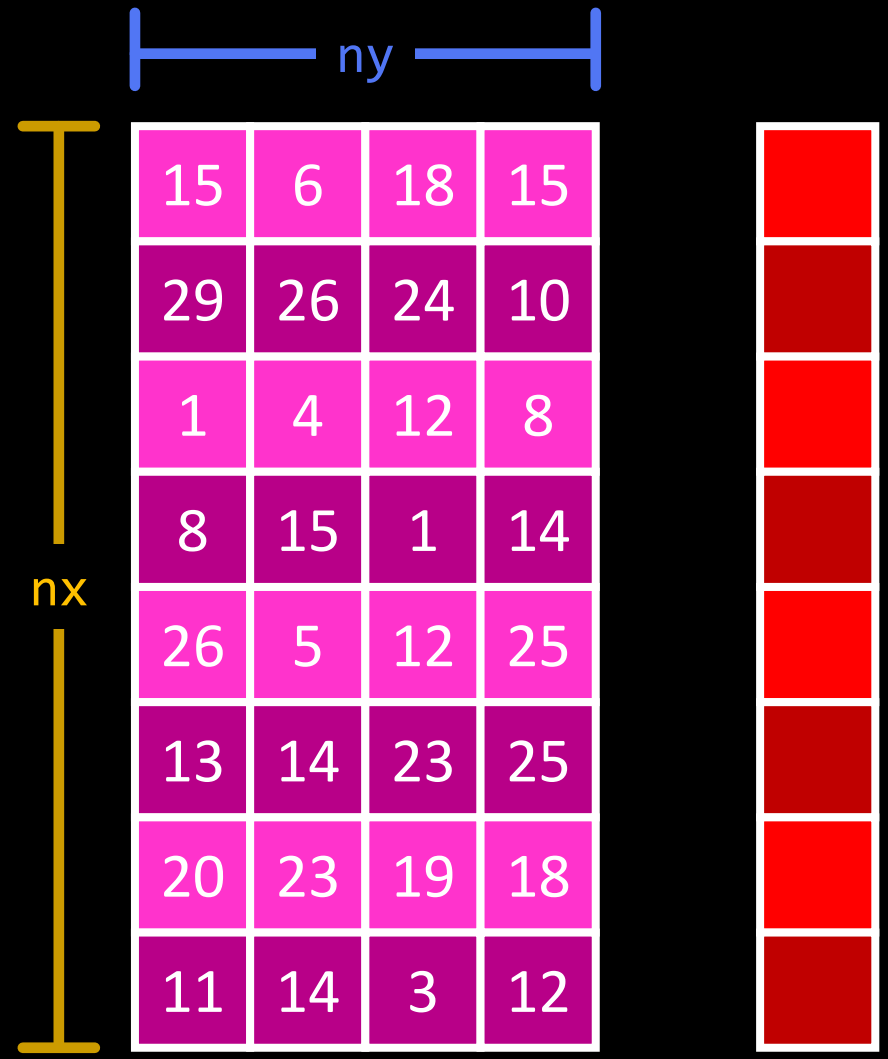


```
int nx(…), ny(…);
thrust::universal_vector<float> M(nx * ny);

auto row_ids_begin =
  thrust::make_transform_iterator(
    thrust::make_counting_iterator(0),
    [=] __host__ __device__ (int x) {
      return x / ny;
    });
auto row_ids_end = row_ids_begin + nx * ny;
```

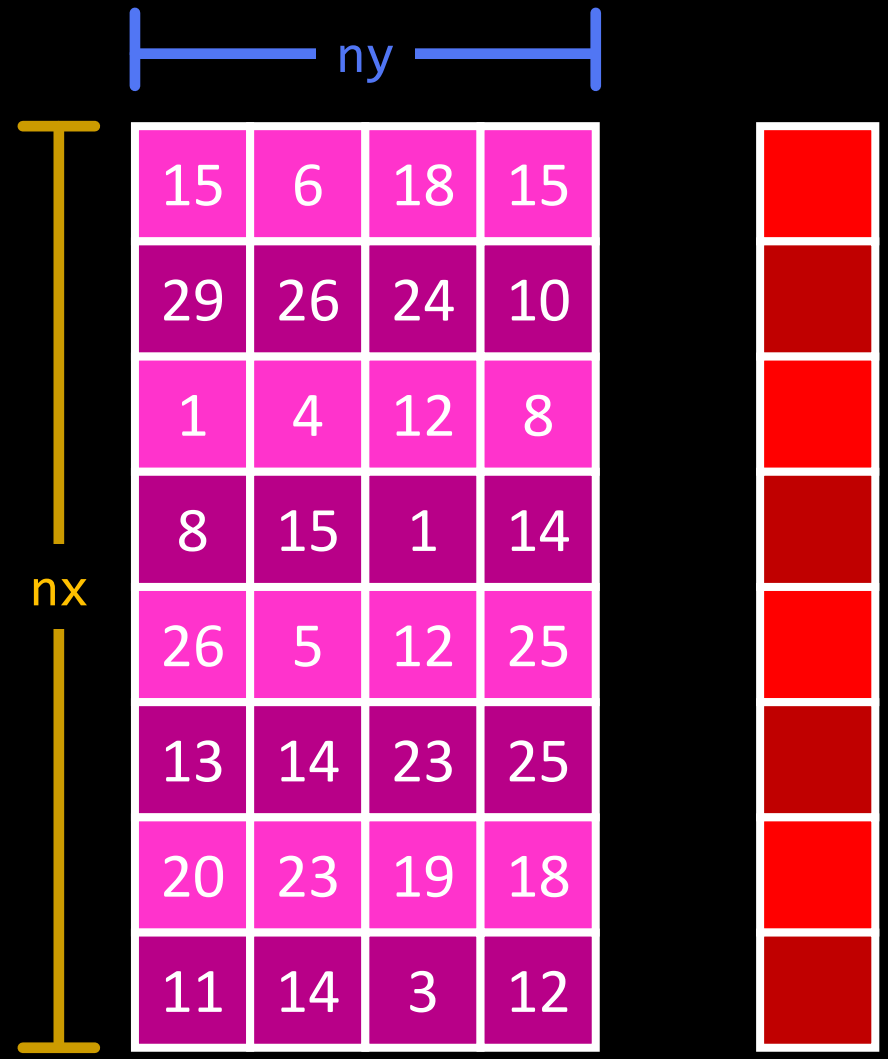
Each set of consecutive keys that compare equal defines a group.



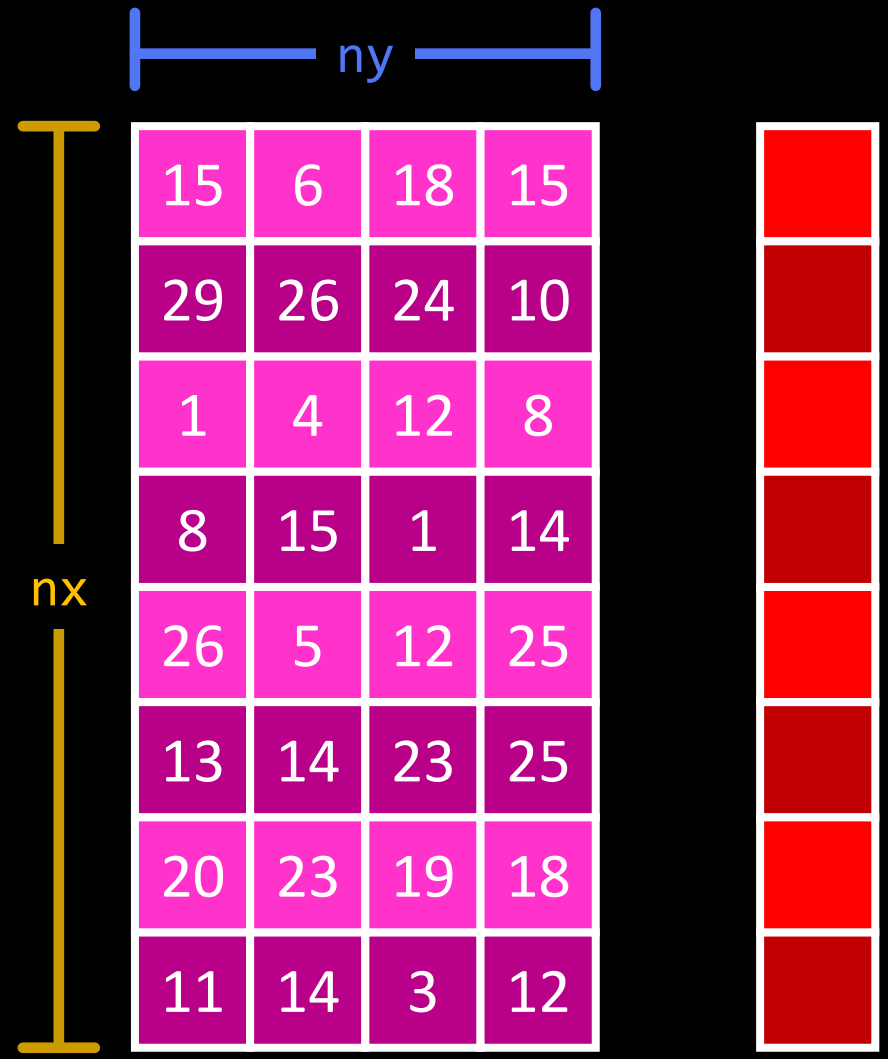
```
int nx(…), ny(…);
thrust::universal_vector<float> M(nx * ny);

auto row_ids_begin =
  thrust::make_transform_iterator(
    thrust::make_counting_iterator(0),
    [=] __host__ __device__ (int x) {
      return x / ny;
    });
auto row_ids_end = row_ids_begin + nx * ny;

thrust::universal_vector<float> sums(nx);
```



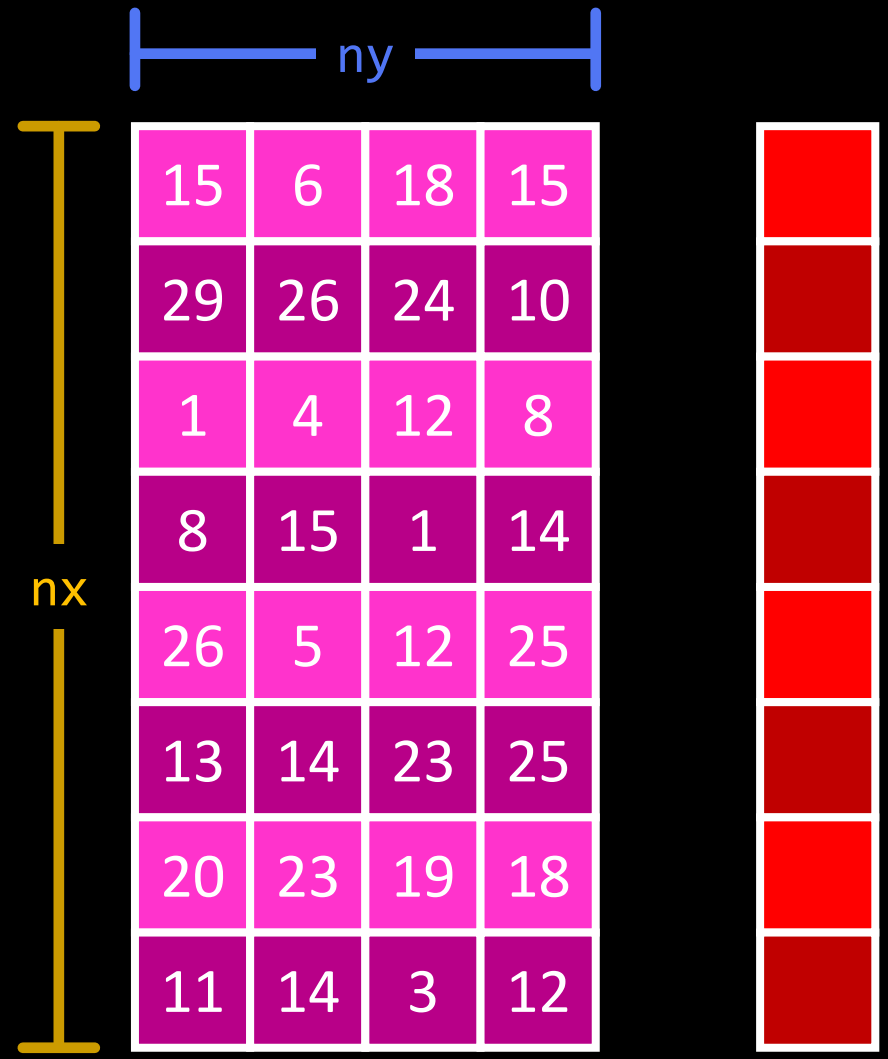
```
int nx(…), ny(…);
thrust::universal_vector<float> M(nx * ny);

auto row_ids_begin =
  thrust::make_transform_iterator(
    thrust::make_counting_iterator(0),
    [=] __host__ __device__ (int x) {
      return x / ny;
    });
auto row_ids_end = row_ids_begin + nx * ny;

thrust::universal_vector<float> sums(nx);

thrust::reduce_by_key(thrust::cuda::par,
  …);
```



```cpp
int nx(…), ny(…);
thrust::universal_vector<float> M(nx * ny);

auto row_ids_begin =
  thrust::make_transform_iterator(
    thrust::make_counting_iterator(0),
    [=] __host__ __device__ (int x) {
      return x / ny;
    });
auto row_ids_end = row_ids_begin + nx * ny;

thrust::universal_vector<float> sums(nx);

thrust::reduce_by_key(thrust::cuda::par,
  // Keys input.
  …);
```

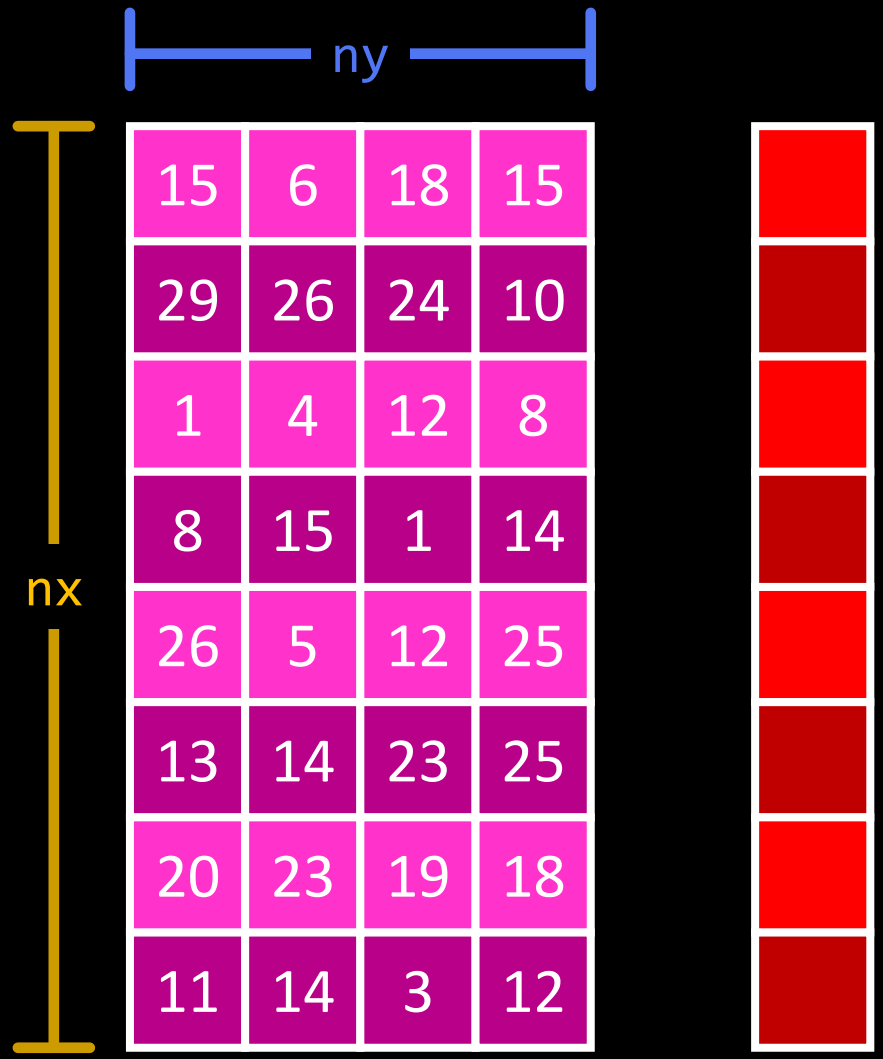


```
int nx(…), ny(…);
thrust::universal_vector<float> M(nx * ny);

auto row_ids_begin =
  thrust::make_transform_iterator(
    thrust::make_counting_iterator(0),
    [=] __host__ __device__ (int x) {
      return x / ny;
    });
auto row_ids_end = row_ids_begin + nx * ny;

thrust::universal_vector<float> sums(nx);

thrust::reduce_by_key(thrust::cuda::par,
  row_ids_begin, row_ids_end,
  // Values input.
  …);
```

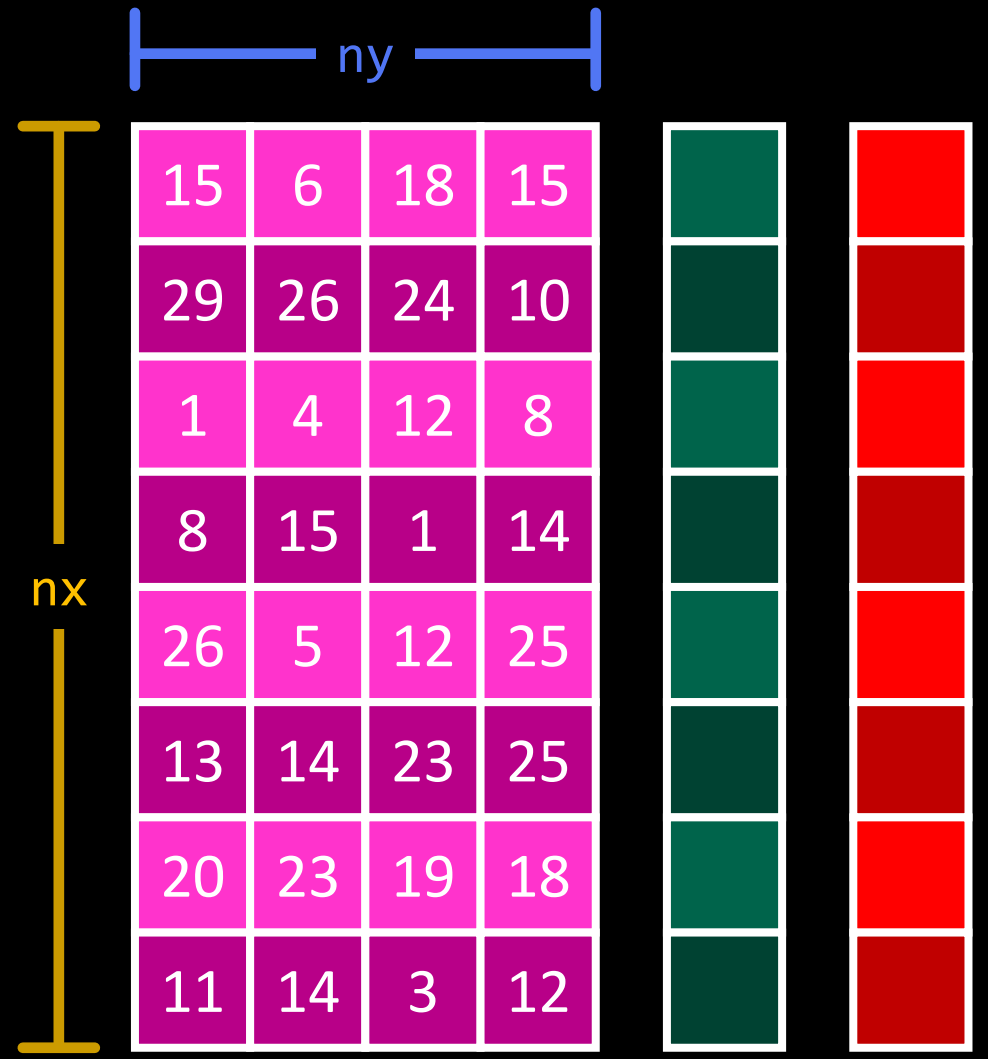


```
int nx(…), ny(…);
thrust::universal_vector<float> M(nx * ny);

auto row_ids_begin =
  thrust::make_transform_iterator(
    thrust::make_counting_iterator(0),
    [=] __host__ __device__ (int x) {
      return x / ny;
    });
auto ro

thrust:

thrust::reduce_by_key(thrust::cuda::par,
  row_ids_begin, row_ids_end,
  M.begin(),
  …);
```

The underlying algorithm is applied to the values in each group.

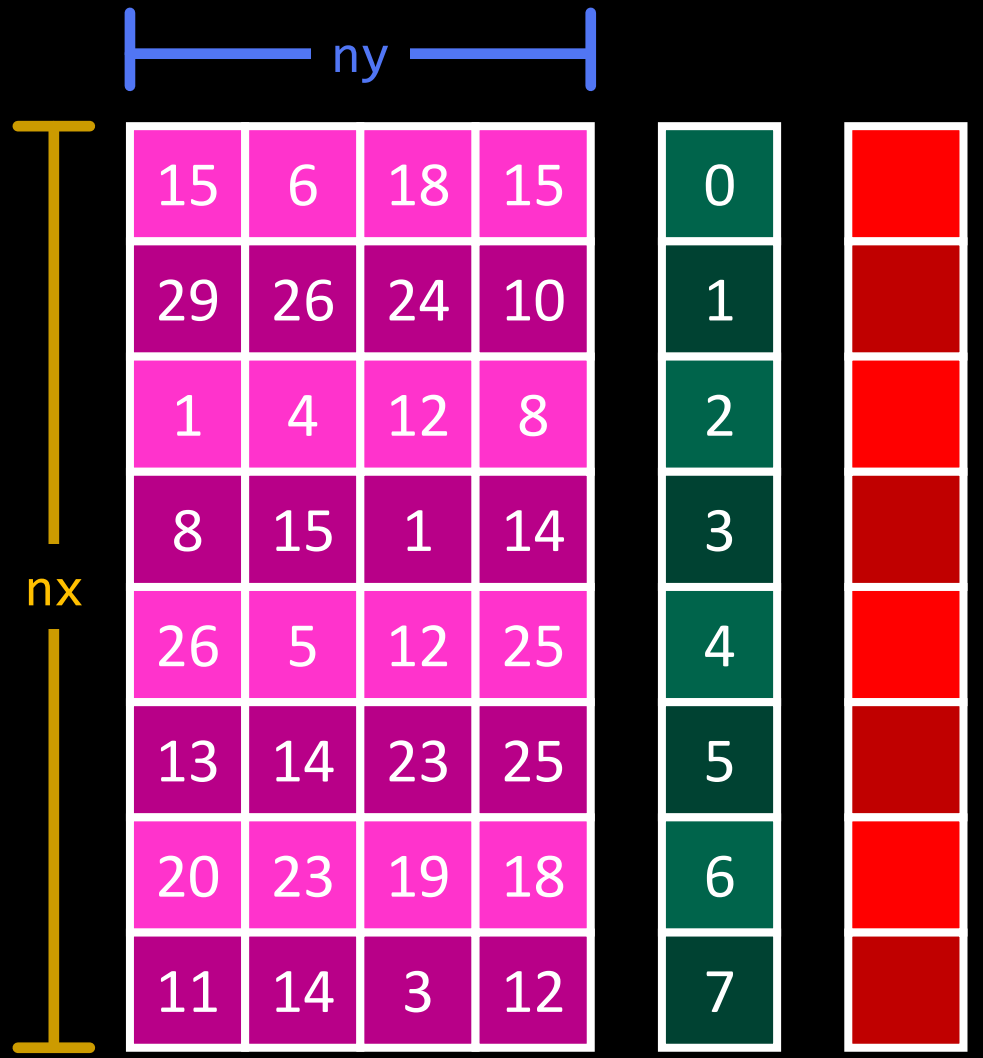


```cpp
int nx(…), ny(…);
thrust::universal_vector<float> M(nx * ny);

auto row_ids_begin =
  thrust::make_transform_iterator(
    thrust::make_counting_iterator(0),
    [=] __host__ __device__ (int x) {
      return x / ny;
    });
auto row_ids_end = row_ids_begin + nx * ny;

thrust::universal_vector<float> sums(nx);

thrust::reduce_by_key(thrust::cuda::par,
  row_ids_begin, row_ids_end,
  M.begin(),
  // Keys output.
  …);
```



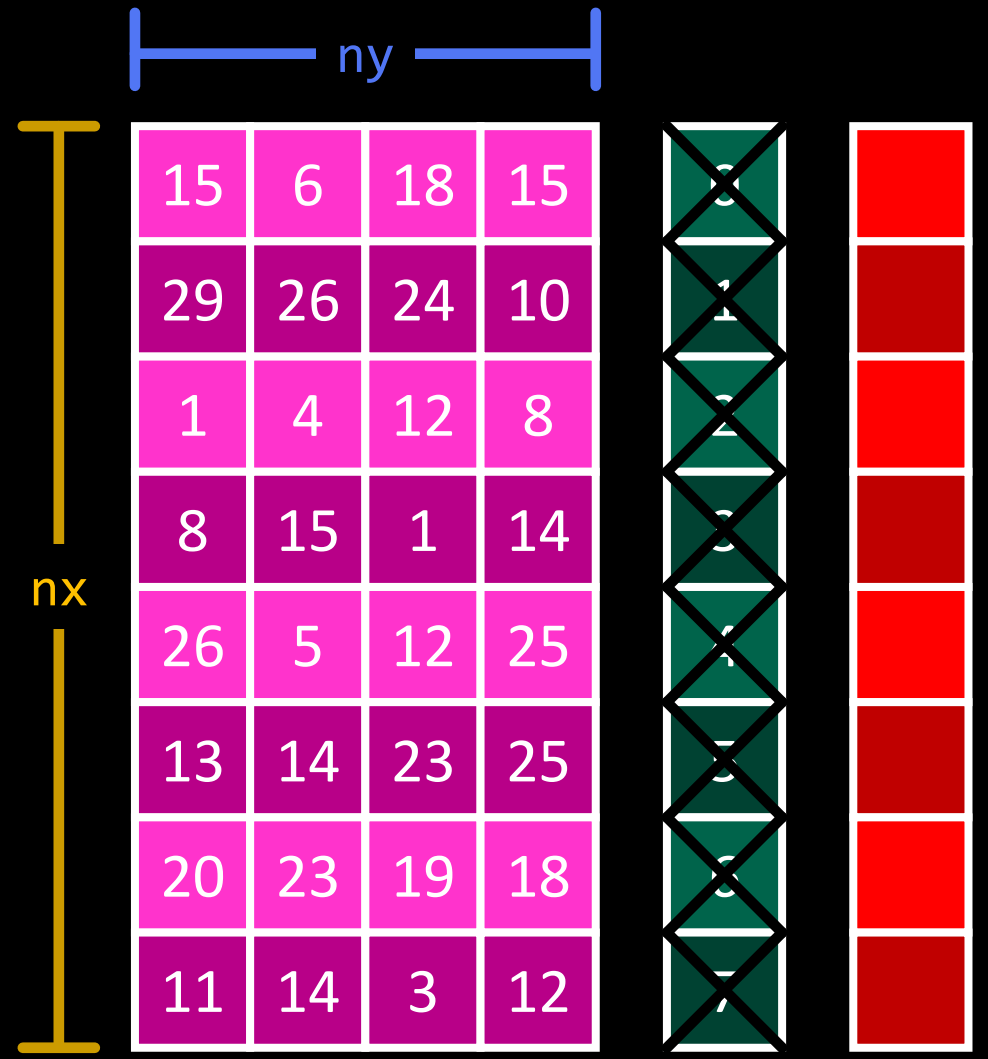
```
int nx(…), ny(…);
thrust::universal_vector<float> M(nx * ny);

auto row_ids_begin =
  thrust::make_transform_iterator(
    thrust::make_counting_iterator(0),
    [=] __host__ __device__ (int x) {
      return x / ny;
    });
auto ro

thrust:

thrust::reduce_by_key(thrust::cuda::par,
  row_ids_begin, row_ids_end,
  M.begin(),
  // Keys output.
  …);
```

The first key in each group is copied to the key output range.

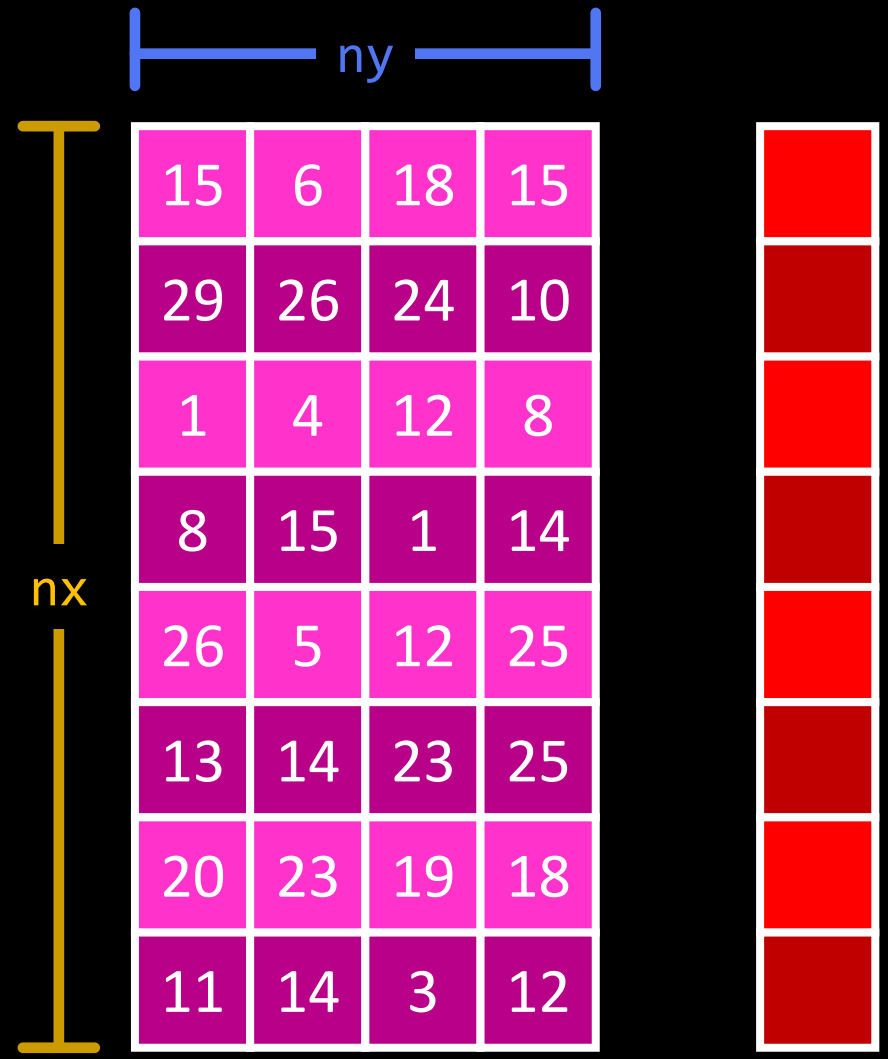


```
int nx(…), ny(…);
thrust::universal_vector<float> M(nx * ny);

auto row_ids_begin =
  thrust::make_transform_iterator(
    thrust::make_counting_iterator(0),
    [=] __host__ __device__ (int x) {
      return x / ny;
    });
auto ro

thrust:

thrust::reduce_by_key(thrust::cuda::par,
  row_ids_begin, row_ids_end,
  M.begin(),
  thrust::make_discard_iterator(),
  …);
```

The first key in each group is copied to the key output range.

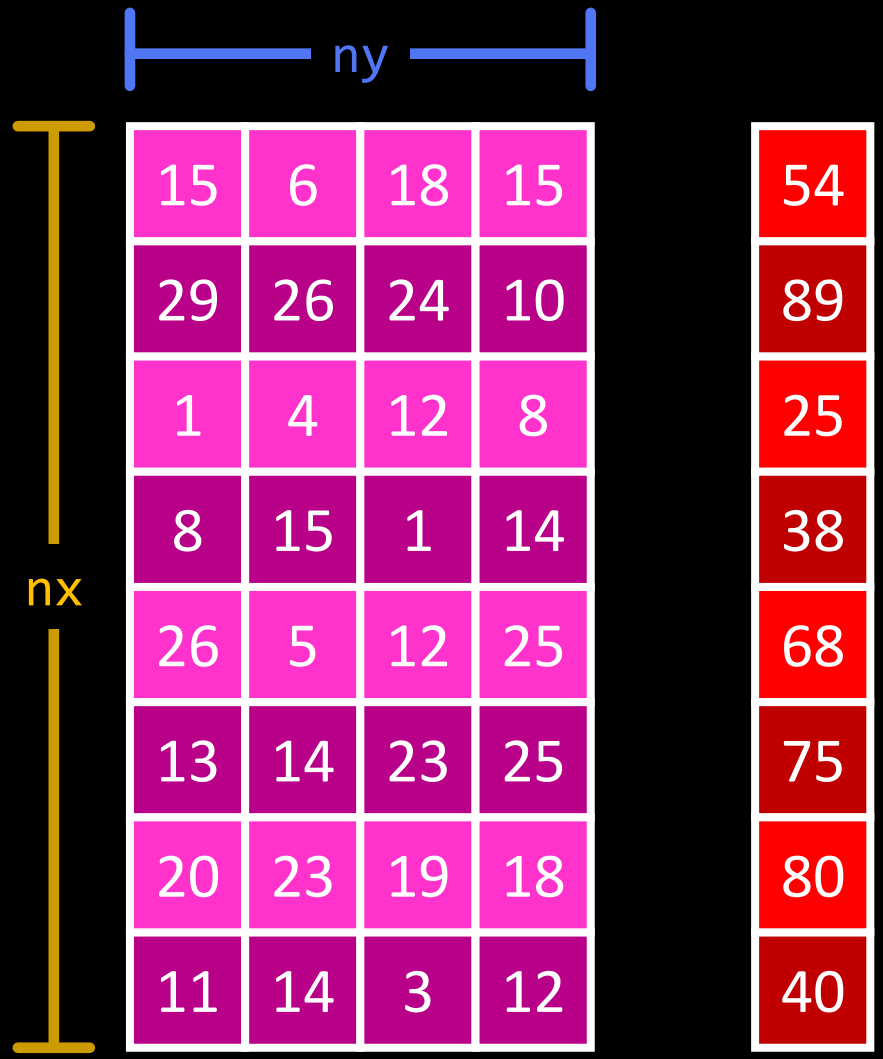


```cpp
int nx(…), ny(…);
thrust::universal_vector<float> M(nx * ny);

auto row_ids_begin =
  thrust::make_transform_iterator(
    thrust::make_counting_iterator(0),
    [=] __host__ __device__ (int x) {
      return x / ny;
    });
auto row_ids_end = row_ids_begin + nx * ny;

thrust::universal_vector<float> sums(nx);

thrust::reduce_by_key(thrust::cuda::par,
  row_ids_begin, row_ids_end,
  M.begin(),
  thrust::make_discard_iterator(),
  // Values output.);
```

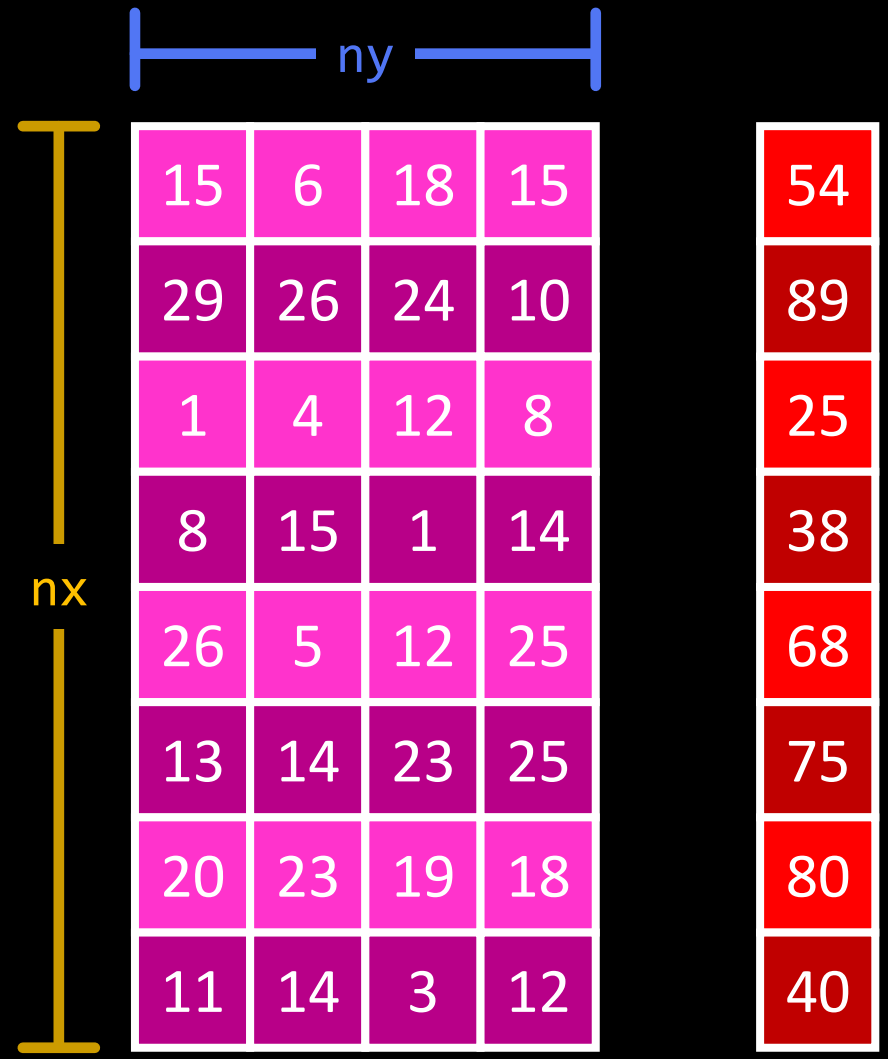


```
int nx(…), ny(…);
thrust::universal_vector<float> M(nx * ny);

auto row_ids_begin =
  thrust::make_transform_iterator(
    thrust::make_counting_iterator(0),
    [=] __host__ __device__ (int x) {
      return x / ny;
    });
auto ro

thrust:

thrust::reduce_by_key(thrust::cuda::par,
  row_ids_begin, row_ids_end,
  M.begin(),
  thrust::make_discard_iterator(),
  sums.begin());
```

The algorithm result for each group is written to the values output.

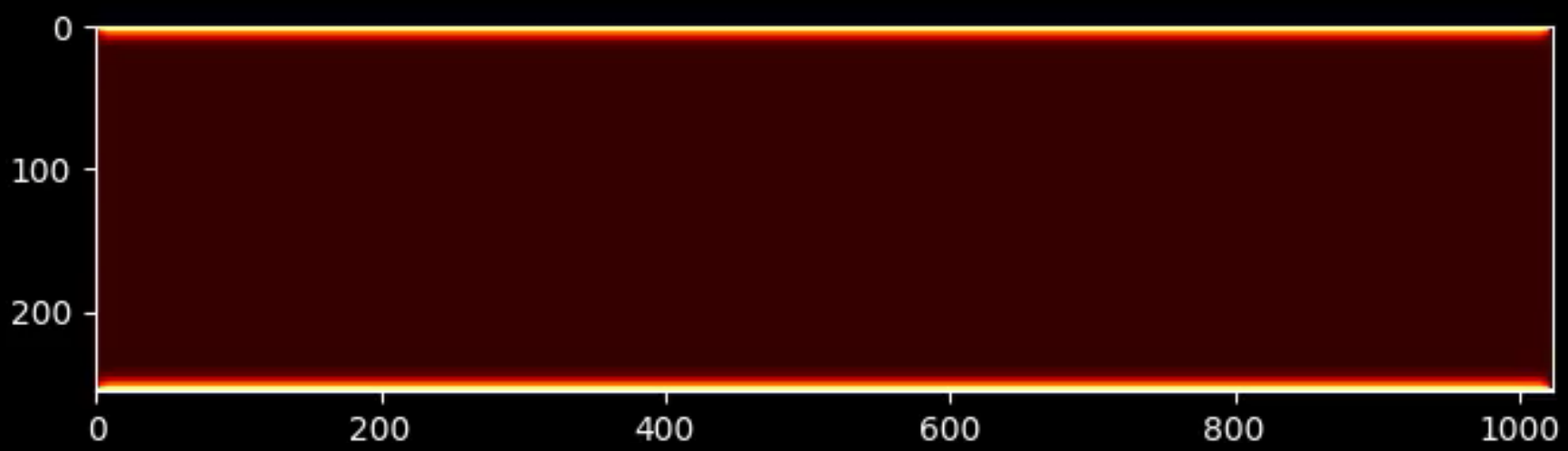


```cpp
int nx(…), ny(…);
thrust::universal_vector<float> M(nx * ny);

auto row_ids_begin =
  thrust::make_transform_iterator(
    thrust::make_counting_iterator(0),
    [=] __host__ __device__ (int x) {
      return x / ny;
    });
auto row_ids_end = row_ids_begin + nx * ny;

thrust::universal_vector<float> sums(nx);

thrust::reduce_by_key(thrust::cuda::par,
  row_ids_begin, row_ids_end,
  M.begin(),
  thrust::make_discard_iterator(),
  sums.begin());
```

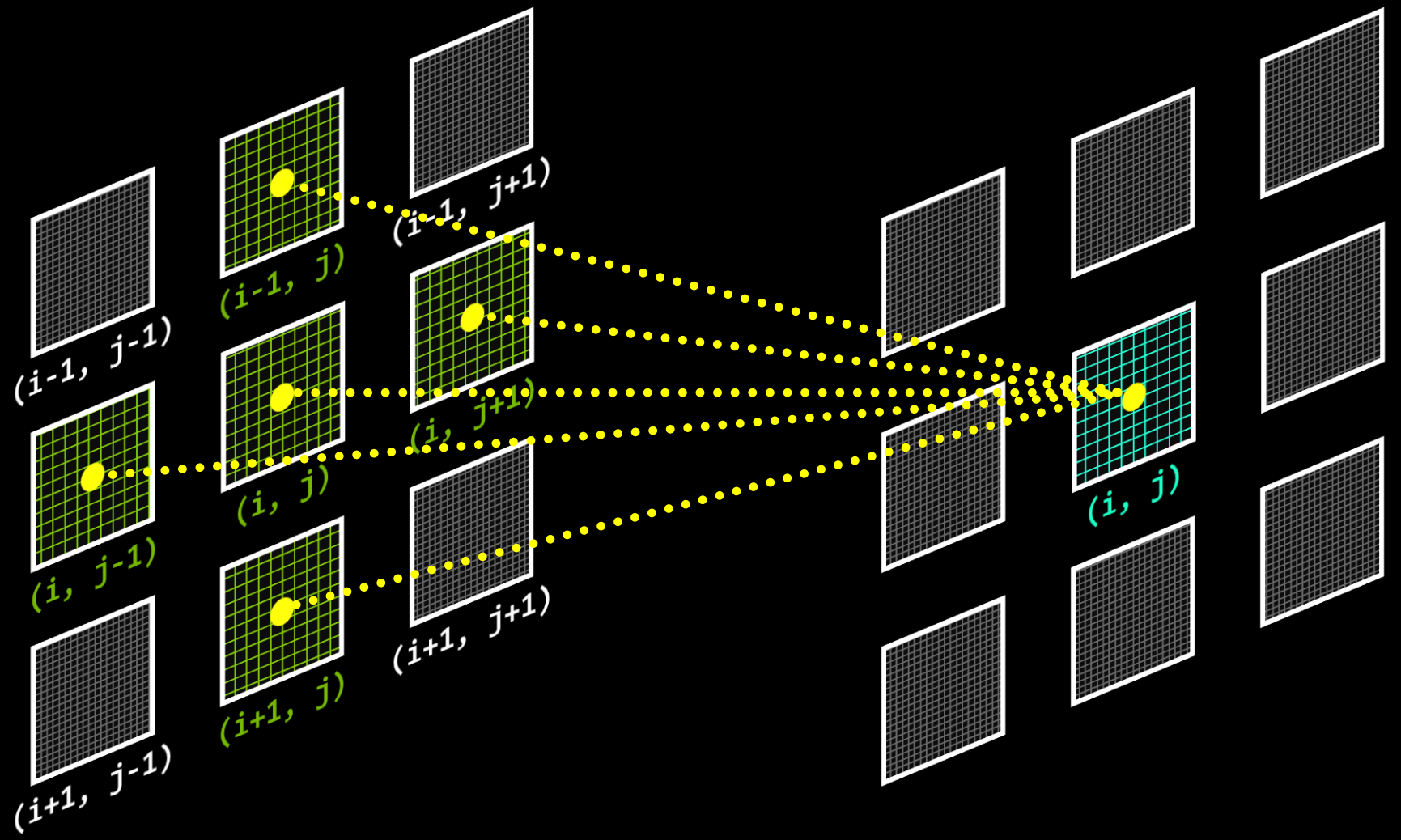


By Key Algorithms

- `reduce_by_key`
- `(inclusive|exclusive)_scan_by_key`
- `sort_by_key`
- `unique_by_key`
- `merge_by_key`
- `set_(union|difference|intersection)_by_key`



NVIDIA.



```
void heat_equation(auto policy, int nx, int ny,
                   const thrust::universal_vector<float>& U_data,
                   thrust::universal_vector<float>&       V_data);
```

```
void heat_equation(auto policy, int nx, int ny,
                   const thrust::universal_vector<float>& U_data,
                   thrust::universal_vector<float>&       V_data) {
  …

  thrust::for_each_n(policy, thrust::make_counting_iterator(0), U.size(), …);
}
```

```
void heat_equation(auto policy, int nx, int ny,
                   const thrust::universal_vector<float>& U_data,
                   thrust::universal_vector<float>&       V_data) {
  …

  thrust::for_each_n(policy, thrust::make_counting_iterator(0), U.size(),
    [?, nx, ny] __host__ __device__ (int xy) {
      …
    });
}
```

```
void heat_equation(auto policy, int nx, int ny,
                   const thrust::universal_vector<float>& U_data,
                   thrust::universal_vector<float>&       V_data) {
  const float* U = thrust::raw_pointer_cast(U_data.data());
  const float* V = thrust::raw_pointer_cast(V_data.data());
  thrust::for_each_n(policy, thrust::make_counting_iterator(0), U.size(),
    [U, V, nx, ny] __host__ __device__ (int xy) {
      …
    });
}
```

```
void heat_equation(auto policy, int nx, int ny,
                   const thrust::universal_vector<float>& U_data,
                   thrust::universal_vector<float>&       V_data) {
  const float* U = thrust::raw_pointer_cast(U_data.data());
  const float* V = thrust::raw_pointer_cast(V_data.data());
  thrust::for_each_n(policy, thrust::make_counting_iterator(0), U.size(),
    [U, V, nx, ny] __host__ __device__ (int xy) {
      int x = xy / ny;
      int y = xy % ny;

      …
    });
}
```

```
void heat_equation(auto policy, int nx, int ny,
                   const thrust::universal_vector<float>& U_data,
                   thrust::universal_vector<float>&       V_data) {
  const float* U = thrust::raw_pointer_cast(U_data.data());
  const float* V = thrust::raw_pointer_cast(V_data.data());
  thrust::for_each_n(policy, thrust::make_counting_iterator(0), U.size(),
    [U, V, nx, ny] __host__ __device__ (int xy) {
      int x = xy / ny;
      int y = xy % ny;

      if (x > 0 && y > 0 && x < nx - 1 && y < ny - 1) {
        …
      } else …
    });
}
```

```
void heat_equation(auto policy, int nx, int ny,
                   const thrust::universal_vector<float>& U_data,
                   thrust::universal_vector<float>&       V_data) {
  const float* U = thrust::raw_pointer_cast(U_data.data());
  const float* V = thrust::raw_pointer_cast(V_data.data());
  thrust::for_each_n(policy, thrust::make_counting_iterator(0), U.size(),
    [U, V, nx, ny] __host__ __device__ (int xy) {
      int x = xy / ny;
      int y = xy % ny;

      if (x > 0 && y > 0 && x < nx - 1 && y < ny - 1) {
        auto d2tdx2 = U[x*ny + y - 1]   - U[x*ny + y] * 2 + U[x*ny + y + 1];
        auto d2tdy2 = U[(x - 1)*ny + y] - U[x*ny + y] * 2 + U[(x + 1)*ny + y];

        V[x*ny + y] = U[x*ny + y] + 0.2f * (d2tdx2 + d2tdy2);
      } else …
    });
}
```

```
void heat_equation(auto policy, int nx, int ny,
                   const thrust::universal_vector<float>& U_data,
                   thrust::universal_vector<float>&       V_data) {
  const float* U = thrust::raw_pointer_cast(U_data.data());
  const float* V = thrust::raw_pointer_cast(V_data.data());
  thrust::for_each_n(policy, thrust::make_counting_iterator(0), U.size(),
    [U, V, nx, ny] __host__ __device__ (int xy) {
      int x = xy / ny;
      int y = xy % ny;

      if (x > 0 && y > 0 && x < nx - 1 && y < ny - 1) {
        auto d2tdx2 = U[x*ny + y - 1]   - U[x*ny + y] * 2 + U[x*ny + y + 1];
        auto d2tdy2 = U[(x - 1)*ny + y] - U[x*ny + y] * 2 + U[(x + 1)*ny + y];

        V[x*ny + y] = U[x*ny + y] + 0.2f * (d2tdx2 + d2tdy2);
      } else {
        V[x*ny + y] = U[x*ny + y];
      }
    });
}
```

mdspan: A non-owning handle to multidimensional data

```
template <class T, class Extents, class LayoutPolicy = …, class AccessorPolicy = …>
class std::mdspan;
```

```
mdspan A(data, N, M}; // Row-major: C/NumPy.
mdspan A(data, layout_right::mapping(N, M));

A(i, j)     == data(i * M + j)
A.stride(0) == M
A.stride(1) == 1
```

Location	Element
0	a_{00}
1	a_{01}
2	a_{10}
3	a_{11}

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}$$

```
// Column-major: Fortran/MATLAB.
mdspan B(data, layout_left::mapping(N, M}));

B(i, j)     == data(i + j * N)
B.stride(0) == 1
B.stride(1) == N
```

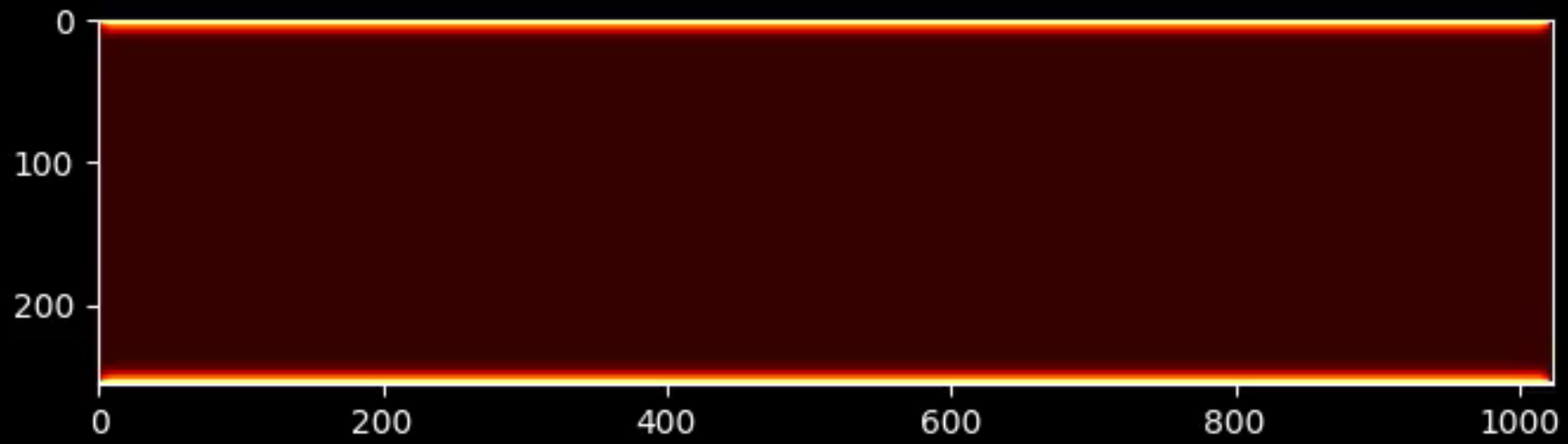
Location	Element
0	a_{00}
1	a_{10}
2	a_{01}
3	a_{11}

Learn More: S51755 C++ Standard Parallelism (2023)

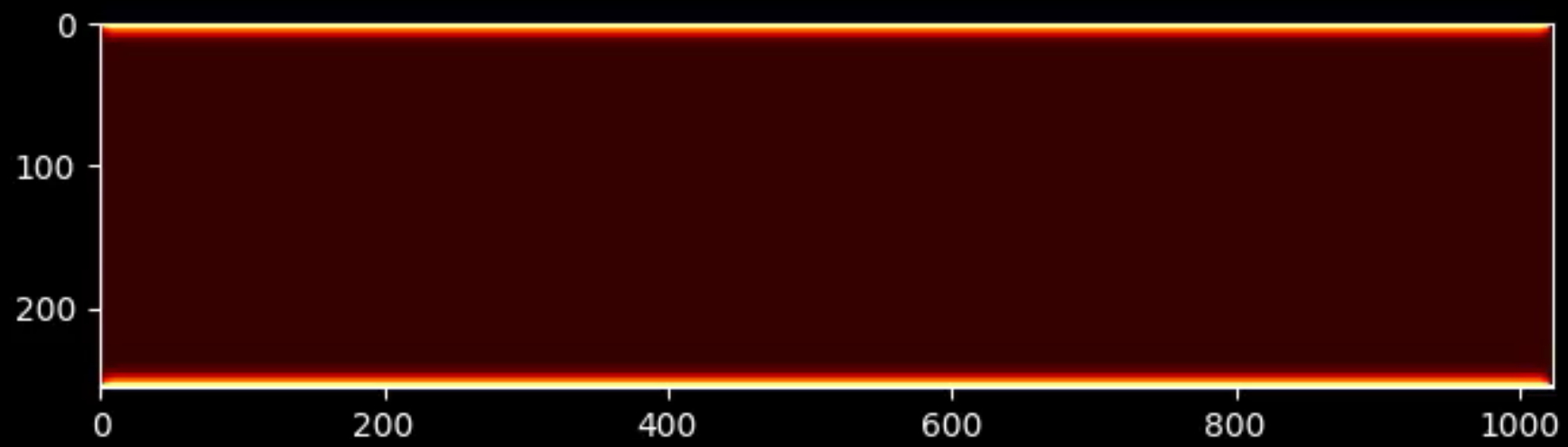
```
void heat_equation(auto policy, int nx, int ny,
                   const thrust::universal_vector<float>& U_data,
                   thrust::universal_vector<float>&       V_data) {
  const float* U = thrust::raw_pointer_cast(U_data.data());
  const float* V = thrust::raw_pointer_cast(V_data.data());
  thrust::for_each_n(policy, thrust::make_counting_iterator(0), U.size(),
    [U, V, nx, ny] __host__ __device__ (int xy) {
      int x = xy / ny;
      int y = xy % ny;

      if (x > 0 && y > 0 && x < nx - 1 && y < ny - 1) {
        auto d2tdx2 = U[x*ny + y - 1]   - U[x*ny + y] * 2 + U[x*ny + y + 1];
        auto d2tdy2 = U[(x - 1)*ny + y] - U[x*ny + y] * 2 + U[(x + 1)*ny + y];

        V[x*ny + y] = U[x*ny + y] + 0.2f * (d2tdx2 + d2tdy2);
      } else {
        V[x*ny + y] = U[x*ny + y];
      }
    });
}
```

```
void heat_equation(auto policy,
                   cuda::std::mdspan<float const, cuda::std::dims<2>> U,
                   cuda::std::mdspan<float, cuda::std::dims<2>>       V) {
  thrust::for_each_n(policy, thrust::make_counting_iterator(0), U.size(),
    [U, V] __host__ __device__ (int xy) {
      int x = xy / U.extent(1);
      int y = xy % U.extent(1);

      if (x > 0 && y > 0 && x < I.extent(0) - 1 && y < U.extent(1) - 1) {
        auto d2tdx2 = U(x, y - 1) - U(x, y) * 2 + U(x, y + 1);
        auto d2tdy2 = U(x - 1, y) - U(x, y) * 2 + U(x + 1, y);

        V(x, y) = U(x, y) + 0.2f * (d2tdx2 + d2tdy2);
      } else {
        V(x, y) = U(x, y);
      }
    });
}
```

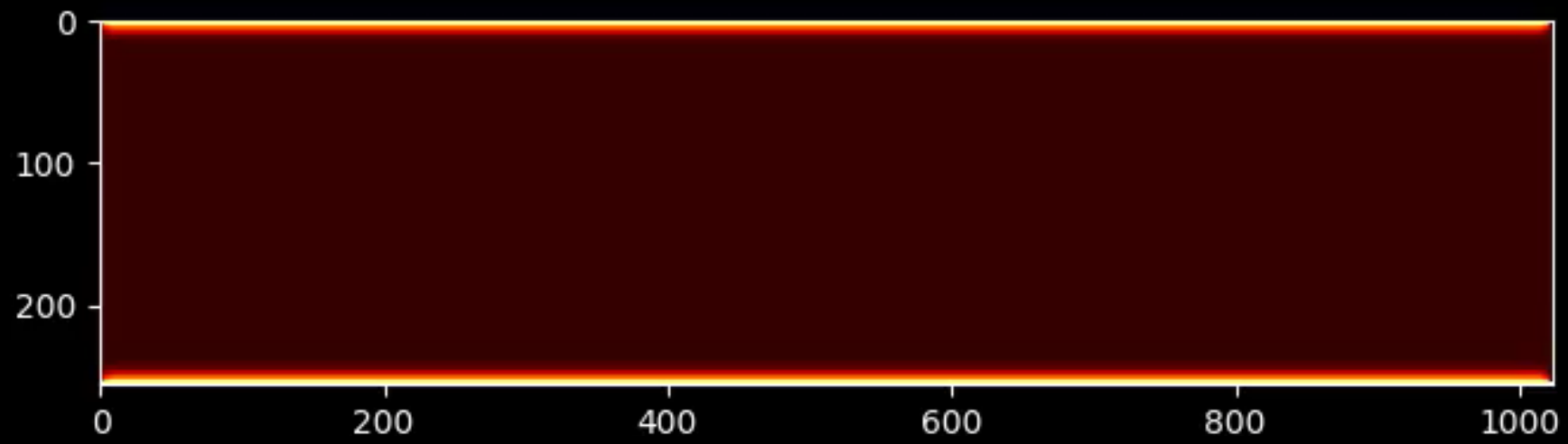
```
void initialize_oven(auto policy, cuda::std::mdspan<float, cuda::std::dims<2>> U);
```



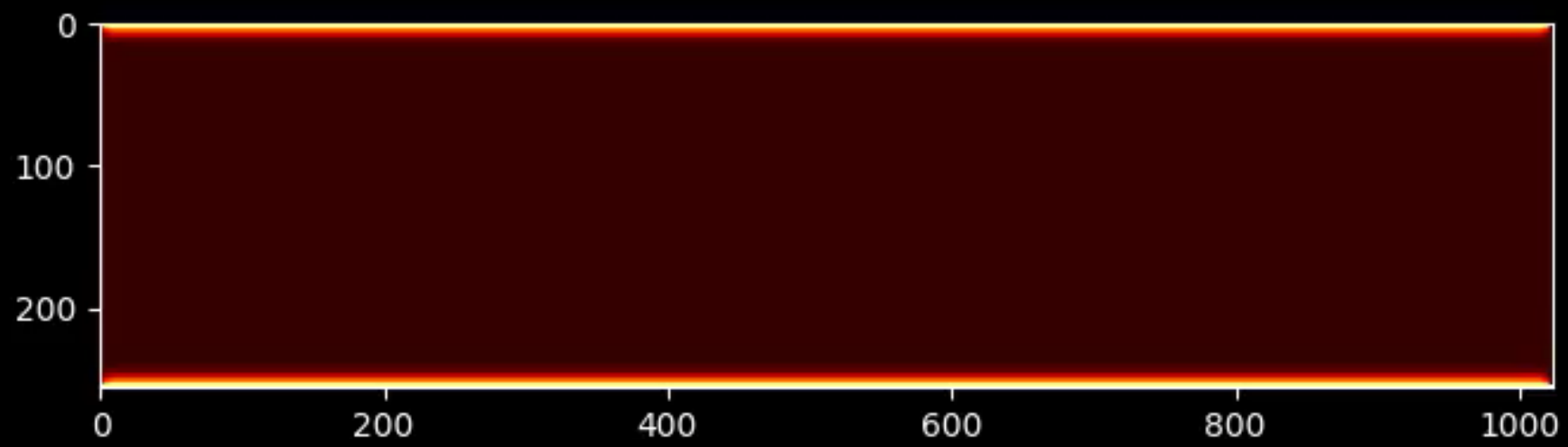
```
void initialize_oven(auto policy, cuda::std::mdspan<float, cuda::std::dims<2>> U) {
  auto nx = U.extent(0);
  auto top = cuda::std::submdspan(U, 0, cuda::std::full_extent);

  auto bot = cuda::std::submdspan(U, nx-1, cuda::std::full_extent);
}
```



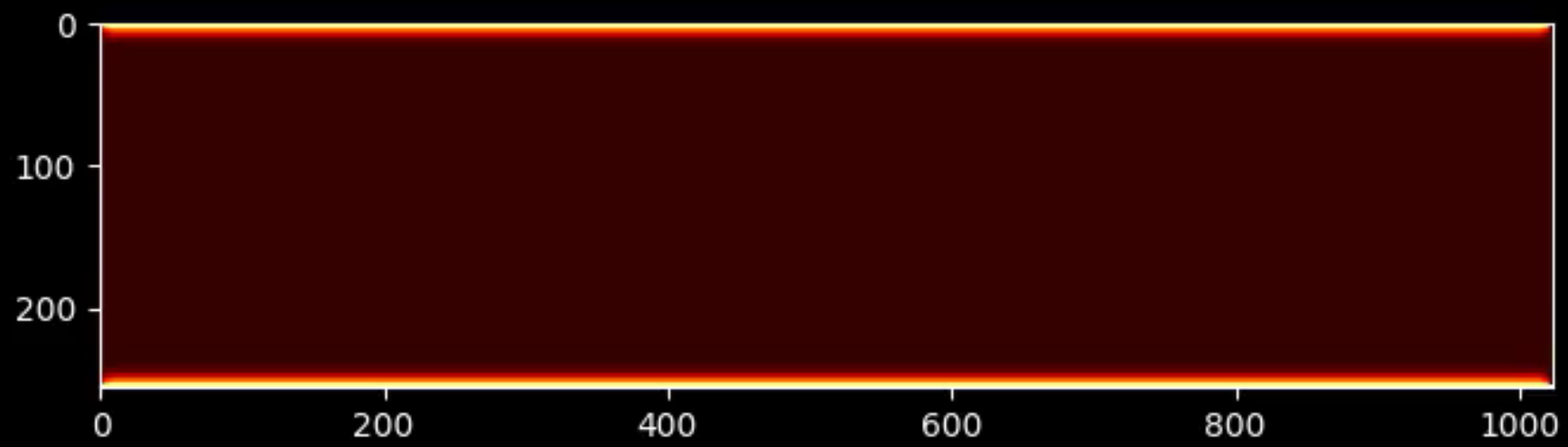
```
void initialize_oven(auto policy, cuda::std::mdspan<float, cuda::std::dims<2>> U) {
  auto nx = U.extent(0);
  auto top = cuda::std::submdspan(U, 0, cuda::std::full_extent);
  auto mid = cuda::std::submdspan(U, cuda::std::tuple(1, nx-2), cuda::std::full_extent);
  auto bot = cuda::std::submdspan(U, nx-1, cuda::std::full_extent);
}
```



```
void initialize_oven(auto policy, cuda::std::mdspan<float, cuda::std::dims<2>> U) {
  auto nx = U.extent(0);
  auto top = cuda::std::submdspan(U, 0, cuda::std::full_extent);
  auto mid = cuda::std::submdspan(U, cuda::std::tuple(1, nx-2), cuda::std::full_extent);
  auto bot = cuda::std::submdspan(U, nx-1, cuda::std::full_extent);

  thrust::fill_n(policy, top.data(), top.size(), 90.0);

  thrust::fill_n(policy, bot.data(), bot.size(), 90.0);
}
```



```
void initialize_oven(auto policy, cuda::std::mdspan<float, cuda::std::dims<2>> U) {
  auto nx = U.extent(0);
  auto top = cuda::std::submdspan(U, 0, cuda::std::full_extent);
  auto mid = cuda::std::submdspan(U, cuda::std::tuple(1, nx-2), cuda::std::full_extent);
  auto bot = cuda::std::submdspan(U, nx-1, cuda::std::full_extent);

  thrust::fill_n(policy, top.data(), top.size(), 90.0);
  thrust::fill_n(policy, mid.data(), mid.size(), 15.0);
  thrust::fill_n(policy, bot.data(), bot.size(), 90.0);
}
```



```
int nx(…), ny(…), write_steps(…), compute_steps(…);
```

```
int nx(…), ny(…), write_steps(…), compute_steps(…);

thrust::universal_vector<float> U_data(nx * ny), V_data(nx * ny);
cuda::std::mdspan U(thrust::raw_pointer_cast(U_data.data()), nx, ny);
cuda::std::mdspan V(thrust::raw_pointer_cast(V_data.data()), nx, ny);
```

```
int nx(…), ny(…), write_steps(…), compute_steps(…);

thrust::universal_vector<float> U_data(nx * ny), V_data(nx * ny);
cuda::std::mdspan U(thrust::raw_pointer_cast(U_data.data()), nx, ny);
cuda::std::mdspan V(thrust::raw_pointer_cast(V_data.data()), nx, ny);

initialize_oven(thrust::cuda::par, U);
```

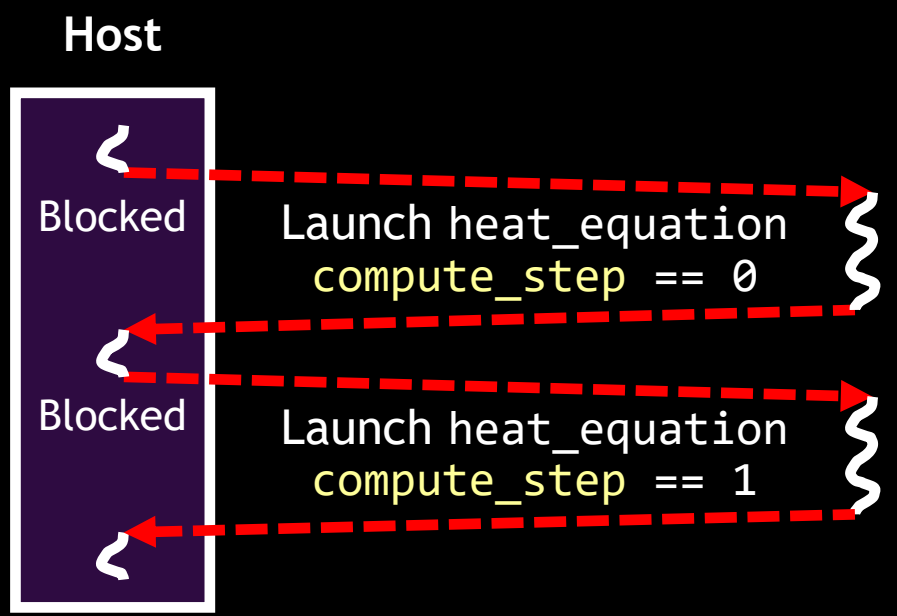
```
int nx(…), ny(…), write_steps(…), compute_steps(…);

thrust::universal_vector<float> U_data(nx * ny), V_data(nx * ny);
cuda::std::mdspan U(thrust::raw_pointer_cast(U_data.data()), nx, ny);
cuda::std::mdspan V(thrust::raw_pointer_cast(V_data.data()), nx, ny);

initialize_oven(thrust::cuda::par, U);

for (auto write_step : std::views::iota(0, write_steps)) {
  …
}
```

```
int nx(…), ny(…), write_steps(…), compute_steps(…);

thrust::universal_vector<float> U_data(nx * ny), V_data(nx * ny);
cuda::std::mdspan U(thrust::raw_pointer_cast(U_data.data()), nx, ny);
cuda::std::mdspan V(thrust::raw_pointer_cast(V_data.data()), nx, ny);

initialize_oven(thrust::cuda::par, U);

for (auto write_step : std::views::iota(0, write_steps)) {
  save_to_file(U);

  for (auto compute_step : std::views::iota(0, compute_steps)) {
    …
  }
}
```

```
int nx(…), ny(…), write_steps(…), compute_steps(…);

thrust::universal_vector<float> U_data(nx * ny), V_data(nx * ny);
cuda::std::mdspan U(thrust::raw_pointer_cast(U_data.data()), nx, ny);
cuda::std::mdspan V(thrust::raw_pointer_cast(V_data.data()), nx, ny);

initialize_oven(thrust::cuda::par, U);

for (auto write_step : std::views::iota(0, write_steps)) {
  save_to_file(U);

  for (auto compute_step : std::views::iota(0, compute_steps)) {
    heat_equation(thrust::cuda::par, U, V);
    …
  }
}
```

```
int nx(…), ny(…), write_steps(…), compute_steps(…);

thrust::universal_vector<float> U_data(nx * ny), V_data(nx * ny);
cuda::std::mdspan U(thrust::raw_pointer_cast(U_data.data()), nx, ny);
cuda::std::mdspan V(thrust::raw_pointer_cast(V_data.data()), nx, ny);

initialize_oven(thrust::cuda::par, U);

for (auto write_step : std::views::iota(0, write_steps)) {
  save_to_file(U);

  for (auto compute_step : std::views::iota(0, compute_steps)) {
    heat_equation(thrust::cuda::par, U, V);
    U.swap(V);
  }
}
```

```
compute_step = 0;
heat_equation(thrust::cuda::par, U, V);
U.swap(V);
compute_step = 1;
heat_equation(thrust::cuda::par, U, V);
U.swap(V);
```

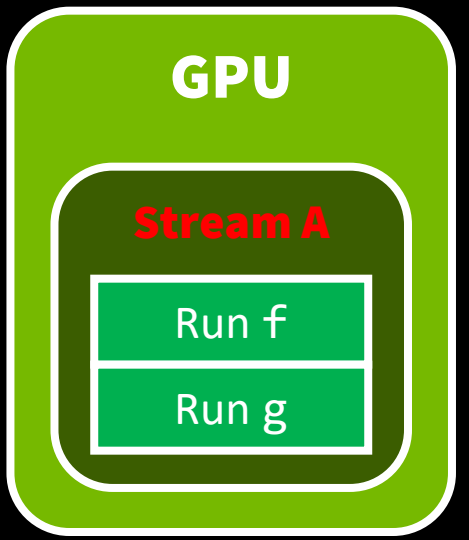


```
int nx(…), ny(…), write_steps(…), compute_steps(…);

thrust::universal_vector<float> U_data(nx * ny), V_data(nx * ny);
cuda::std::mdspan U(thrust::raw_pointer_cast(U_data.data()), nx, ny);
cuda::std::mdspan V(thrust::raw_pointer_cast(V_data.data()), nx, ny);

initialize_oven(thrust::cuda::par, U);

for (auto write_step : std::views::iota(0, write_steps)) {
  save_to_file(U);

  for (auto compute_step : std::views::iota(0, compute_steps)) {
    heat_equation(thrust::cuda::par, U, V);
    U.swap(V);
  }
}
```

```
int nx(…), ny(…), write_steps(…), compute_steps(…);

thrust::universal_vector<float> U_data(nx * ny), V_data(nx * ny);
cuda::std::mdspan U(thrust::raw_pointer_cast(U_data.data()), nx, ny);
cuda::std::mdspan V(thrust::raw_pointer_cast(V_data.data()), nx, ny);

initialize_oven(thrust::cuda::par_nosync, U);

for (auto write_step : std::views::iota(0, write_steps)) {
  save_to_file(U);

  for (auto compute_step : std::views::iota(0, compute_steps)) {
    heat_equation(thrust::cuda::par_nosync, U, V);
    U.swap(V);
  }
}
```

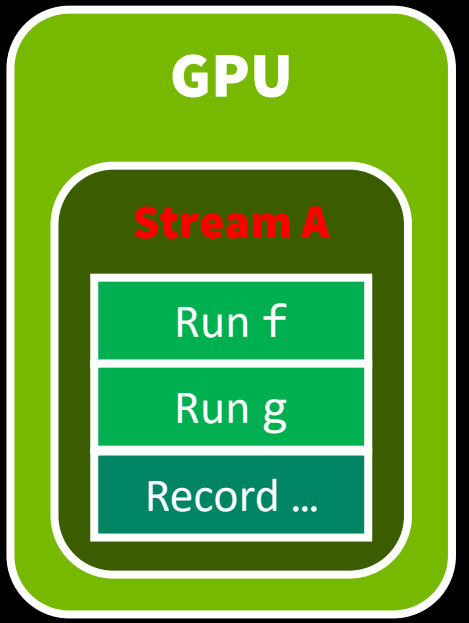
```
int nx(…), ny(…), write_steps(…), compute_steps(…);

thrust::universal_vector<float> U_data(nx * ny), V_data(nx * ny);
cuda::std::mdspan U(thrust::raw_pointer_cast(U_data.data()), nx, ny);
cuda::std::mdspan V(thrust::raw_pointer_cast(V_data.data()), nx, ny);

initialize_oven(thrust::cuda::par_nosync, U);

for (auto write_step : std::views::iota(0, write_steps)) {
  cudaDeviceSynchronize();
  save_to_file(U);

  for (auto compute_step : std::views::iota(0, compute_steps)) {
    heat_equation(thrust::cuda::par_nosync, U, V);
    U.swap(V);
  }
}
```

```
auto pol(thrust::cuda::par_nosync);

cudax::stream A;

thrust::for_each(pol.on(A), …, f);
thrust::for_each(pol.on(A), …, g);
```



A stream is a chain of device work that is executed in order.

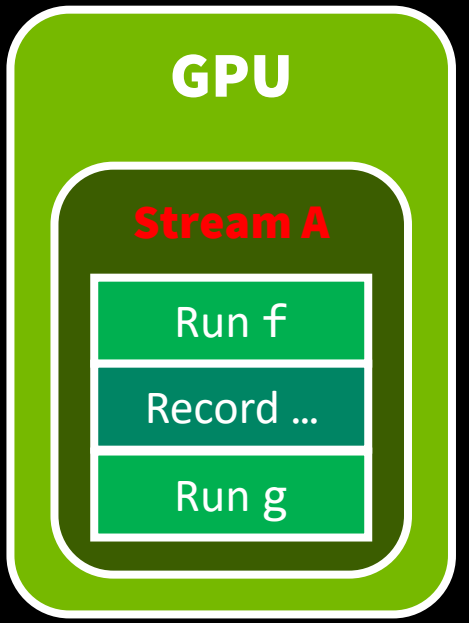
```
auto pol(thrust::cuda::par_nosync);

cudax::stream A;

thrust::for_each(pol.on(A), …, f);
thrust::for_each(pol.on(A), …, g);
A.wait();
```



A stream is a chain of device work that is executed in order.

They can record events that host code can sync with.

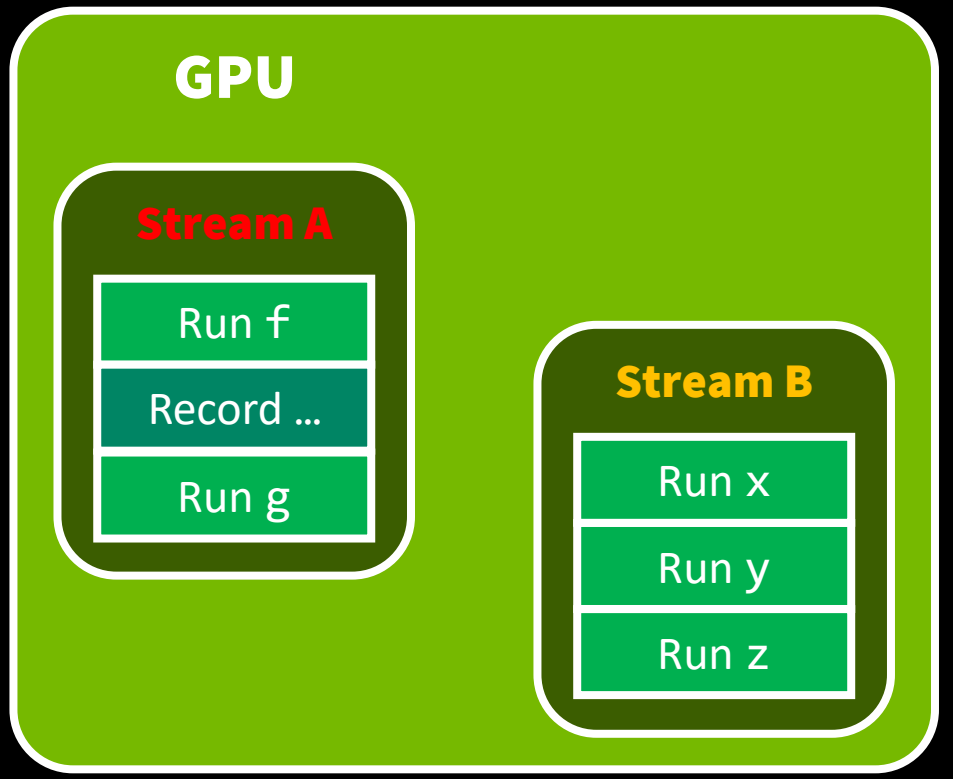
```
auto pol(thrust::cuda::par_nosync);

cudax::stream A;

thrust::for_each(pol.on(A), …, f);
A.wait();
thrust::for_each(pol.on(A), …, g);
```



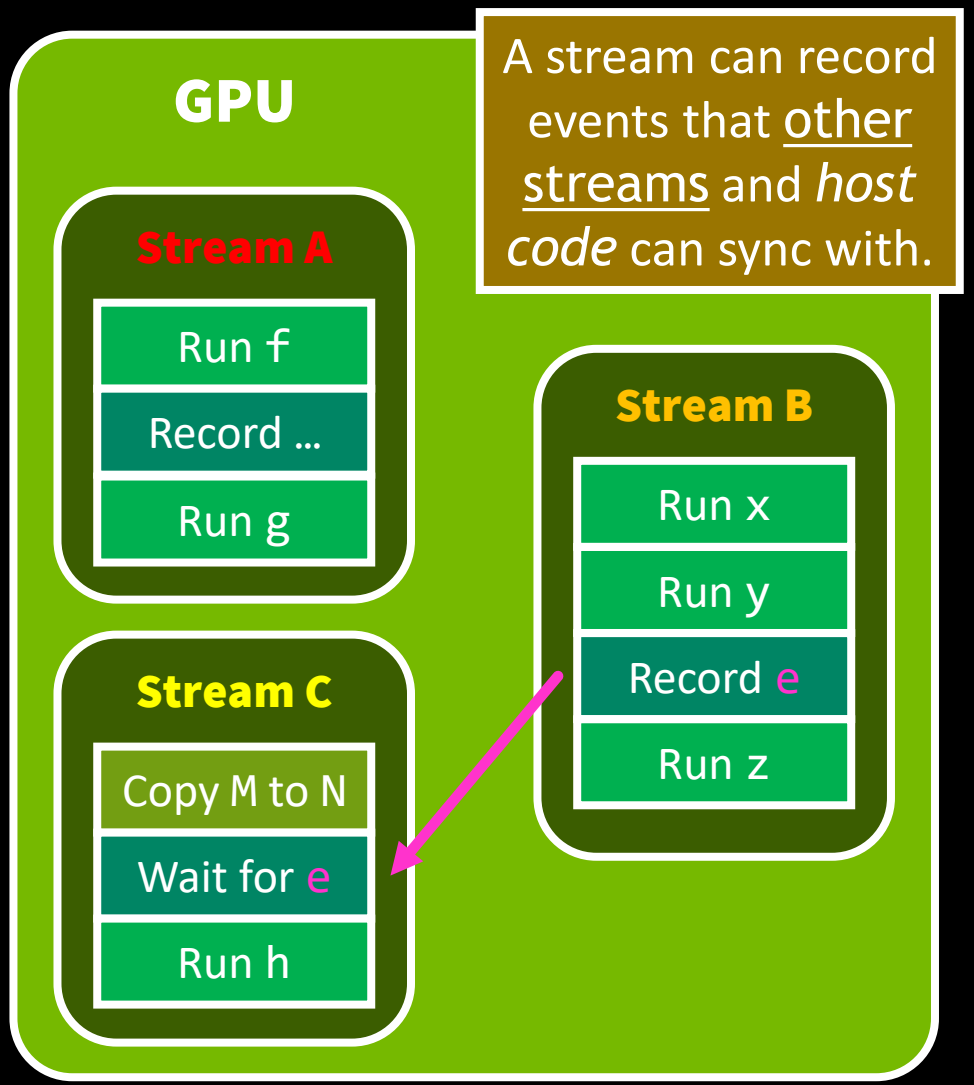
A stream is a chain of device work that is executed in order.

They can record events that host code can sync with.

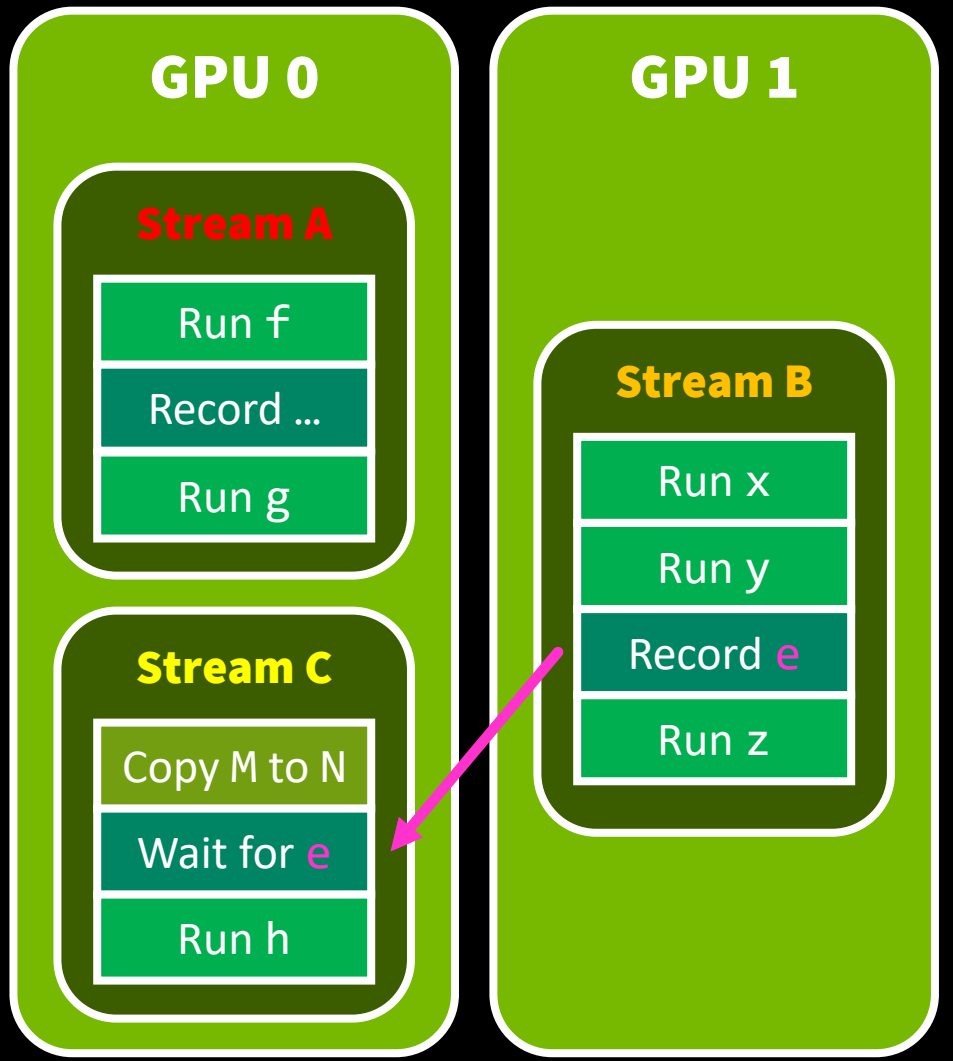
```
auto pol(thrust::cuda::par_nosync);

cudax::stream A;

thrust::for_each(pol.on(A), …, f);
A.wait();
thrust::for_each(pol.on(A), …, g);

cudax::stream B;

thrust::for_each(pol.on(B), …, x);
thrust::for_each(pol.on(B), …, y);
thrust::for_each(pol.on(B), …, z);
```



```
auto pol(thrust::cuda::par_nosync);

cudax::stream A;

thrust::for_each(pol.on(A), …, f);
A.wait();
thrust::for_each(pol.on(A), …, g);

cudax::stream B;

thrust::for_each(pol.on(B), …, x);
thrust::for_each(pol.on(B), …, y);
auto e = B.record_event();
thrust::for_each(pol.on(B), …, z);

cudax::stream C;

thrust::copy(pol.on(C), M, …, N);
C.wait(e);
thrust::for_each(pol.on(C), … h);
```



```
auto pol(thrust::cuda::par_nosync);

cudax::stream A(cudax::devices[0]);

thrust::for_each(pol.on(A), …, f);
A.wait();
thrust::for_each(pol.on(A), …, g);

cudax::stream B(cudax::devices[1]);

thrust::for_each(pol.on(B), …, x);
thrust::for_each(pol.on(B), …, y);
auto e = B.record_event();
thrust::for_each(pol.on(B), …, z);

cudax::stream C(cudax::devices[0]);

thrust::copy(pol.on(C), M, …, N);
C.wait(e);
thrust::for_each(pol.on(C), … h);
```



```
int nx(…), ny(…), write_steps(…), compute_steps(…);

thrust::universal_vector<float> U_data(nx * ny), V_data(nx * ny);
cuda::std::mdspan U(…), V(…);

initialize_oven(thrust::cuda::par_nosync, U);

for (auto write_step : std::views::iota(0, write_steps)) {
  cudaDeviceSynchronize();
  save_to_file(U);

  for (auto compute_step : std::views::iota(0, compute_steps)) {
    heat_equation(thrust::cuda::par_nosync, U, V);
    U.swap(V);
  }
}
```

```
int nx(…), ny(…), write_steps(…), compute_steps(…);

cudax::stream stream;
auto policy(thrust::cuda::par_nosync.on(stream.get()));

thrust::universal_vector<float> U_data(nx * ny), V_data(nx * ny);
cuda::std::mdspan U(…), V(…);

initialize_oven(policy, U);

for (auto write_step : std::views::iota(0, write_steps)) {
  stream.wait();
  save_to_file(U);

  for (auto compute_step : std::views::iota(0, compute_steps)) {
    heat_equation(policy, U, V);
    U.swap(V);
  }
}
```

```
auto policy = thrust::cuda::par_nosync;
thrust::universal_vector A(…), B(…);

thrust::for_each_n(
  policy, A.begin(), A.size(), f0);
// Doesn't return anything.

auto it1 = thrust::transform(policy,
  A.begin(), A.end(), B.begin(), f1);
// Returns an iterator to the end of the output (B).
// Doesn't depend on the computation; it's just B.begin() + A.size().
```

```
auto policy = thrust::cuda::par_nosync;
thrust::universal_vector A(…), B(…);

thrust::for_each_n(
  policy, A.begin(), A.size(), f0);
// Doesn't return anything.

auto it1 = thrust::transform(policy,
  A.begin(), A.end(), B.begin(), f1);
// Returns an iterator to the end of the output (B).
// Doesn't depend on the computation; it's just B.begin() + A.size().
```

Some Thrust algorithms either:

- Return a result that depends on the computation, or
- Allocate temporary storage for the computation.

These interfaces will always block.

```
auto policy = thrust::cuda::par_nosync;
thrust::universal_vector A(…), B(…);

thrust::for_each_n(
  policy, A.begin(), A.size(), f0);
// Doesn't return anything.

auto it1 = thrust::transform(policy,
  A.begin(), A.end(), B.begin(), f1);
// Returns an iterator to the end of the output (B).
// Doesn't depend on the computation; it's just B.begin() + A.size().

auto it2 = thrust::inclusive_scan(policy, A.begin(), A.end(), B.begin(), f2);
// Also returns the end of the output, but needs temporary storage.
```

Some Thrust algorithms either:

- Return a result that depends on the computation, or
- Allocate temporary storage for the computation.

These interfaces will always block.

```
auto policy = thrust::cuda::par_nosync;
thrust::universal_vector A(…), B(…);

thrust::for_each_n(
  policy, A.begin(), A.size(), f0);
// Doesn't return anything.

auto it1 = thrust::transform(policy,
  A.begin(), A.end(), B.begin(), f1);
// Returns an iterator to the end of the output (B).
// Doesn't depend on the computation; it's just B.begin() + A.size().

auto it2 = thrust::inclusive_scan(policy, A.begin(), A.end(), B.begin(), f2);
// Also returns the end of the output, but needs temporary storage.

auto it3 = thrust::copy_if(policy, A.begin(), A.end(), B.begin(), f3);
// Returns the end of the output, but it depends on the computation.
```

Some Thrust algorithms either:

- Return a result that depends on the computation, or
- Allocate temporary storage for the computation.

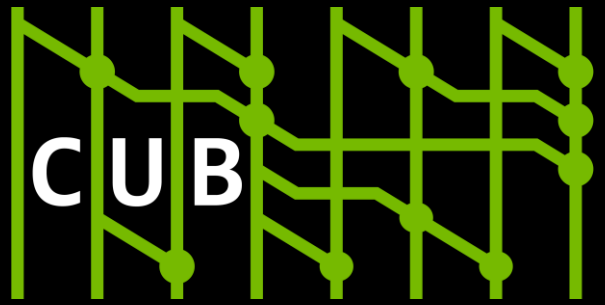
These interfaces will always block.

```
auto policy = thrust::cuda::par_nosync;
thrust::universal_vector A(…), B(…);

thrust::for_each_n(
  policy, A.begin(), A.size(), f0);
// Doesn't return anything.

auto it1 = thrust::transform(policy,
  A.begin(), A.end(), B.begin(), f1);
// Returns an iterator to the end of the output (B).
// Doesn't depend on the computation; it's just B.begin() + A.size().

auto it2 = thrust::inclusive_scan(policy, A.begin(), A.end(), B.begin(), f2);
// Also returns the end of the output, but needs temporary storage.

auto it3 = thrust::copy_if(policy, A.begin(), A.end(), B.begin(), f3);
// Returns the end of the output, but it depends on the computation.

auto r4 = thrust::reduce(policy, A.begin(), A.end(), T{}, f4);
auto r5 = thrust::find_if(policy, A.begin(), A.end(), f5);
// Returns the result of the computation itself.
```

Some Thrust algorithms either:

- Return a result that depends on the computation, or
- Allocate temporary storage for the computation.

These interfaces will always block.



The CUDA C++ Algorithm Authoring Toolkit

https://nvidia.github.io/cccl/cub

Learn More: S72575 How You Should Write a CUDA C++ Kernel

Device-Wide Kernels

- cub::DeviceSegmentedReduce
- cub::DeviceHistogram
- cub::DeviceRadixSort
- cub::DeviceSelect
- ...

Block-Wide Cooperative Primitives

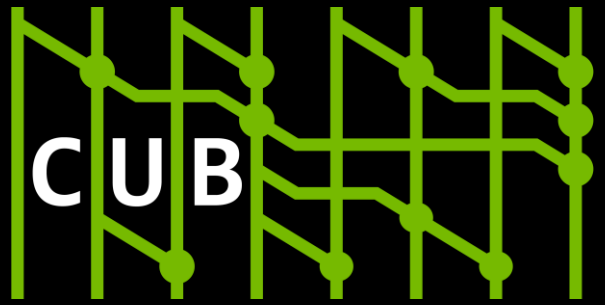
- cub::BlockReduce
- cub::BlockExchange
- cub::BlockRunLengthDecode
- cub::BlockLoad/Store
- ...

Warp-Wide Cooperative Primitives

- cub::WarpReduce
- cub::WarpExchange
- cub::WarpMergeSort
- cub::WarpLoad/Store
- ...

Per-Thread Primitives

- cub::ThreadReduce
- cub::ThreadScan
- cub::MergePathSearch
- cub::ThreadLoad/Store
- ...



The CUDA C++ Algorithm Authoring Toolkit

https://nvidia.github.io/cccl/cub

Learn More: S72575 How You Should Write a CUDA C++ Kernel

Device-Wide Kernels

- `cub::DeviceSegmentedReduce`
- `cub::DeviceHistogram`
- `cub::DeviceRadixSort`
- `cub::DeviceSelect`
- `...`

Block-Wide Cooperative Primitives

- `cub::BlockReduce`
- `cub::BlockExchange`
- `cub::BlockRunLengthDecode`
- `cub::BlockLoad/Store`
- `...`

Warp-Wide Cooperative Primitives

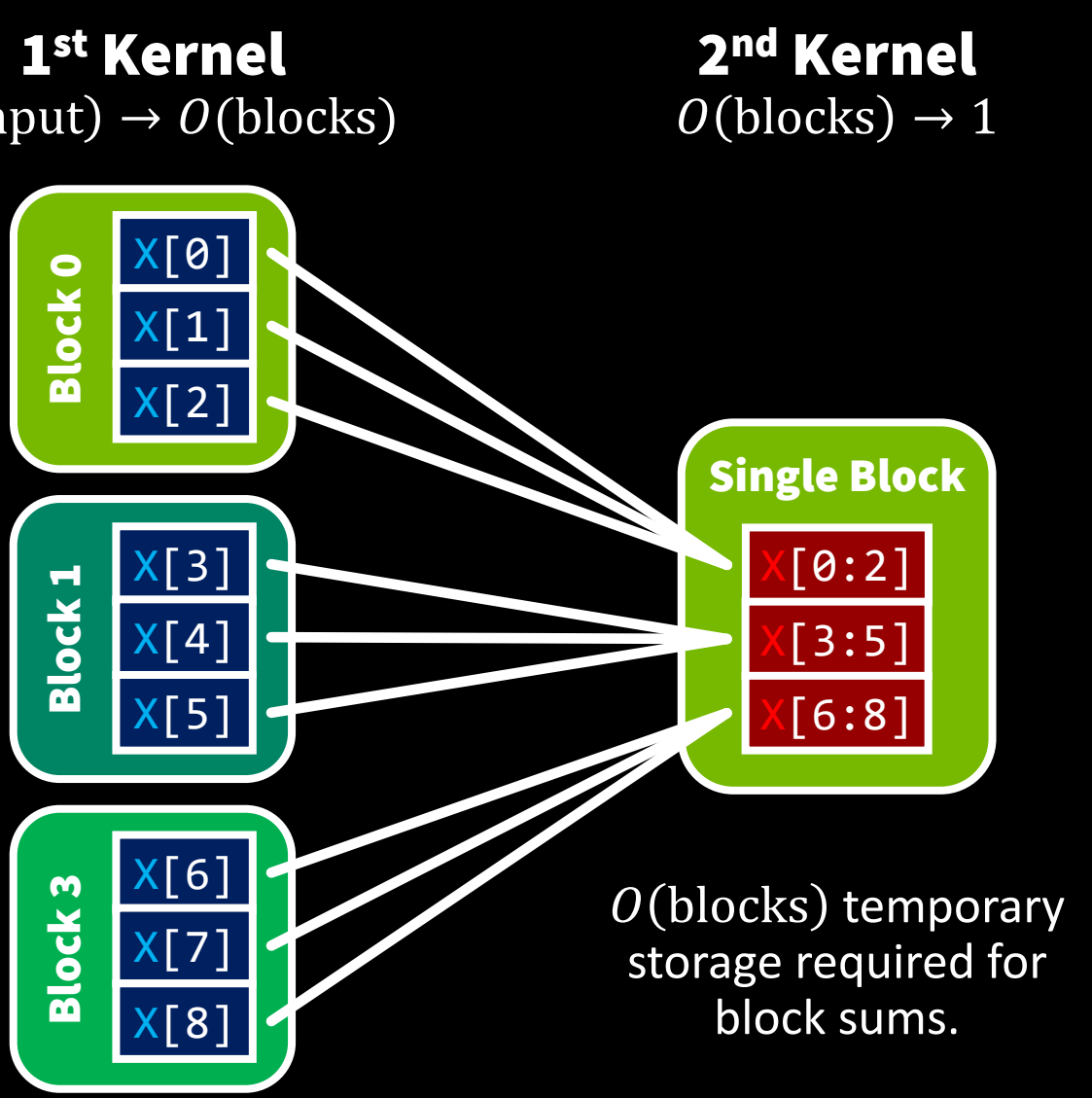
- `cub::WarpReduce`
- `cub::WarpExchange`
- `cub::WarpMergeSort`
- `cub::WarpLoad/Store`
- `...`

Per-Thread Primitives

- `cub::ThreadReduce`
- `cub::ThreadScan`
- `cub::MergePathSearch`
- `cub::ThreadLoad/Store`
- `...`

```
thrust::universal_vector X(…);

auto res = thrust::reduce(
  thrust::cuda::par,
  X.begin(), X.end(),
  T{}, cuda::maximum{});
```

```
thrust::universal_vector X(…);

auto res = thrust::reduce(
  thrust::cuda::par,
  X.begin(), X.end(),
  T{}, cuda::maximum{});
```



```
thrust::universal_vector X(…);
thrust::universal_vector res(1);
```

```
thrust::universal_vector X(…);

auto res = thrust::reduce(
  thrust::cuda::par,
  X.begin(), X.end(),
  T{}, cuda::maximum{});
```

```
thrust::universal_vector X(…);

auto res = thrust::reduce(
  thrust::cuda::par,
  X.begin(), X.end(),
  T{}, cuda::maximum{});
```

```
thrust::universal_vector X(…);
thrust::universal_vector res(1);

// Determine temporary storage size.
// Doesn't launch anything.
int tmp_size = 0;
cub::DeviceReduce::Reduce(
  nullptr,
  tmp_size,
  X.begin(), res.begin(), X.size(),
  cuda::maximum{}, T{});
```

```
thrust::universal_vector X(…);

auto res = thrust::reduce(
  thrust::cuda::par,
  X.begin(), X.end(),
  T{}, cuda::maximum{});
```

```
thrust::universal_vector X(…);
thrust::universal_vector res(1);

// Determine temporary storage size.
// Doesn't launch anything.
int tmp_size = 0;
cub::DeviceReduce::Reduce(
  nullptr,
  tmp_size,
  X.begin(), res.begin(), X.size(),
  cuda::maximum{}, T{});
```

```
thrust::universal_vector X(…);

auto res = thrust::reduce(
  thrust::cuda::par,
  X.begin(), X.end(),
  T{}, cuda::maximum{});
```

```
thrust::universal_vector X(…);
thrust::universal_vector res(1);

// Determine temporary storage size.
// Doesn't launch anything.
int tmp_size = 0;
cub::DeviceReduce::Reduce(
  nullptr,
  tmp_size,
  X.begin(), res.begin(), X.size(),
  cuda::maximum{}, T{});

thrust::device_vector<cuda::std::byte>
  tmp(tmp_size);
```

```
thrust::universal_vector X(…);

auto res = thrust::reduce(
  thrust::cuda::par,
  X.begin(), X.end(),
  T{}, cuda::maximum{});
```

```
thrust::universal_vector X(…);
thrust::universal_vector res(1);

// Determine temporary storage size.
// Doesn't launch anything.
int tmp_size = 0;
cub::DeviceReduce::Reduce(
  nullptr,
  tmp_size,
  X.begin(), res.begin(), X.size(),
  cuda::maximum{}, T{});

thrust::device_vector<cuda::std::byte>
  tmp(tmp_size);

// Launch kernel.
cub::DeviceReduce::Reduce(
  thrust::raw_pointer_cast(tmp.data()),
  tmp_size,
  X.begin(), res.begin(), X.size(),
  cuda::maximum{}, T{});
```

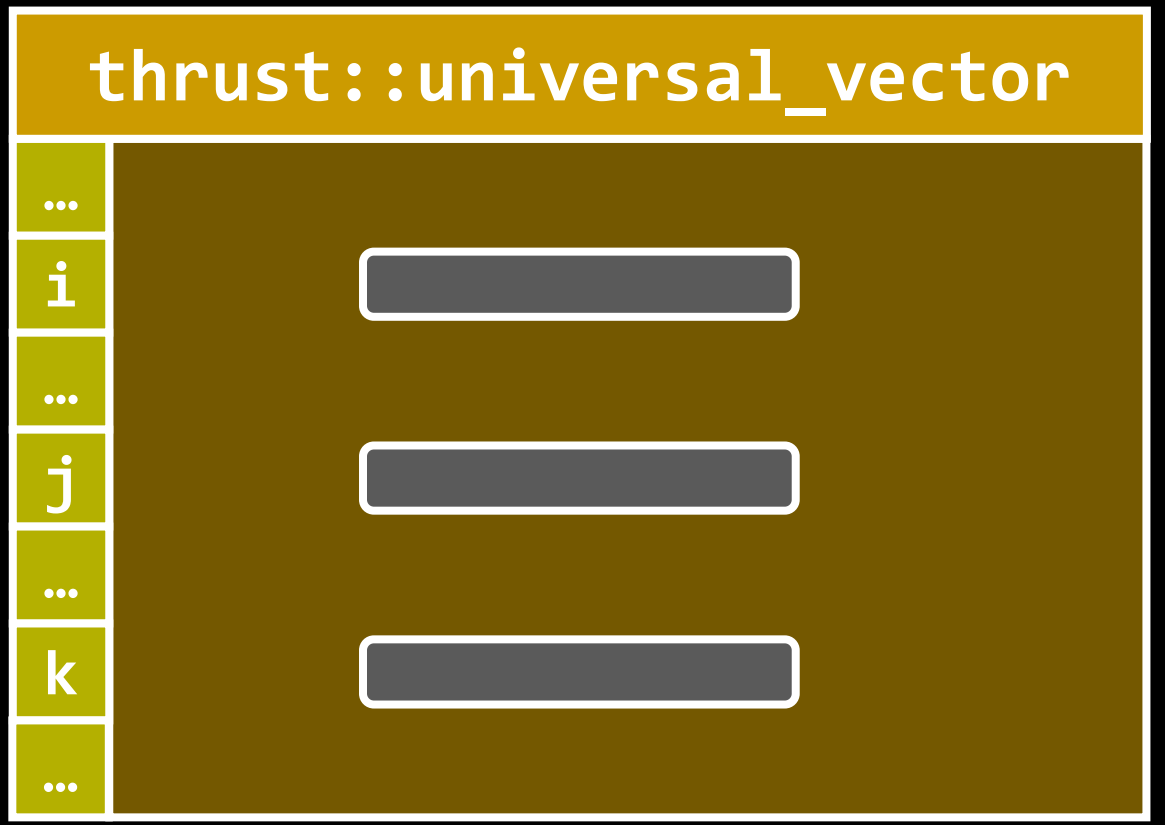
```
void heat_equation(auto policy,
                   cuda::std::mdspan<float const, cuda::std::dims<2>> U,
                   cuda::std::mdspan<float, cuda::std::dims<2>>       V) {
  thrust::for_each_n(policy, thrust::make_counting_iterator(0), U.size(),
    [U, V] __host__ __device__ (int xy) {
      int x = xy / U.extent(1);
      int y = xy % U.extent(1);

      if (x > 0 && y > 0 && x < I.extent(0) - 1 && y < U.extent(1) - 1) {
        auto d2tdx2 = U(x, y - 1) - U(x, y) * 2 + U(x, y + 1);
        auto d2tdy2 = U(x - 1, y) - U(x, y) * 2 + U(x + 1, y);

        V(x, y) = U(x, y) + 0.2f * (d2tdx2 + d2tdy2);
      } else {
        V(x, y) = U(x, y);
      }
    });
}
```

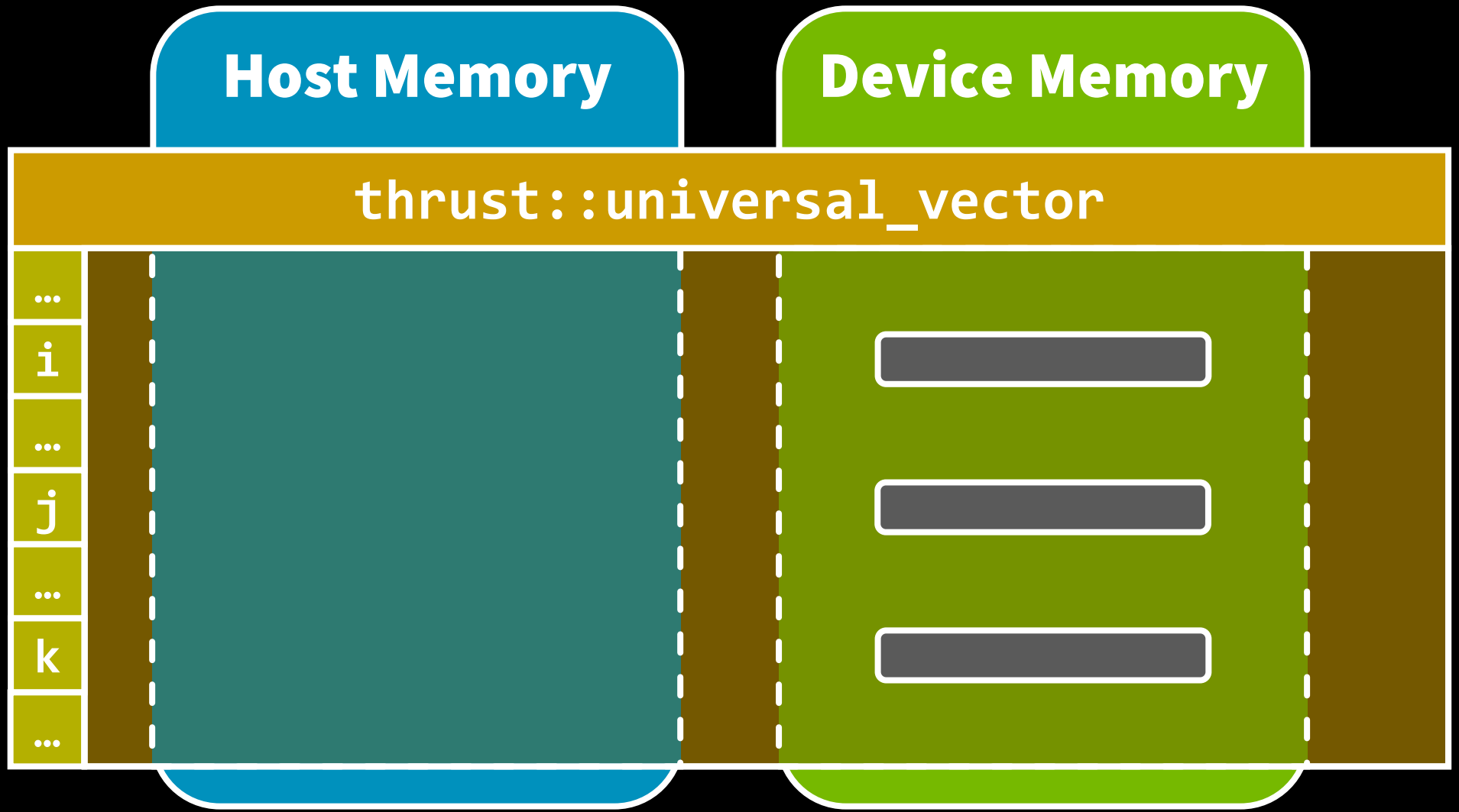
```
void heat_equation(cudax::stream_ref stream,
                   cuda::std::mdspan<float const, cuda::std::dims<2>> U,
                   cuda::std::mdspan<float, cuda::std::dims<2>>       V) {
  cub::DeviceFor::ForEachN(thrust::make_counting_iterator(0), U.size(),
    [U, V] __host__ __device__ (int xy) {
      int x = xy / U.extent(1);
      int y = xy % U.extent(1);

      if (x > 0 && y > 0 && x < I.extent(0) - 1 && y < U.extent(1) - 1) {
        auto d2tdx2 = U(x, y - 1) - U(x, y) * 2 + U(x, y + 1);
        auto d2tdy2 = U(x - 1, y) - U(x, y) * 2 + U(x + 1, y);

        V(x, y) = U(x, y) + 0.2f * (d2tdx2 + d2tdy2);
      } else {
        V(x, y) = U(x, y);
      }
    }, stream);
}
```

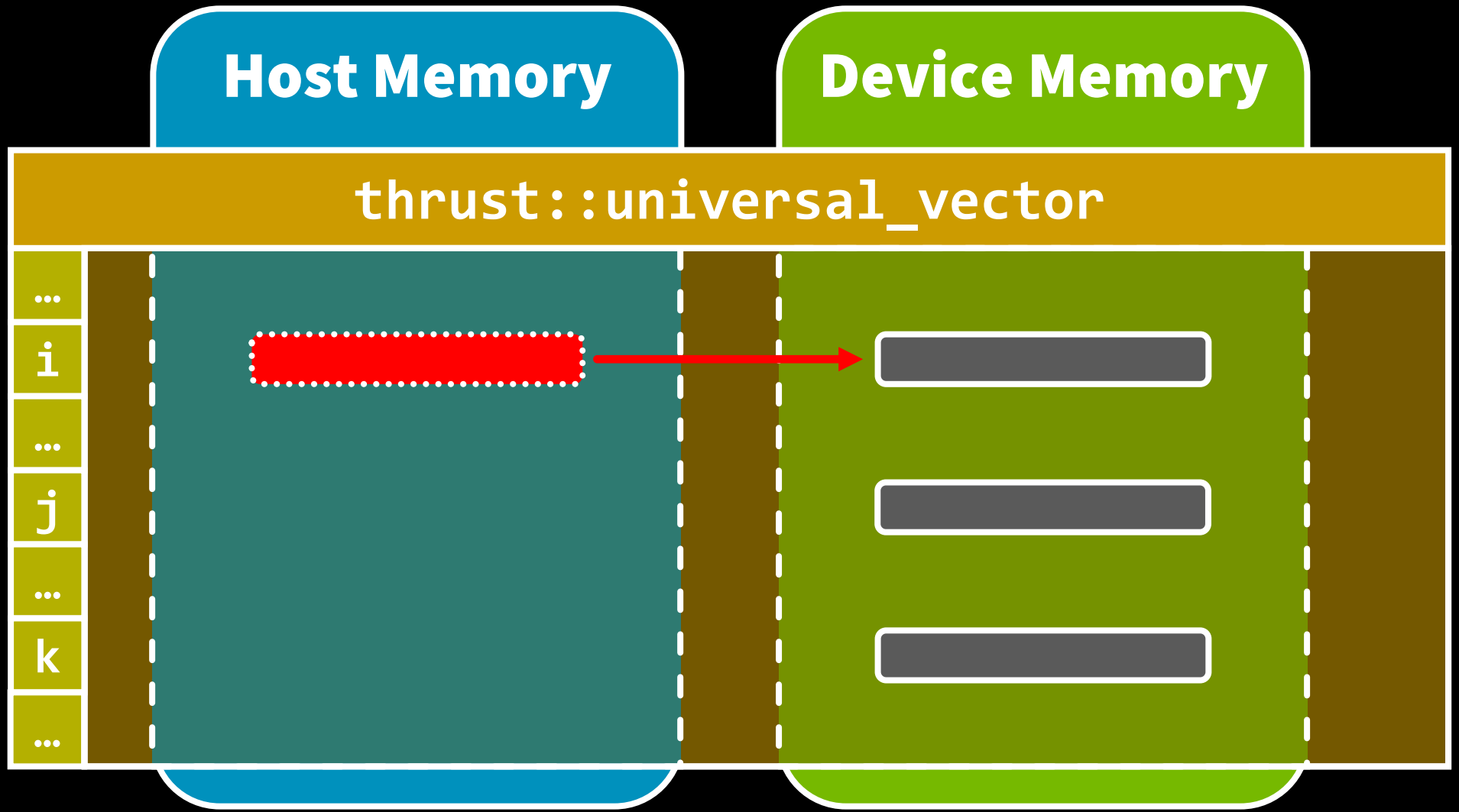
```
void heat_equation(cudax::stream_ref stream,
                   cuda::std::mdspan<float const, cuda::std::dims<2>> U,
                   cuda::std::mdspan<float, cuda::std::dims<2>>       V) {
  cub::DeviceFor::ForEachN(thrust::make_counting_iterator(0), U.size(),
    [U, V] __host__ __device__ (int xy) {
      int x = xy / U.extent(1);
      int y = xy % U.extent(1);

      if (x > 0 && y > 0 && x < I.extent(0) - 1 && y < U.extent(1) - 1) {
        auto d2tdx2 = U(x, y - 1) - U(x, y) * 2 + U(x, y + 1);
        auto d2tdy2 = U(x - 1, y) - U(x, y) * 2 + U(x + 1, y);

        V(x, y) = U(x, y) + 0.2f * (d2tdx2 + d2tdy2);
      } else {
        V(x, y) = U(x, y);
      }
    }, stream);
}
```

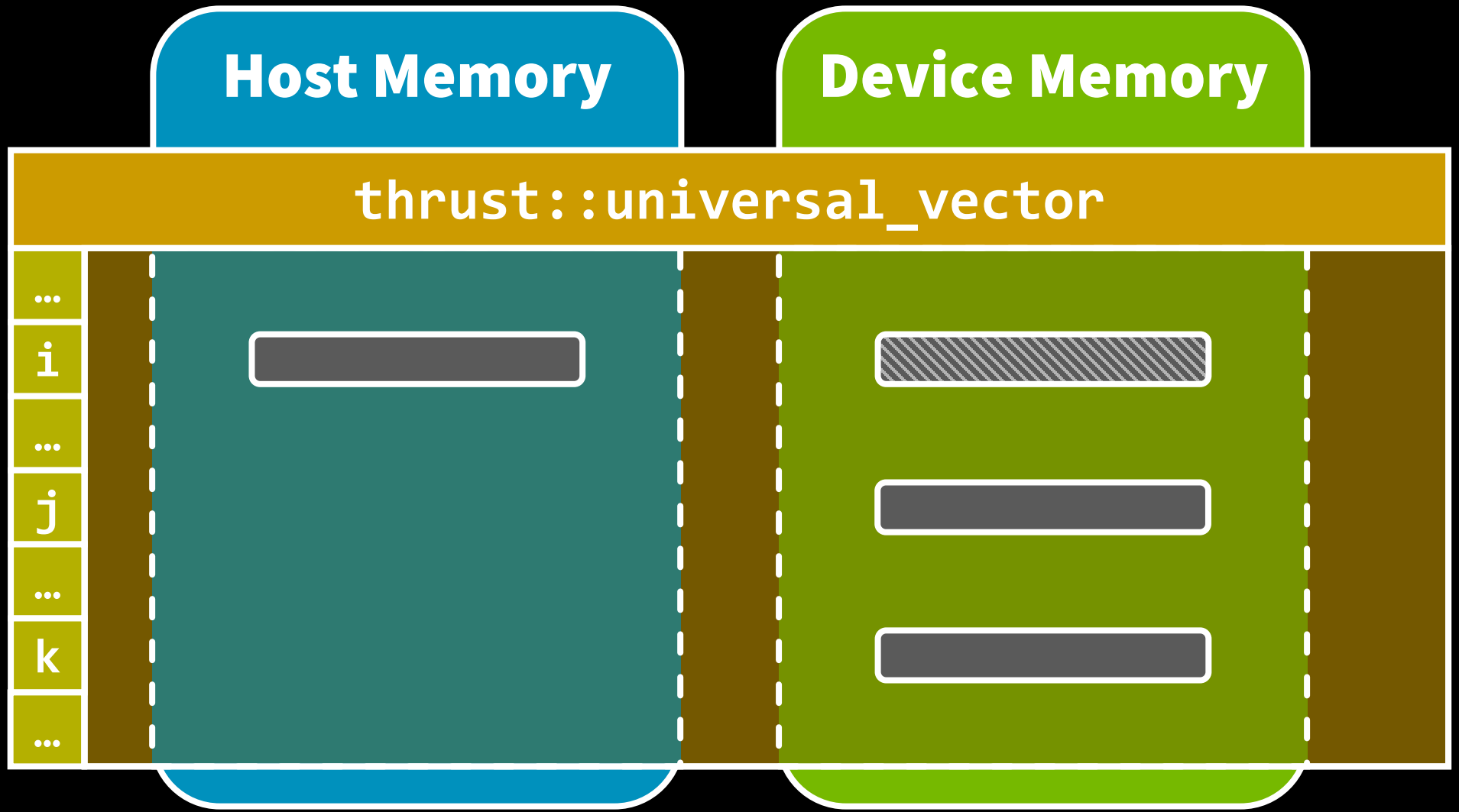
```
void heat_equation(cudax::stream_ref stream,
                   cuda::std::mdspan<float const, cuda::std::dims<2>> U,
                   cuda::std::mdspan<float, cuda::std::dims<2>>       V) {
  cub::DeviceFor::ForEachN(thrust::make_counting_iterator(0), U.size(),
    [U, V] __host__ __device__ (int xy) {
      int x = xy / U.extent(1);
      int y = xy % U.extent(1);

      if (x > 0 && y > 0 && x < I.extent(0) - 1 && y < U.extent(1) - 1) {
        auto d2tdx2 = U(x, y - 1) - U(x, y) * 2 + U(x, y + 1);
        auto d2tdy2 = U(x - 1, y) - U(x, y) * 2 + U(x + 1, y);

        V(x, y) = U(x, y) + 0.2f * (d2tdx2 + d2tdy2);
      } else {
        V(x, y) = U(x, y);
      }
    }, stream);
}
```

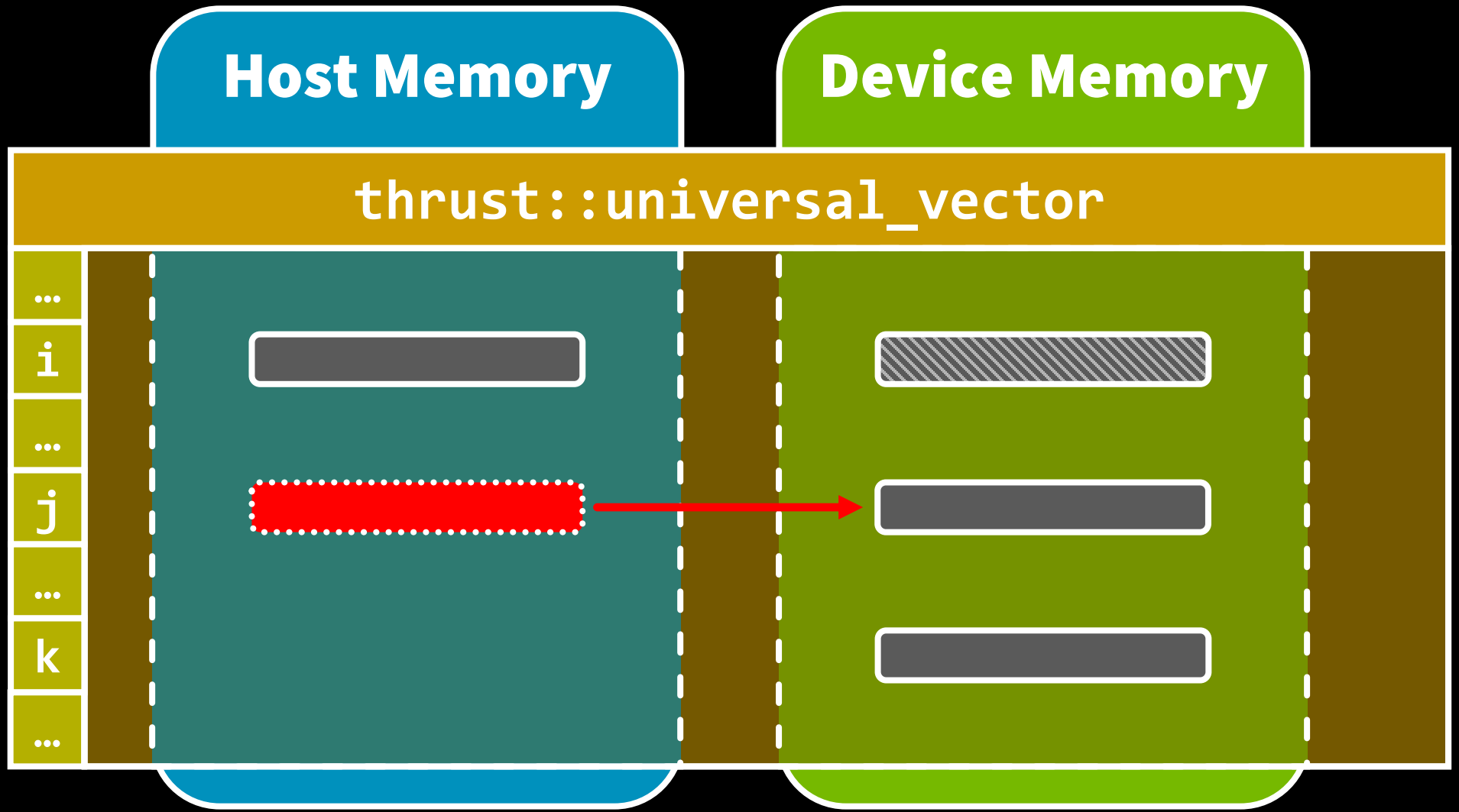
```
void heat_equation(cudax::stream_ref stream,
                   cuda::std::mdspan<float const, cuda::std::dims<2>> U,
                   cuda::std::mdspan<float, cuda::std::dims<2>>       V) {
  cub::DeviceFor::ForEachInExtents(U.extents(),
    [U, V] __host__ __device__ (int idx, int x, int y) {
      if (x > 0 && y > 0 && x < I.extent(0) - 1 && y < U.extent(1) - 1) {
        auto d2tdx2 = U(x, y - 1) - U(x, y) * 2 + U(x, y + 1);
        auto d2tdy2 = U(x - 1, y) - U(x, y) * 2 + U(x + 1, y);

        V(x, y) = U(x, y) + 0.2f * (d2tdx2 + d2tdy2);
      } else {
        V(x, y) = U(x, y);
      }
    }, stream);
}
```

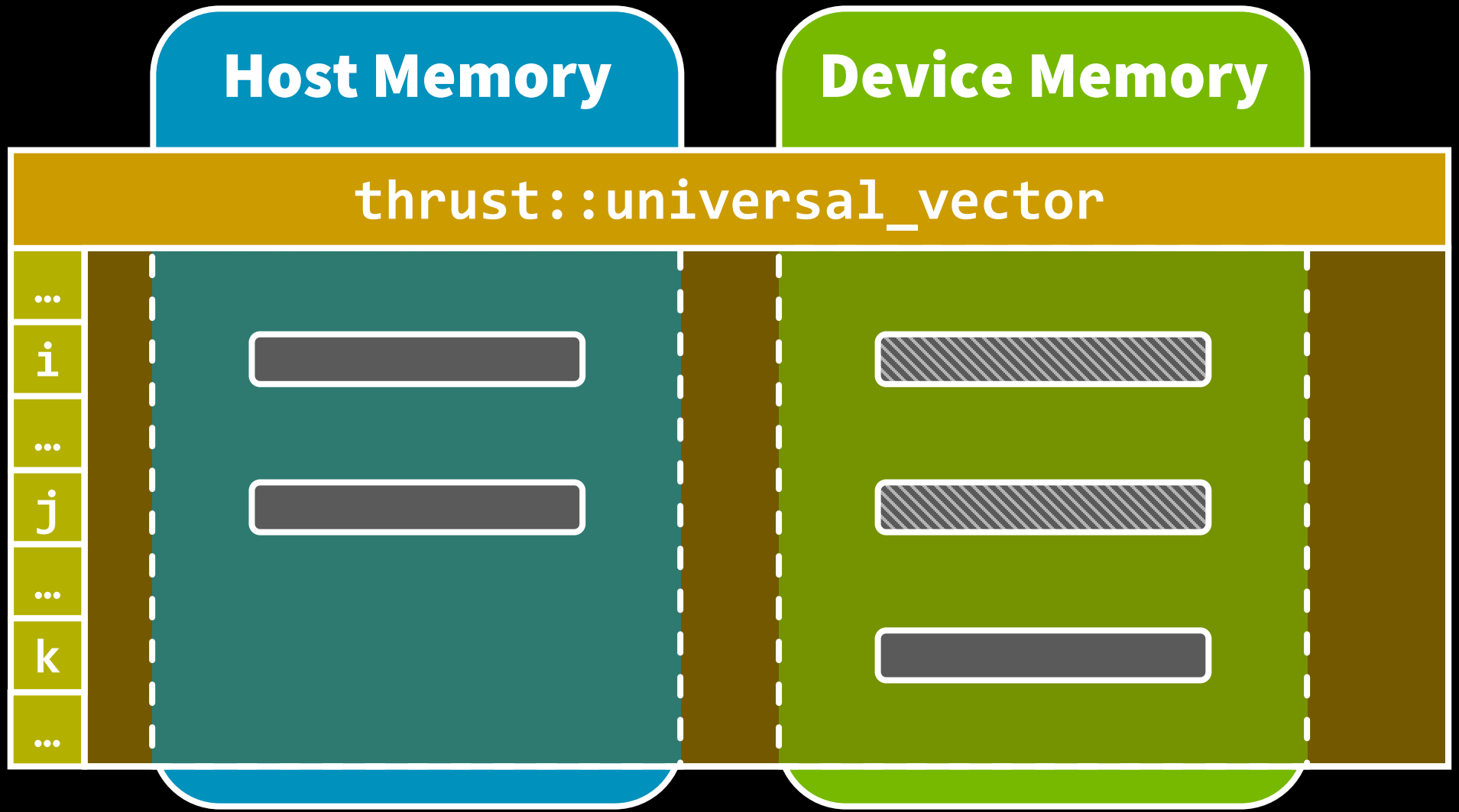


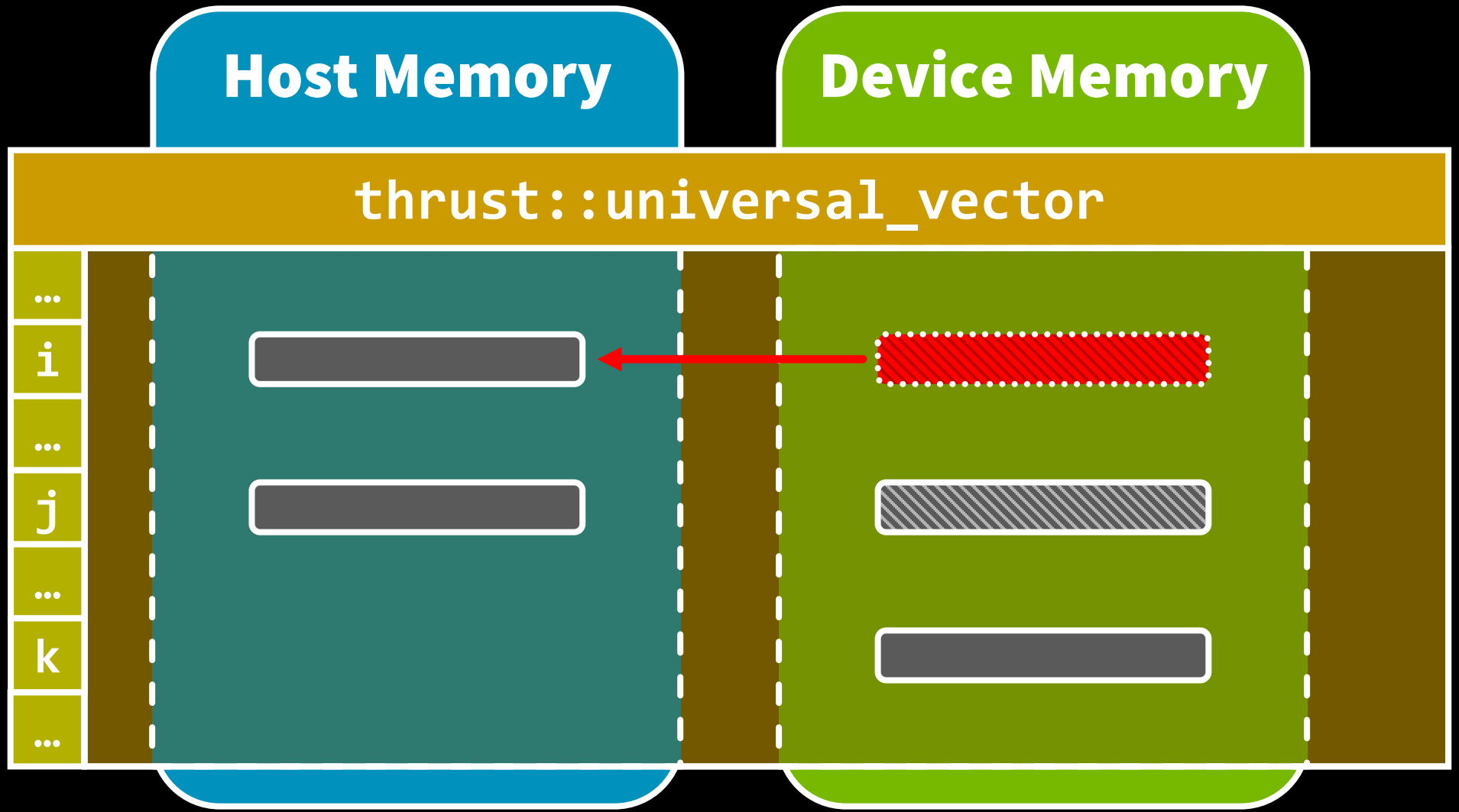


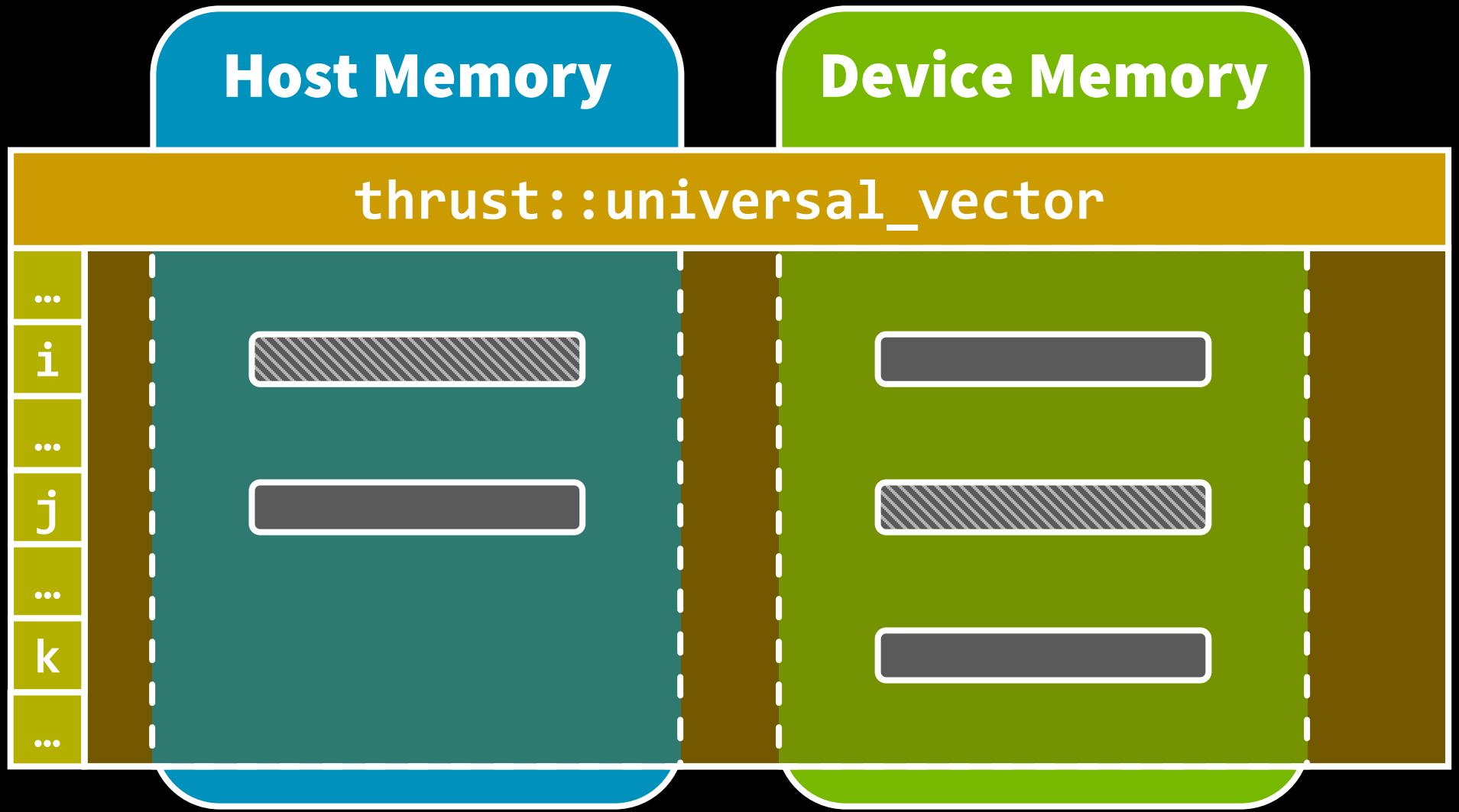


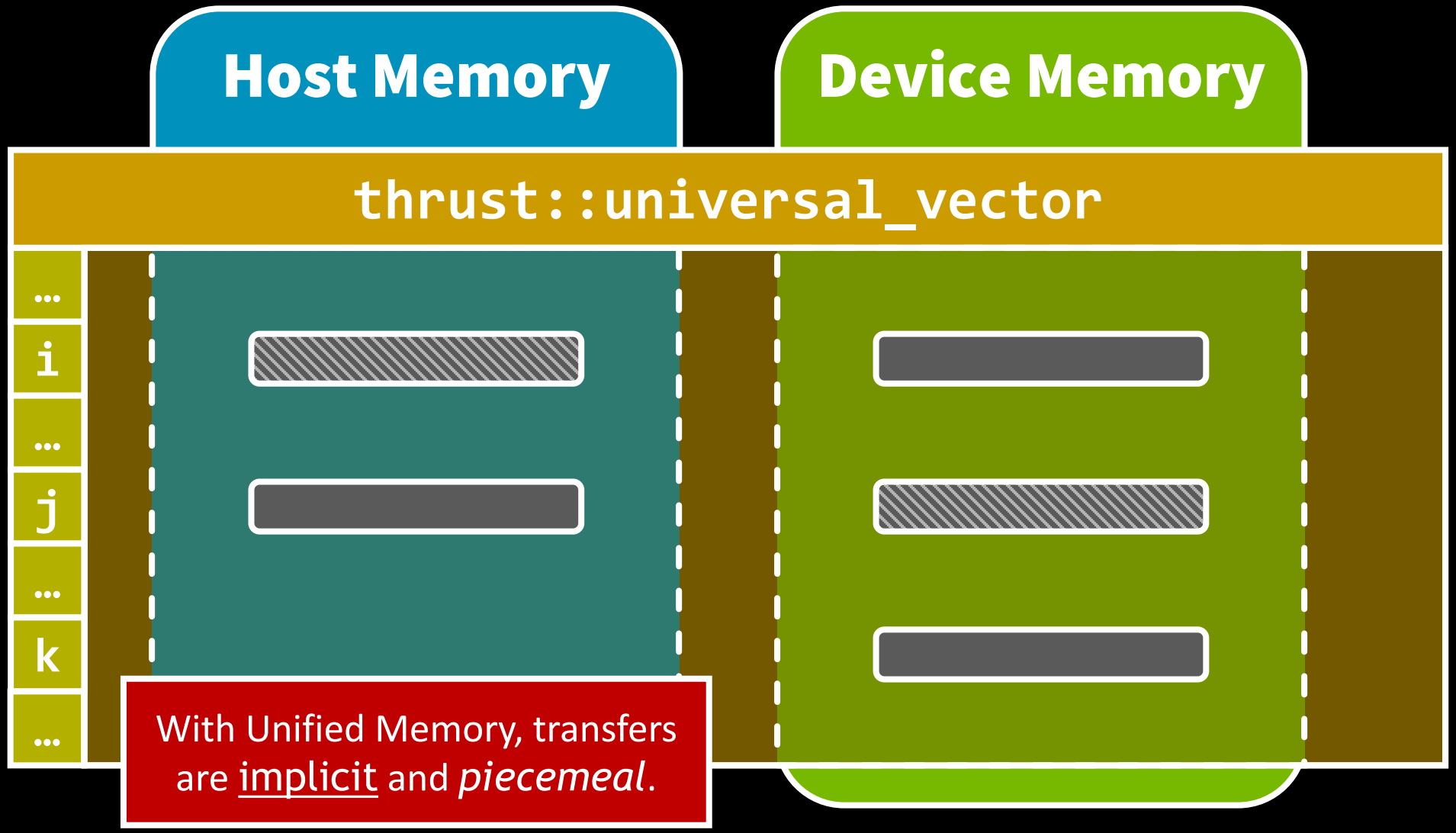












Thrust Container	Accessible in…	Transfers are…
`universal_vector`	Host & device code	Implicit upon access

Thrust Container	Accessible in…	Transfers are…
`universal_vector`	Host & device code	Implicit upon access
`host_vector`	Host code only	
`device_vector`	Device code only	

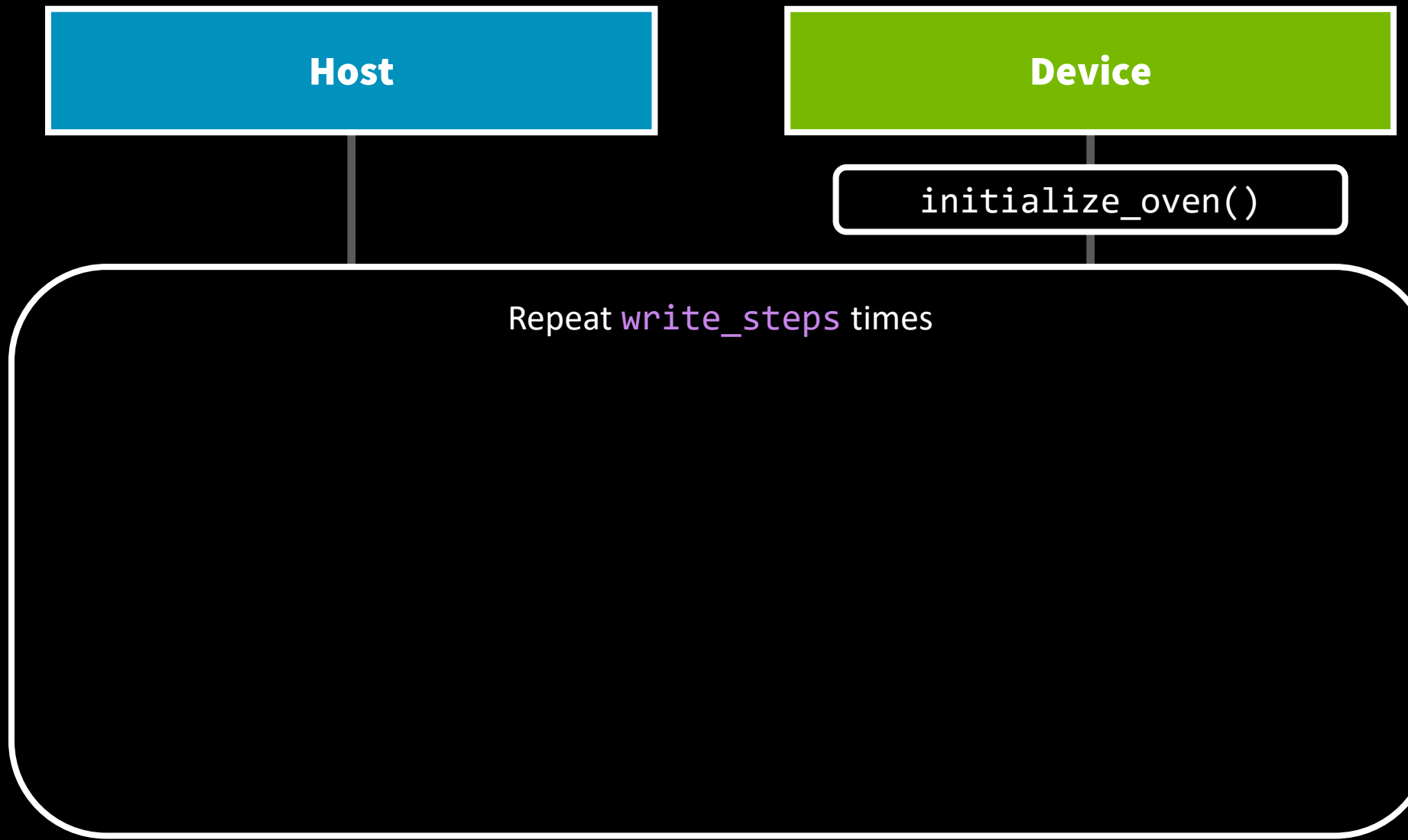
Thrust Container	Accessible in…	Transfers are…
`universal_vector`	Host & device code	Implicit upon access
`host_vector`	Host code only	Explicit via `thrust::copy`
`device_vector`	Device code only	

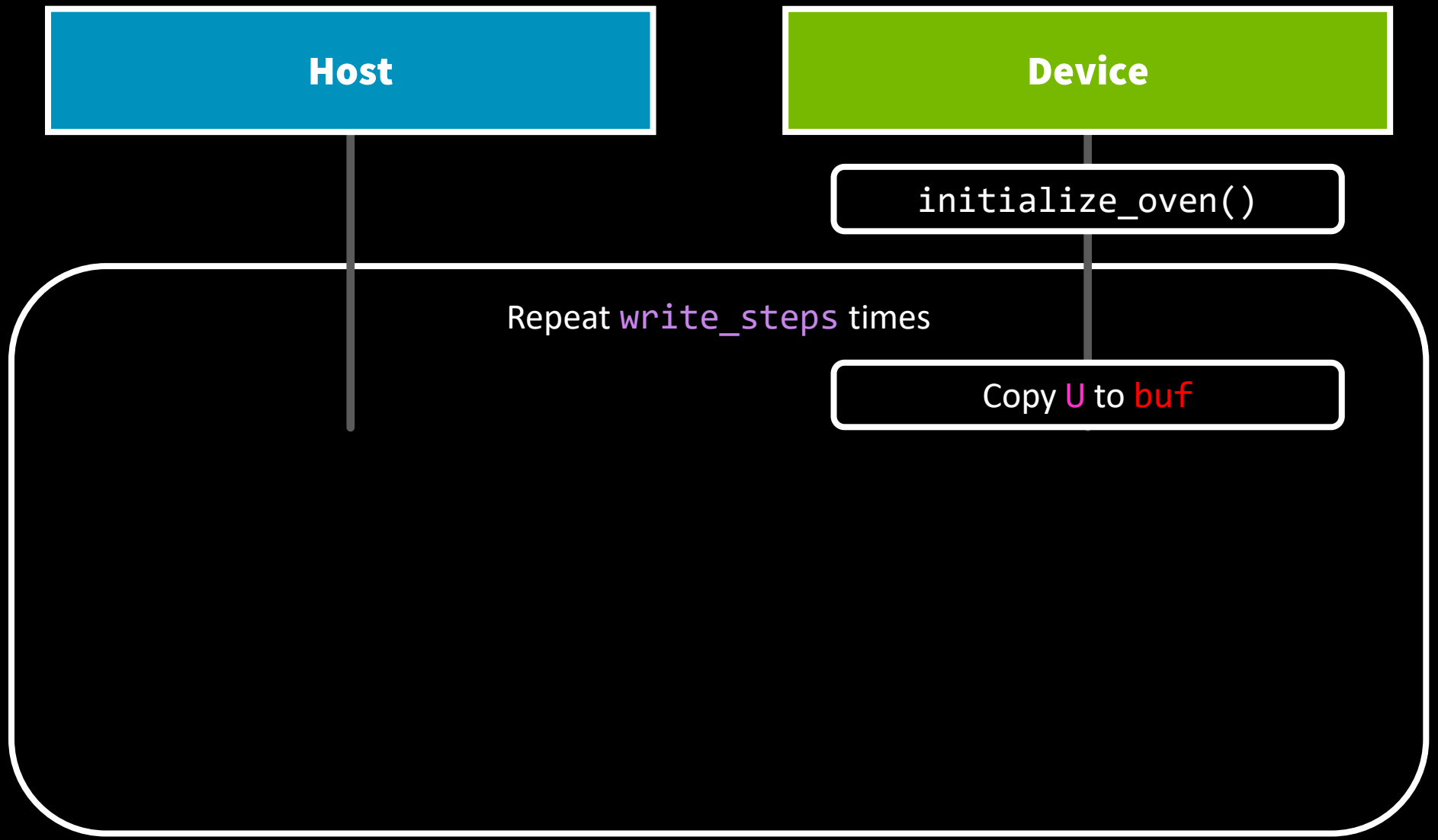
```
int nx(…), ny(…), write_steps(…), compute_steps(…);

cudax::stream stream;
auto policy(thrust::cuda::par_nosync.on(stream.get()));

thrust::universal_vector<float> U_data(nx * ny), V_data(nx * ny);
cuda::std::mdspan U(…), V(…);

initialize_oven(stream, U);

for (auto write_step : std::views::iota(0, write_steps)) {
  stream.wait();
  save_to_file(U);

  for (auto compute_step : std::views::iota(0, compute_steps)) {
    heat_equation(stream, U, V);
    U.swap(V);
  }
}
```

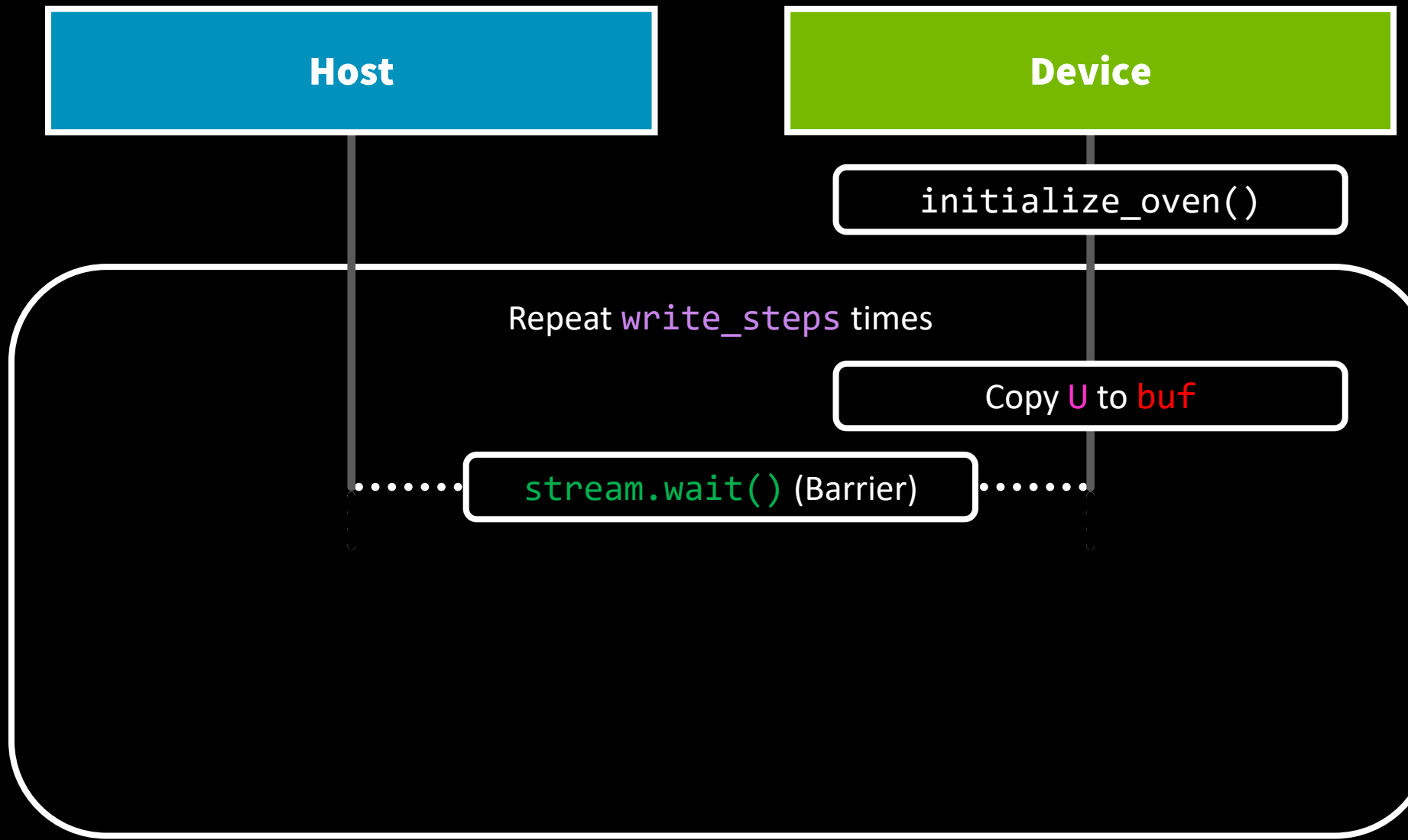
```
int nx(…), ny(…), write_steps(…), compute_steps(…);

cudax::stream stream;
auto policy(thrust::cuda::par_nosync.on(stream.get()));

thrust::universal_vector<float> U_data(nx * ny), V_data(nx * ny);
cuda::std::mdspan U(…), V(…);

initialize_oven(stream, U);

for (auto write_step : std::views::iota(0, write_steps)) {
  stream.wait();
  save_to_file(U); // Implicit device to host transfer.

  for (auto compute_step : std::views::iota(0, compute_steps)) {
    heat_equation(stream, U, V);
    U.swap(V);
  }
}
```

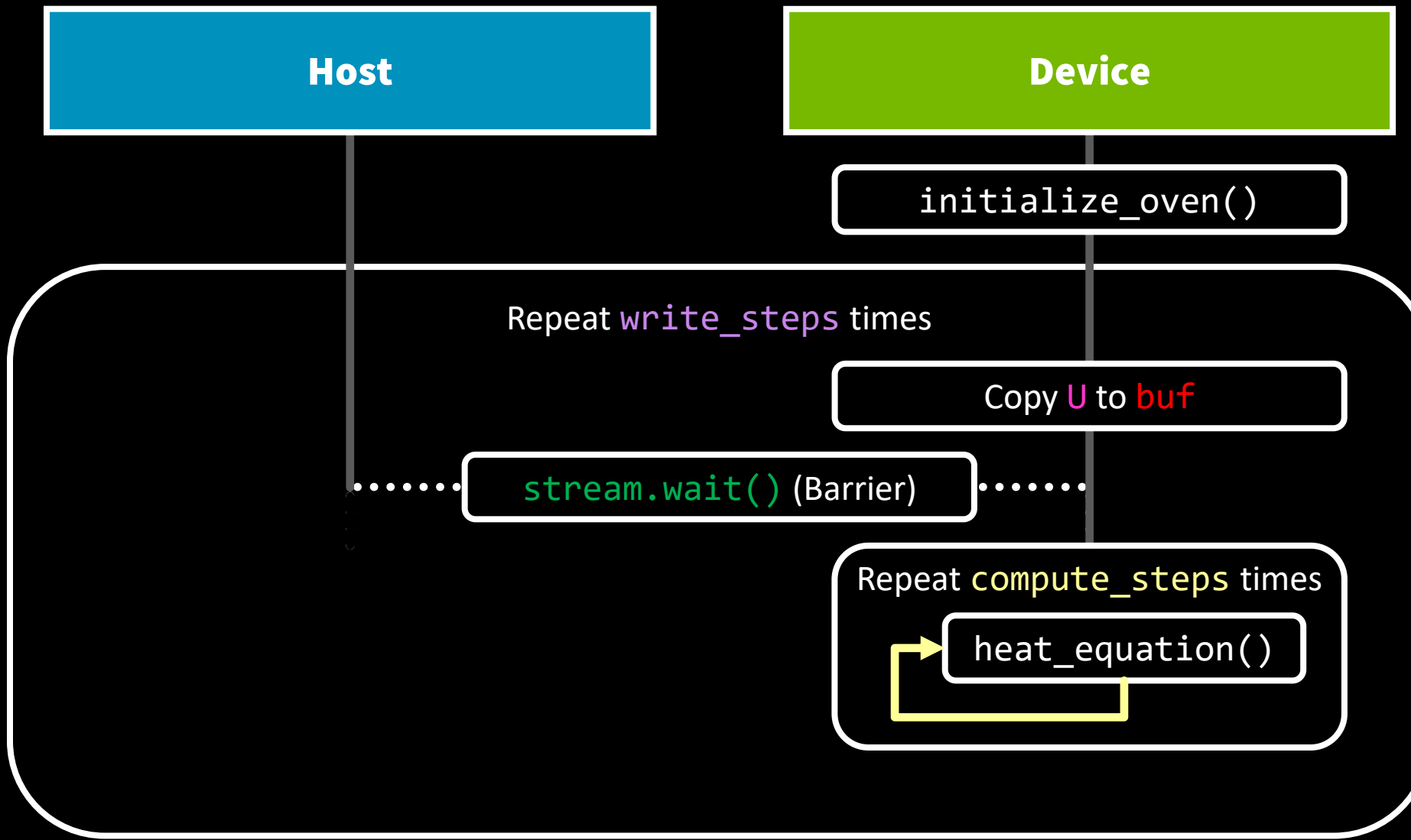
```
int nx(…), ny(…), write_steps(…), compute_steps(…);

cudax::stream stream;
auto policy(thrust::cuda::par_nosync.on(stream.get()));

thrust::universal_vector<float> U_data(nx * ny), V_data(nx * ny);
thrust::host_vector<float> buf_data(nx * ny);
cuda::std::mdspan U(…), V(…), buf(…);

initialize_oven(stream, U);

for (auto write_step : std::views::iota(0, write_steps)) {
  thrust::copy_n(policy, U.data(), U.size(), buf); // Device to host.
  stream.wait();

  for (auto compute_step : std::views::iota(0, compute_steps)) {
    heat_equation(stream, U, V);
    U.swap(V);
  }

  save_to_file(buf); // No transfer.
}
```

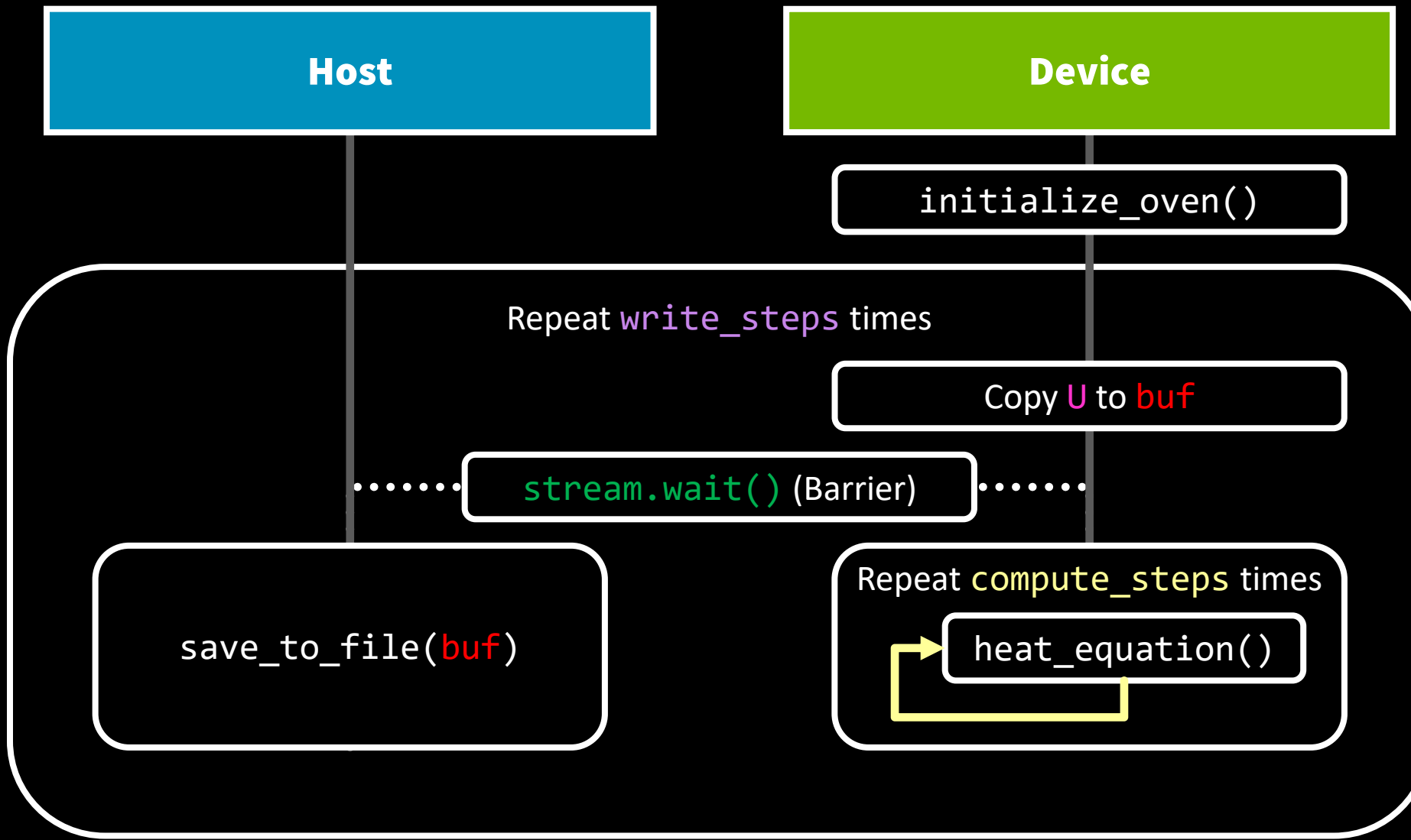
```
int nx(…), ny(…), write_steps(…), compute_steps(…);

cudax::stream stream;
auto policy(thrust::cuda::par_nosync.on(stream.get()));

thrust::universal_vector<float> U_data(nx * ny), V_data(nx * ny);
thrust::host_vector<float> buf_data(nx * ny);
cuda::std::mdspan U(…), V(…), buf(…);

initialize_oven(stream, U);

for (auto write_step : std::views::iota(0, write_steps)) {
  thrust::copy_n(policy, U.data(), U.size(), buf); // Device to host.
  stream.wait();

  for (auto compute_step : std::views::iota(0, compute_steps)) {
    heat_equation(stream, U, V);
    U.swap(V);
  }

  save_to_file(buf); // No transfer.
}
```

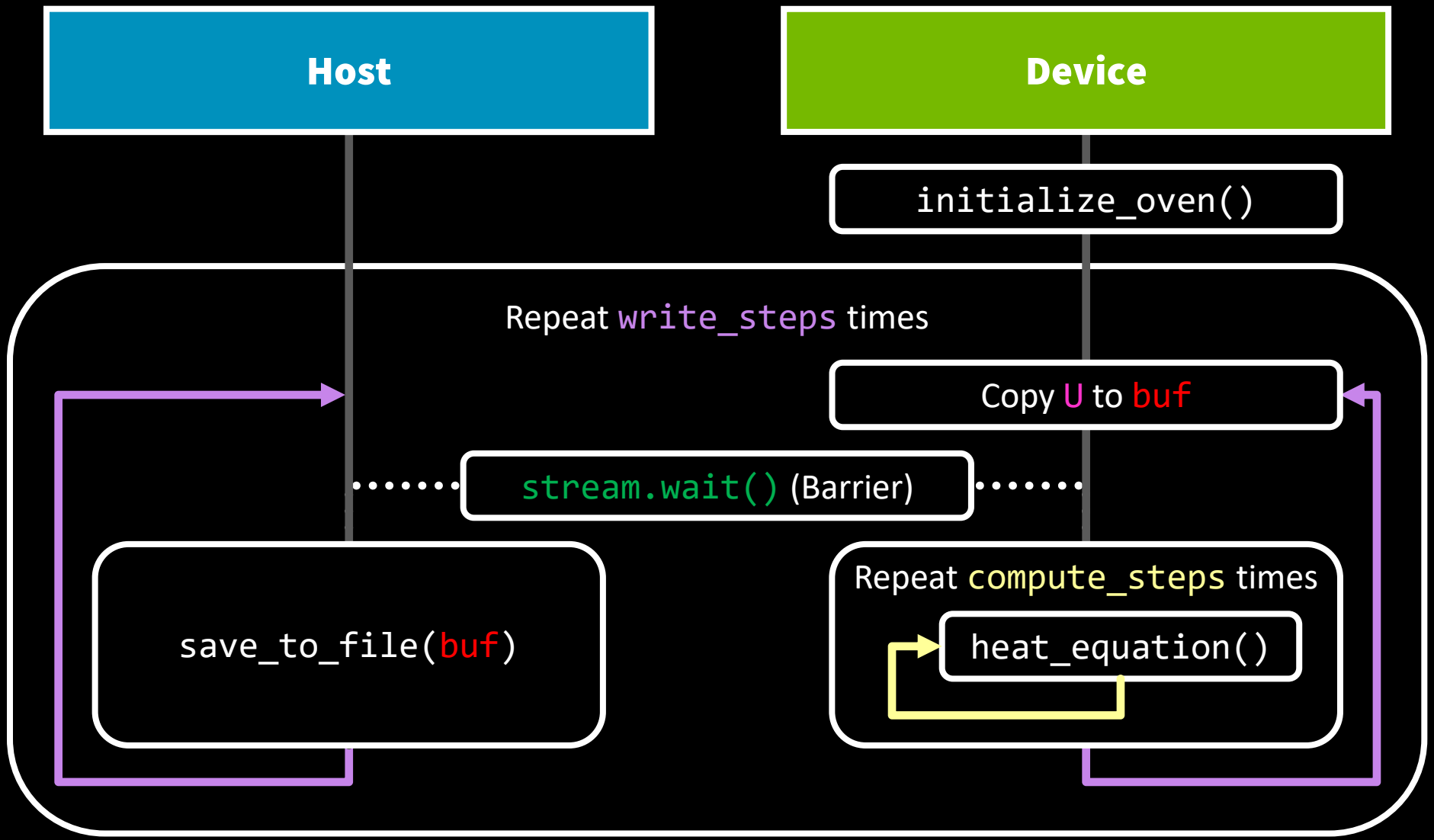


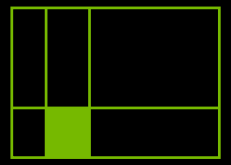




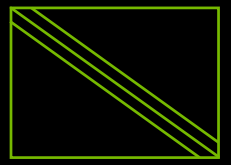






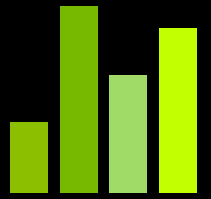


cuBLAS

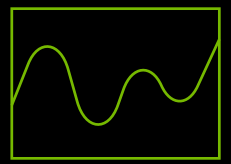


cuSPARSE

cuFile (GDS)



NVBench



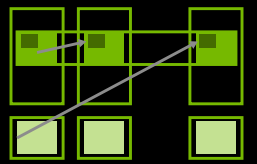
cuFFT

cuDSS

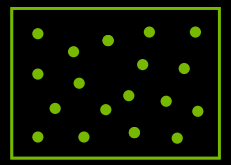
CV CUDA

cuLitho

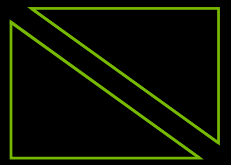
Cooperative Groups



NVSHMEM



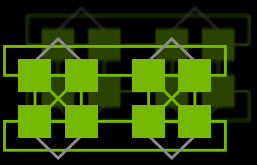
cuRAND



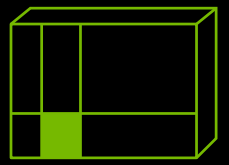
cuSOLVER

CUDA-Q

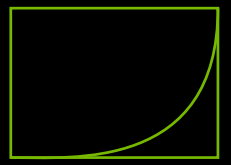
CUTLASS



NCCL



cuTENSOR



CUDA Math

CUDA Runtime

GTC CUDA Developer Track
nvda.ws/4ccb7Qh

Accelerated Computing Hub
nvda.ws/3QZRuRI

GENERAL CUDA

S72571 What's CUDA All About Anyways?
S72897 How To Write A CUDA Program
S72527 Debugging & Optimizing CUDA with Intelligent Developer Tools

CUDA PYTHON

S72450 Accelerated Python: Tour of Community & Ecosystem
S72448 The CUDA Python Developer's Toolbox
S72449 1001 Ways to Write CUDA Kernels in Python
S74639 Tensor Core Programming in Python with CUTLASS 4.0

CUDA C++

S72574 Building CUDA Software at the Speed-of-Light
S72572 The CUDA C++ Developer's Toolbox
S72575 How You Should Write a CUDA C++ Kernel

MULTI-GPU PROGRAMMING

S72576 Getting Started with Multi-GPU Scaling: Distributed Libraries
S72579 Going Deeper with Multi-GPU Scaling: Task-based Runtimes
S72578 Advanced Multi-GPU Scaling: Communication Libraries

PERFORMANCE OPTIMIZATION

S72683 CUDA Techniques to Maximize Memory Bandwidth & Hide Latency
S72685 CUDA Techniques to Maximize Compute & Instruction Throughput
S72686 CUDA Techniques to Maximize Concurrency & System Utilization
S72687 Get the Most Performance From Grace Hopper

GTC CUDA Developer Track
nvda.ws/4ccb7Qh

Accelerated Computing Hub
nvda.ws/3QZRuRI

GENERAL CUDA

S72571 What's CUDA All About Anyways?
S72897 How To Write A CUDA Program
S72527 Debugging & Optimizing CUDA with Intelligent Developer Tools

CUDA PYTHON

S72450 Accelerated Python: Tour of Community & Ecosystem
S72448 The CUDA Python Developer's Toolbox
S72449 1001 Ways to Write CUDA Kernels in Python
S74639 Tensor Core Programming in Python with CUTLASS 4.0

CUDA C++

S72574 Building CUDA Software at the Speed-of-Light
S72572 The CUDA C++ Developer's Toolbox
S72575 How You Should Write a CUDA C++ Kernel

MULTI-GPU PROGRAMMING

S72576 Getting Started with Multi-GPU Scaling: Distributed Libraries
S72579 Going Deeper with Multi-GPU Scaling: Task-based Runtimes
S72578 Advanced Multi-GPU Scaling: Communication Libraries

PERFORMANCE OPTIMIZATION

S72683 CUDA Techniques to Maximize Memory Bandwidth & Hide Latency
S72685 CUDA Techniques to Maximize Compute & Instruction Throughput
S72686 CUDA Techniques to Maximize Concurrency & System Utilization
S72687 Get the Most Performance From Grace Hopper