Revamping llm.c with the CUDA C++ Core Libraries (CCCL)

Jake Hemstad, Software Engineering Manager, NVIDIA

Georgii Evtushenko, Senior Software Engineer, NVIDIA

CUDA C++ Core Libraries (CCCL)

Mission: We make CUDA C++ a speed-of-light delight

Library	Key Features	Device APIs	Host APIs	Availability	
				GitHub	CUDA Toolkit
Thrust	• High-level CPU/GPU parallel algorithms	✓	✓+	✓	✓
CUB	• Low-level GPU parallel algorithms	✓+	✓	✓	✓
libcu++	• Heterogeneous C++ Standard Library • Hardware feature abstractions	✓	✓	✓	✓
Cooperative Groups	• Name, synchronize, and communicate among thread groups	✓	✗	🏗️	✓
nvbench	• Framework for CUDA-aware benchmarking	✗	✓	✓	✗

github.com/NVIDIA/cccl

The CUDA C++ Core Libraries

```
Standard C++ = C++ Language
             + Standard Library
```

The C++ Standard Library provides…

- General purpose abstractions
- Data structures
- Algorithms

…that simplify and enhance C++ applications

Without the Standard Library, C++ is tedious and error-prone

```
CUDA C++ = C++ Language
         + Host Standard Library
         + CUDA Language Extensions
         + CUDA C++ Core Libraries
```

The CUDA C++ Core Libraries provide…

- *Heterogeneous* C++ Standard Library
- Fundamental CUDA abstractions
- High-performance parallel algorithms

…that simplify and enhance CUDA C++ applications

Without CCCL, CUDA C++ is tedious and error-prone

The CUDA C++ Spectrum

Finding the Right Tool for the Job

Speed-of-Light Abstractions

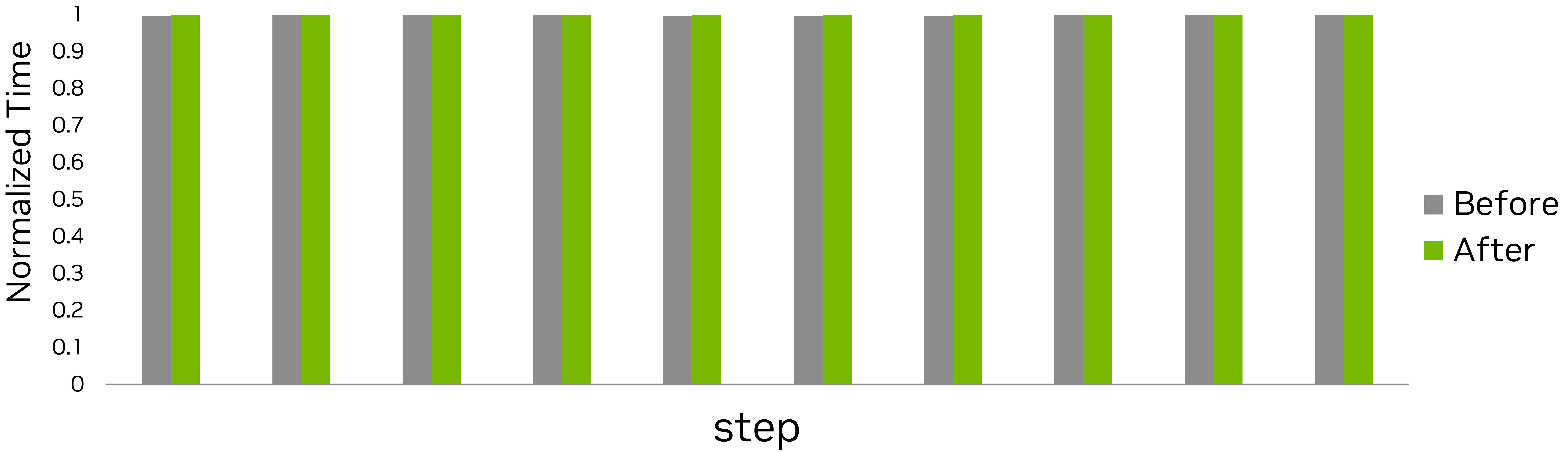
Performance comparison on Hopper

Before modifications

```
step 61/74: train loss 3.308659 (36.644057 ms)
step 62/74: train loss 3.258297 (36.643250 ms)
step 63/74: train loss 3.308180 (36.675600 ms)
step 64/74: train loss 3.744773 (36.681992 ms)
step 65/74: train loss 3.525836 (36.666792 ms)
step 66/74: train loss 3.304968 (36.637838 ms)
step 67/74: train loss 3.218528 (36.645540 ms)
step 68/74: train loss 3.406032 (36.685613 ms)
step 69/74: train loss 3.252334 (36.704001 ms)
step 70/74: train loss 3.077519 (36.681118 ms)
step 71/74: train loss 3.067738 (36.646326 ms)
step 72/74: train loss 3.085185 (36.732816 ms)
step 73/74: train loss 3.667693 (36.654056 ms)
step 74/74: train loss 3.467426 (36.672249 ms)
```

After modifications

```
step 61/74: train loss 3.309026 (36.734715 ms)
step 62/74: train loss 3.258694 (36.704587 ms)
step 63/74: train loss 3.308773 (36.689833 ms)
step 64/74: train loss 3.745108 (36.700659 ms)
step 65/74: train loss 3.525969 (36.770350 ms)
step 66/74: train loss 3.305243 (36.754562 ms)
step 67/74: train loss 3.219051 (36.744185 ms)
step 68/74: train loss 3.405844 (36.662269 ms)
step 69/74: train loss 3.252328 (36.682613 ms)
step 70/74: train loss 3.076691 (36.758848 ms)
step 71/74: train loss 3.067311 (36.768566 ms)
step 72/74: train loss 3.085387 (36.751530 ms)
step 73/74: train loss 3.667906 (36.697977 ms)
step 74/74: train loss 3.467444 (36.697323 ms)
```



Build System

current

```
CC ?= clang
CFLAGS = -O3 -Ofast -Wno-unused-result -march=native
LDFLAGS =
LDLIBS = -lm
INCLUDES =

# Check if OpenMP is available
# This is done by attempting to compile an empty file with OpenMP flags
# OpenMP makes the code a lot faster so I advise installing it
# e.g. on MacOS: brew install libomp
# e.g. on Ubuntu: sudo apt-get install libomp-dev
# later, run the program by prepending the number of threads, e.g.: OMP_NUM_THREADS=8
./gpt2
ifeq ($(shell uname), Darwin)
  # Check if the libomp directory exists
  ifeq ($(shell [ -d /opt/homebrew/opt/libomp/lib ] && echo "exists"), exists)
    # macOS with Homebrew and directory exists
    CFLAGS += -Xclang -fopenmp -DOMP
    LDFLAGS += -L/opt/homebrew/opt/libomp/lib
    LDLIBS += -lomp
    INCLUDES += -I/opt/homebrew/opt/libomp/include
    $(info NICE Compiling with OpenMP support)
  else ifeq ($(shell [ -d /usr/local/opt/libomp/lib ] && echo "exists"), exists)
    CFLAGS += -Xclang -fopenmp -DOMP
    LDFLAGS += -L/usr/local/opt/libomp/lib
    LDLIBS += -lomp
    INCLUDES += -I/usr/local/opt/libomp/include
    $(info NICE Compiling with OpenMP support)
  else
    $(warning OOPS Compiling without OpenMP support)
  endif
else
  ifeq ($(shell echo | $(CC) -fopenmp -x c -E - > /dev/null 2>&1; echo $$?), 0)
    # Ubuntu or other Linux distributions
    CFLAGS += -fopenmp -DOMP
    LDLIBS += -lgomp
    $(info NICE Compiling with OpenMP support)
  else
    $(warning OOPS Compiling without OpenMP support)
  endif
endif

# PHONY means these targets will always be executed
.PHONY: all train_gpt2 test_gpt2 train_gpt2cu test_gpt2cu

# default target is all
all: train_gpt2 test_gpt2 train_gpt2cu test_gpt2cu

train_gpt2: train_gpt2.c
  $(CC) $(CFLAGS) $(INCLUDES) $(LDFLAGS) $< $(LDLIBS) -o $@

test_gpt2: test_gpt2.c
  $(CC) $(CFLAGS) $(INCLUDES) $(LDFLAGS) $< $(LDLIBS) -o $@

# possibly may want to disable warnings? e.g. append -Xcompiler -Wno-unused-result
train_gpt2cu: train_gpt2.cu
  nvcc -O3 --use_fast_math $< -lcublas -lcublasLt -o $@

test_gpt2cu: test_gpt2.cu
  nvcc -O3 --use_fast_math $< -lcublas -lcublasLt -o $@

clean:
  rm -f train_gpt2 test_gpt2 train_gpt2cu test_gpt2cu
```

alternative

```
cmake_minimum_required(VERSION 3.25.0)
project(train_gpt2cu LANGUAGES C CXX CUDA)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CUDA_STANDARD 20)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${CMAKE_SOURCE_DIR}")
set(CMAKE_CUDA_ARCHITECTURES "native")

include(cmake/CPM.cmake)
CPMAddPackage("gh:NVIDIA/cccl#main")    # For demonstration, CCCL is also available from CTK
CPMAddPackage("gh:NVIDIA/nvbench#main")

find_package(OpenMP REQUIRED)
find_package(CUDAToolkit)

add_executable(train_gpt2 train_gpt2.cpp)
target_link_libraries(train_gpt2 PRIVATE OpenMP::OpenMP_CXX)

add_executable(train_gpt2cu train_gpt2.cu)
target_link_libraries(train_gpt2cu CCCL::CCCL CUDA::cublas CUDA::cublasLt)
target_compile_options(train_gpt2cu
  PRIVATE -O3 $<$<COMPILE_LANGUAGE:CUDA>:--use_fast_math --extended-lambda>)
```

- Same code gen
- Cross-platform (works on Windows)
- Reduced compiler dependencies
- Less error-prone (warns about missing CUDA arch)
- Setup-free dependency management

Build System

current

```makefile
CC ?= clang
CFLAGS = -O3 -Ofast -Wno-unused-result -march=native
LDFLAGS =
LDLIBS = -lm
INCLUDES =

# Check if OpenMP is available
# This is done by attempting to compile an empty file with OpenMP flags
# OpenMP makes the code a lot faster so I advise installing it
# e.g. on MacOS: brew install libomp
# e.g. on Ubuntu: sudo apt-get install libomp-dev
# later, run the program by prepending the number of threads, e.g.: OMP_NUM_THREADS=8
./gpt2
ifeq ($(shell uname), Darwin)
  # Check if the libomp directory exists
  ifeq ($(shell [ -d /opt/homebrew/opt/libomp/lib ] && echo "exists"), exists)
    # macOS with Homebrew and directory exists
    CFLAGS += -Xclang -fopenmp -DOMP
    LDFLAGS += -L/opt/homebrew/opt/libomp/lib
    LDLIBS += -lomp
    INCLUDES += -I/opt/homebrew/opt/libomp/include
    $(info NICE Compiling with OpenMP support)
  else ifeq ($(shell [ -d /usr/local/opt/libomp/lib ] && echo "exists"), exists)
    CFLAGS += -Xclang -fopenmp -DOMP
    LDFLAGS += -L/usr/local/opt/libomp/lib
    LDLIBS += -lomp
    INCLUDES += -I/usr/local/opt/libomp/include
    $(info NICE Compiling with OpenMP support)
  else
    $(warning OOPS Compiling without OpenMP support)
  endif
else
  ifeq ($(shell echo | $(CC) -fopenmp -x c -E - > /dev/null 2>&1; echo $$?), 0)
    # Ubuntu or other Linux distributions
    CFLAGS += -fopenmp -DOMP
    LDLIBS += -lgomp
    $(info NICE Compiling with OpenMP support)
  else
    $(warning OOPS Compiling without OpenMP support)
  endif
endif

# PHONY means these targets will always be executed
.PHONY: all train_gpt2 test_gpt2 train_gpt2cu test_gpt2cu

# default target is all
all: train_gpt2 test_gpt2 train_gpt2cu test_gpt2cu

train_gpt2: train_gpt2.c
  $(CC) $(CFLAGS) $(INCLUDES) $(LDFLAGS) $< $(LDLIBS) -o $@

test_gpt2: test_gpt2.c
  $(CC) $(CFLAGS) $(INCLUDES) $(LDFLAGS) $< $(LDLIBS) -o $@

# possibly may want to disable warnings? e.g. append -Xcompiler -Wno-unused-result
train_gpt2cu: train_gpt2.cu
  nvcc -O3 --use_fast_math $< -lcublas -lcublasLt -o $@

test_gpt2cu: test_gpt2.cu
  nvcc -O3 --use_fast_math $< -lcublas -lcublasLt -o $@

clean:
  rm -f train_gpt2 test_gpt2 train_gpt2cu test_gpt2cu
```

alternative

```cmake
cmake_minimum_required(VERSION 3.25.0)
project(train_gpt2cu LANGUAGES C CXX CUDA)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CUDA_STANDARD 20)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${CMAKE_SOURCE_DIR}")
set(CMAKE_CUDA_ARCHITECTURES "native")

include(cmake/CPM.cmake)
CPMAddPackage("gh:NVIDIA/cccl#main")
CPMAddPackage("gh:NVIDIA/nvbench#main")

find_package(OpenMP REQUIRED)
find_package(CUDAToolkit)

add_executable(train_gpt2 train_gpt2.cpp)
target_link_libraries(train_gpt2 PRIVATE OpenMP::OpenMP_CXX)

add_executable(train_gpt2cu train_gpt2.cu)
target_link_libraries(train_gpt2cu CCCL::CCCL CUDA::cublas CUDA::cublasLt)
target_compile_options(train_gpt2cu
  PRIVATE -O3 $<$<COMPILE_LANGUAGE:CUDA>:--use_fast_math --extended-lambda>)
```

- Same code gen
- Cross-platform (works on Windows)
- Reduced compiler dependencies
- Less error-prone (warns about missing CUDA arch)
- Setup-free dependency management

Memory Management

Thrust containers

current

```
int B = 8;
int T = 1024;
int C = 768;

float* out = (float*)malloc(B * T * C * sizeof(float));
float* mean = (float*)malloc(B * T * sizeof(float));
float* rstd = (float*)malloc(B * T * sizeof(float));
float* inp = make_random_float(B * T * C);
float* weight = make_random_float(C);
float* bias = make_random_float(C);

float* d_out;
float* d_mean;
float* d_rstd;
float* d_inp;
float* d_weight;
float* d_bias;
cudaCheck(cudaMalloc(&d_out, B * T * C * sizeof(float)));
cudaCheck(cudaMalloc(&d_mean, B * T * sizeof(float)));
cudaCheck(cudaMalloc(&d_rstd, B * T * sizeof(float)));
cudaCheck(cudaMalloc(&d_inp, B * T * C * sizeof(float)));
cudaCheck(cudaMalloc(&d_weight, C * sizeof(float)));
cudaCheck(cudaMalloc(&d_bias, C * sizeof(float)));
cudaCheck(cudaMemcpy(d_inp, inp, B * T * C * sizeof(float),
                     cudaMemcpyHostToDevice));
cudaCheck(cudaMemcpy(d_weight, weight, C * sizeof(float),
                     cudaMemcpyHostToDevice));
cudaCheck(cudaMemcpy(d_bias, bias, C * sizeof(float),
                     cudaMemcpyHostToDevice));

int block_sizes[] = {32, 64, 128, 256, 512, 1024};
float* out_gpu = (float*)malloc(B * T * C * sizeof(float));
float* mean_gpu = (float*)malloc(B * T * sizeof(float));
float* rstd_gpu = (float*)malloc(B * T * sizeof(float));

/// ...

free(out);
free(mean);
free(rstd);
free(inp);
free(weight);
free(bias);
cudaCheck(cudaFree(d_out));
cudaCheck(cudaFree(d_mean));
cudaCheck(cudaFree(d_rstd));
cudaCheck(cudaFree(d_inp));
cudaCheck(cudaFree(d_weight));
cudaCheck(cudaFree(d_bias));
```

alternative

```
int B = 8;
int T = 1024;
int C = 768;

thrust::host_vector<float> h_inp(B * T * C);
thrust::host_vector<float> h_weight(C);
thrust::host_vector<float> h_bias(C);

thrust::default_random_engine gen(42);
thrust::uniform_real_distribution<float> dis(-1.0f, 1.0f);
thrust::generate(h_inp.begin(), h_inp.end(), [&] { return dis(gen); });
thrust::generate(h_weight.begin(), h_weight.end(), [&] { return dis(gen); });
thrust::generate(h_bias.begin(), h_bias.end(), [&] { return dis(gen); });

thrust::device_vector<float> d_out(B * T * C);
thrust::device_vector<float> d_mean(B * T);
thrust::device_vector<float> d_rstd(B * T);
thrust::device_vector<float> d_inp(h_inp);
thrust::device_vector<float> d_weight(h_weight);
thrust::device_vector<float> d_bias(h_bias);

template <class T>
using pinned_vector = thrust::host_vector<
  T, thrust::mr::stateless_resource_allocator<
    T, thrust::system::cuda::universal_host_pinned_memory_resource>>;
```

- Type-safe
- Less error-prone
- Customizable

Memory Management

Thrust containers

current

```
int B = 8;
int T = 1024;
int C = 768;

float* out = (float*)malloc(B * T * C * sizeof(float));
float* mean = (float*)malloc(B * T * sizeof(float));
float* rstd = (float*)malloc(B * T * sizeof(float));
float* inp = make_random_float(B * T * C);
float* weight = make_random_float(C);
float* bias = make_random_float(C);

float* d_out;
float* d_mean;
float* d_rstd;
float* d_inp;
float* d_weight;
float* d_bias;
cudaCheck(cudaMalloc(&d_out, B
cudaCheck(cudaMalloc(&d_mean, B
cudaCheck(cudaMalloc(&d_rstd, B
cudaCheck(cudaMalloc(&d_inp, B
cudaCheck(cudaMalloc(&d_weight,
cudaCheck(cudaMalloc(&d_bias, C
cudaCheck(cudaMemcpy(d_inp, inp
          cudaMemcpy
cudaCheck(cudaMemcpy(d_weight,
          cudaMemcpy
cudaCheck(cudaMemcpy(d_bias, bi
          cudaMemcpy

int block_sizes[] = {32, 64, 12
float* out_gpu = (float*)malloc(B * T * C * sizeof(float));
float* mean_gpu = (float*)malloc(B * T * sizeof(float));
float* rstd_gpu = (float*)malloc(B * T * sizeof(float));

/// ...

free(out);
free(mean);
free(rstd);
free(inp);
free(weight);
free(bias);
cudaCheck(cudaFree(d_out));
cudaCheck(cudaFree(d_mean));
cudaCheck(cudaFree(d_rstd));
cudaCheck(cudaFree(d_inp));
cudaCheck(cudaFree(d_weight));
cudaCheck(cudaFree(d_bias));
```

alternative

```
int B = 8;
int T = 1024;
int C = 768;

thrust::host_vector<float> h_inp(B * T * C);
thrust::host_vector<float> h_weight(C);
thrust::host_vector<float> h_bias(C);

thrust::default_random_engine gen(42);
thrust::uniform_real_distribution<float> dis(-1.0f, 1.0f);
thrust::generate(h_inp.begin(), h_inp.end(), [&] { return dis(gen); });
thrust::generate(h_weight.begin(), h_weight.end(), [&] { return dis(gen); });
                                                                      });

using pinned_vector = thrust::host_vector<
  T, thrust::mr::stateless_resource_allocator<
     T, thrust::system::cuda::universal_host_pinned_memory_resource>>;
```

```
// Compiles successfully (but shouldn't)
cuda::std::complex<float>* d_complex{};
int* d_int{};
cudaMemcpy(d_int, d_complex, sizeof(cuda::std::complex<float>), cudaMemcpyDeviceToDevice);

// Detects the issue at compile time
thrust::device_vector<cuda::std::complex<float>> complex_vec(10);
thrust::device_vector<int> int_vec = complex_vec;
```

- Type-safe
- Less error-prone
- Customizable

Memory Management

Thrust containers

current

```
int B = 8;
int T = 1024;
int C = 768;

float* out = (float*)malloc(B * T * C * sizeof(float));
float* mean = (float*)malloc(B * T * sizeof(float));
float* rstd = (float*)malloc(B * T * sizeof(float));
float* inp = make_random_float(B * T * C);
float* weight = make_random_float(C);
float* bias = make_random_float(C);

float* d_out;
float* d_mean;
float* d_rstd;
float* d_inp;
float* d_weight;
float* d_bias;
cudaCheck(cudaMalloc(&d_out, B
cudaCheck(cudaMalloc(&d_mean, B
cudaCheck(cudaMalloc(&d_rstd, B
cudaCheck(cudaMalloc(&d_inp, B
cudaCheck(cudaMalloc(&d_weight,
cudaCheck(cudaMalloc(&d_bias, C
cudaCheck(cudaMemcpy(d_inp, inp
                     cudaMemcpy
cudaCheck(cudaMemcpy(d_weight,
                     cudaMemcpy
cudaCheck(cudaMemcpy(d_bias, bi
                     cudaMemcpy

int block_sizes[] = {32, 64, 12
float* out_gpu = (float*)malloc
float* mean_gpu = (float*)malloc
float* rstd_gpu = (float*)malloc(B * T * sizeof(float));

/// ...

free(out);
free(mean);
free(rstd);
free(inp);
free(weight);
free(bias);
cudaCheck(cudaFree(d_out));
cudaCheck(cudaFree(d_mean));
cudaCheck(cudaFree(d_rstd));
cudaCheck(cudaFree(d_inp));
cudaCheck(cudaFree(d_weight));
cudaCheck(cudaFree(d_bias));
```

alternative

```
int B = 8;
int T = 1024;
int C = 768;

thrust::host_vector<float> h_inp(B * T * C);
thrust::host_vector<float> h_weight(C);
thrust::host_vector<float> h_bias(C);

thrust::default_random_engine gen(42);
thrust::uniform_real_distribution<float> dis(-1.0f, 1.0f);
thrust::generate(h_inp.begin(), h_inp.end(), [&] { return dis(gen); });
thrust::generate(h_weight.begin(), h_weight.end(), [&] { return dis(gen); });
                                                                        });
```

```
// type punning
float *d_float{};
cudaMalloc(&d_float, sizeof(float));

int val = 42;
cudaMemcpy(d_float, &val, sizeof(float), cudaMemcpyHostToDevice); // d_float[0] = 5.88545e-44

// vs conversion
thrust::device_vector<float> d_vec(1, 42);                        // d_vec[0]   = 42.0f
```

```
  T, thrust::mr::stateless_resource_allocator<
    T, thrust::system::cuda::universal_host_pinned_memory_resource>>;
```

- Type-safe
- Less error-prone
- Customizable

Memory Management

Thrust containers

current

```
int B = 8;
int T = 1024;
int C = 768;

float* out = (float*)malloc(B * T * C * sizeof(float));
float* mean = (float*)malloc(B * T * sizeof(float));
float* rstd = (float*)malloc(B * T * sizeof(float));
float* inp = make_random_float(B * T * C);
float* weight = make_random_float(C);
float* bias = make_random_float(C);

float* d_out;
float* d_mean;
float* d_rstd;
float* d_inp;
float* d_weight;
float* d_bias;
cudaCheck(cudaMalloc(&d_out, B * T * C * sizeof(float)));
cudaCheck(cudaMalloc(&d_mean, B * T * sizeof(float)));
cudaCheck(cudaMalloc(&d_rstd, B * T * sizeof(float)));
cudaCheck(cudaMalloc(&d_inp, B * T * C * sizeof(float)));
cudaCheck(cudaMalloc(&d_weight, C * sizeof(float)));
cudaCheck(cudaMalloc(&d_bias, C * sizeof(float)));
cudaCheck(cudaMemcpy(d_inp, inp, B * T * C * sizeof(float),
                     cudaMemcpyHostToDevice));
cudaCheck(cudaMemcpy(d_weight, weight, C * sizeof(float),
                     cudaMemcpyHostToDevice));
cudaCheck(cudaMemcpy(d_bias, bias, C * sizeof(float),
                     cudaMemcpyHostToDevice));

int block_sizes[] = {32, 64, 128, 256, 512, 1024};
float* out_gpu = (float*)malloc(B * T * C * sizeof(float));
float* mean_gpu = (float*)malloc(B * T * sizeof(float));
float* rstd_gpu = (float*)malloc(B * T * sizeof(float));

/// ...

free(out);
free(mean);
free(rstd);
free(inp);
free(weight);
free(bias);
cudaCheck(cudaFree(d_out));
cudaCheck(cudaFree(d_mean));
cudaCheck(cudaFree(d_rstd));
cudaCheck(cudaFree(d_inp));
cudaCheck(cudaFree(d_weight));
cudaCheck(cudaFree(d_bias));
```

alternative

```
int B = 8;
int T = 1024;
int C = 768;

thrust::host_vector<float> h_inp(B * T * C);
thrust::host_vector<float> h_weight(C);
thrust::host_vector<float> h_bias(C);

thrust::default_random_engine gen(42);
thrust::uniform_real_distribution<float> dis(-1.0f, 1.0f);
thrust::generate(h_inp.begin(), h_inp.end(), [&] { return dis(gen); });
thrust::generate(h_weight.begin(), h_weight.end(), [&] { return dis(gen); });
thrust::generate(h_bias.begin(), h_bias.end(), [&] { return dis(gen); });

thrust::device_vector<float> d_out(B * T * C);
thrust::device_vector<float> d_mean(B * T);
thrust::device_vector<float> d_rstd(B * T);
thrust::device_vector<float> d_inp(h_inp);
thrust::device_vector<float> d_weight(h_weight);
thrust::device_vector<float> d_bias(h_bias);

template <class T>
using pinned_vector = thrust::host_vector<
  T, thrust::mr::stateless_resource_allocator<
     T, thrust::system::cuda::universal_host_pinned_memory_resource>>;
```

- Type-safe
- Less error-prone
- Customizable

Algorithms

current

```
float dloss_mean = 1.0f / (B*T);

cudaCheck(cudaMemset(grads_acts.losses,
                     dloss_mean,
                     B*T * sizeof(float)));
```

- C API operates on bytes
- Bug is hard to detect

alternative

```
float dloss_mean = 1.0f / (B*T);

thrust::fill_n(thrust::device, grad_acts.losses, B * T, dloss_mean);
```

- Type-safe
- Less error-prone

Algorithms

current

```
float dloss_mean = 1.0f / (B*T);

cudaCheck(cudaMemset(grads_acts.losses,
                     dloss_mean,
                     B*T * sizeof(float)));
```

__host__cudaError_t cudaMemset (void* devPtr, int value, size_t count)

Initializes or sets device memory to a value.

Parameters

`devPtr`
- Pointer to device memory

`value`
- Value to set for each byte of specified memory

`count`
- Size in bytes to set

- C API operates on bytes
- Bug is hard to detect

alternative

```
float dloss_mean = 1.0f / (B*T);

thrust::fill_n(thrust::device, grad_acts.losses, B * T, dloss_mean);
```

- Type-safe
- Less error-prone

Algorithms

current

```
__global__ void gelu_kernel(float* out,
                            const float* inp, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N) {
    float xi = inp[i];
    float cube = 0.044715f * xi * xi * xi;
    out[i] = 0.5f * xi * (1.0f + tanhf(GELU_SCALING_FACTOR * (xi + cube)));
  }
}

void gelu_forward(float* out, const float* inp, int N) {
  const int block_size = 128;
  const int grid_size = CEIL_DIV(N, block_size);
  gelu_kernel<<<grid_size, block_size>>>(out, inp, N);
  cudaCheck(cudaGetLastError());
}
```

- Forces you to “execute” the code in your mind

alternative

```
void gelu_forward(float* out, const float* inp, int N) {
  thrust::transform(thrust::device, inp, inp + N, out,
                    [] __host__ __device__(float xi) {
    float cube = 0.044715f * xi * xi * xi;
    return 0.5f * xi * (1.0f + tanhf(GELU_SCALING_FACTOR * (xi + cube)));
  });
}
```

- Algorithm represents intent, which reduces mental load
- Potential for selecting alternative executors

Algorithms

current

```
__global__ void gelu_kernel(float* out,
                            const float* inp, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N) {
    float xi = inp[i];
    float cube = 0.044715f * xi * xi * xi;
    out[i] = 0.5f * xi * (1.0f + tanhf(GELU_SCALING_FACTOR * (xi + cube)));
  }
}

void gelu_forward(float* out, const float* inp, int N) {
  const int block_size = 128;
  const int grid_size = CEIL_DIV(N, block_size);
  gelu_kernel<<<grid_size, block_size>>>(out, inp, N);
  cudaCheck(cudaGetLastError());
}
```

- Forces you to “execute” the code in your mind

alternative

```
#define THRUST_DEVICE_SYSTEM THRUST_DEVICE_SYSTEM_OMP

void gelu_forward(float* out, const float* inp, int N) {
  thrust::transform(thrust::device, inp, inp + N, out,
                    [] __host__ __device__(float xi) {
    float cube = 0.044715f * xi * xi * xi;
    return 0.5f * xi * (1.0f + tanhf(GELU_SCALING_FACTOR * (xi + cube)));
  });
}
```

- Algorithm represents intent, which reduces mental load
- Potential for selecting alternative executors

Algorithms Customization

current

```
__global__ void residual_forward_kernel(float* out,
                                        float* inp1, float* inp2, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        out[idx] = __ldcs(&inp1[idx]) + __ldcs(&inp2[idx]);
    }
}

void residual_forward(float* out, float* inp1, float* inp2, int N) {
    const int block_size = 256;
    const int grid_size = CEIL_DIV(N, block_size);
    residual_forward_kernel<<<grid_size, block_size>>>(out, inp1, inp2, N);
    cudaCheck(cudaGetLastError());
}
```

alternative

```
void residual_forward(float* out, float* inp1, float* inp2, int N) {
    cub::CacheModifiedInputIterator<cub::LOAD_CS, float> inp1cs(inp1);
    cub::CacheModifiedInputIterator<cub::LOAD_CS, float> inp2cs(inp2);
    thrust::transform(thrust::device,
                      inp1cs, inp1cs + N, inp2cs, out, thrust::plus<float>());
}
```

- High-level abstractions do not sacrifice low-level control
- Generic API not limited to built-in types

Algorithms Customization

current

```
__global__ void residual_forward_kernel(float* out,
                                        float* inp1, float* inp2, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        out[idx] = __ldcs(&inp1[idx]) + __ldcs(&inp2[idx]);
    }
}

void residual_forward(float* out, float* inp1, float* inp2, int N) {
    const int block_size = 256;
    const int grid_size = CEIL_DIV(N, block_size);
    residual_forward_kernel<<<grid_size, block_size>>>(out, inp1, inp2, N);
    cudaCheck(cudaGetLastError());
}
```

alternative

```
void residual_forward(float* out, float* inp1, float* inp2, int N) {
    cub::CacheModifiedInputIterator<cub::LOAD_CS, float> inp1cs(inp1);
    cub::CacheModifiedInputIterator<cub::LOAD_CS, float> inp2cs(inp2);
    thrust::transform(thrust::device,
                      inp1cs, inp1cs + N, inp2cs, out, thrust::plus<float>());
}
```

```
using tex_t    = cub::CacheModifiedInputIterator<cub::LOAD_LDG, cuda::std::complex<float>>;
using stream_t = cub::CacheModifiedInputIterator<cub::LOAD_CS, cuda::std::complex<float>>;

__global__ void kernel(tex_t tex, stream_t stream) {
  cuda::std::complex<float> t = tex[threadIdx.x];    // ld.global.nc.u64
  cuda::std::complex<float> s = stream[threadIdx.x]; // ld.cs.u64
  // ...
```

- High-level abstractions do not imply abandoning low-level features
- Generic API beyond built-in types

Algorithms Customization

current

```
__global__ void residual_forward_kernel(float* out,
                                        float* inp1, float* inp2, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        out[idx] = __ldcs(&inp1[idx]) + __ldcs(&inp2[idx]);
    }
}

void residual_forward(float* out, float* inp1, float* inp2, int N) {
    const int block_size = 256;
    const int grid_size = CEIL_DIV(N, block_size);
    residual_forward_kernel<<<grid_size, block_size>>>(out, inp1, inp2, N);
    cudaCheck(cudaGetLastError());
}
```

alternative

```
void residual_forward(float* out, float* inp1, float* inp2, int N) {
    cub::CacheModifiedInputIterator<cub::LOAD_CS, float> inp1cs(inp1);
    cub::CacheModifiedInputIterator<cub::LOAD_CS, float> inp2cs(inp2);
    thrust::transform(thrust::device,
                      inp1cs, inp1cs + N, inp2cs, out, thrust::plus<float>());
}

void async_residual_forward(float* out, float* inp1, float* inp2, int N) {
    cub::CacheModifiedInputIterator<cub::LOAD_CS, float> inp1cs(inp1);
    cub::CacheModifiedInputIterator<cub::LOAD_CS, float> inp2cs(inp2);
    thrust::transform(thrust::cuda::par_nosync,
                      inp1cs, inp1cs + N, inp2cs, out, thrust::plus<float>());
}

void async_residual_forward(float* out, float* inp1, float* inp2, int N) {
    cub::CacheModifiedInputIterator<cub::LOAD_CS, float> inp1cs(inp1);
    cub::CacheModifiedInputIterator<cub::LOAD_CS, float> inp2cs(inp2);
    thrust::transform(thrust::cuda::par.on(stream),
                      inp1cs, inp1cs + N, inp2cs, out, thrust::plus<float>());
}
```

- CUDA execution policies are not limited to thrust::device

Tuple

current

```
__global__ void permute_kernel(float* q, float* k, float* v,
                               const float* inp, int B, int N, int NH, int d) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < B * NH * N * d) {
        int b = idx / (NH * N * d);
        int rest = idx % (NH * N * d);
        int nh_ = rest / (N * d);
        rest = rest % (N * d);
        int n = rest / d;
        int d_ = rest % d;
        // ...


__global__ void unpermute_kernel(float* inp, float *out, int B,
                                 int N, int NH, int d) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < B * NH * N * d) {
        int b = idx / (NH * N * d);
        int rest = idx % (NH * N * d);
        int nh_ = rest / (N * d);
        rest = rest % (N * d);
        int n = rest / d;
        int d_ = rest % d;
        // ...
```

alternative

```
__host__ __device__
cuda::std::tuple<int, int, int, int>
i2n(int idx, int E1, int E2, int E3) {
    int b = idx / (E1 * E2 * E3);
    int rest = idx % (E1 * E2 * E3);
    int nh_ = rest / (E2 * E3);
    rest = rest % (E2 * E3);
    int t = rest / E3;
    int hs = rest % E3;
    return {b, t, nh_, hs};
}

__global__ void permute_kernel(float* q, float* k, float* v,
                               const float* inp, int B, int N, int NH, int d) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < B * NH * N * d) {
        auto [b, n, nh_, d_] = i2n(idx, NH, T, HS);
        // ...


__global__ void unpermute_kernel(float* inp, float *out, int B,
                                 int N, int NH, int d) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < B * NH * N * d) {
        auto [b, n, nh_, d_] = i2n(idx, NH, T, HS);
        // ...
```

- libcu++ makes many standard types accessible in device code:
  - cuda::std::variant
  - cuda::std::tuple
  - cuda::std::pair etc.
- DRY

Fancy Iterators

current

```
__global__ void unpermute_kernel(float* inp, float *out,
                                 int B, int N, int NH, int d) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < B * NH * N * d) {
        int b = idx / (NH * N * d);
        int rest = idx % (NH * N * d);
        int nh_ = rest / (N * d);
        rest = rest % (N * d);
        int n = rest / d;
        int d_ = rest % d;

        int other_idx = (b * NH * N * d) + (n * NH * d) + (nh_ * d) + d_;
        out[other_idx] = __ldcs(&inp[idx]);
    }
}

num_blocks = CEIL_DIV(B * T * C, block_size);
unpermute_kernel<<<num_blocks, block_size>>>(vaccum, out, B, T, NH, HS);
```

alternative

```
auto map = thrust::make_transform_iterator(
  thrust::make_counting_iterator(0), [=] __host__ __device__(int idx) {
    auto [b, n, nh_, d_] = i2n(idx, NH, T, HS);
    return (b * NH * T * HS) + (n * NH * HS) + (nh_ * HS) + d_;
  });
cub::CacheModifiedInputIterator<cub::LOAD_CS, float> vaccumcs(vaccum);
thrust::scatter(thrust::device, vaccumcs, vaccumcs + B * T * C, map, out);
```

Fancy Iterators

current

```
__global__ void unpermute_kernel(float* inp, float *out,
                                 int B, int N, int NH, int d) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < B * NH * N * d) {
        int b = idx / (NH * N * d);
        int rest = idx % (NH * N * d);
        int nh_ = rest / (N * d);
        rest = rest % (N * d);
```

```
}

num_
unpe
```

alternative

```
auto map = thrust::make_transform_iterator(
  thrust::make_counting_iterator(0), [=] __host__ __device__(int idx) {
    auto [b, n, nh_, d_] = i2n(idx, NH, T, HS);
    return (b * NH * T * HS) + (n * NH * HS) + (nh_ * HS) + d_;
  });
cub::CacheModifiedInputIterator<cub::LOAD_CS, float> vaccumcs(vaccum);
thrust::scatter(thrust::device, vaccumcs, vaccumcs + B * T * C, map, out);
```

```
auto c = thrust::make_counting_iterator(10);

std::cout << c[0]   << std::endl; // 10
std::cout << c[1]   << std::endl; // 11
std::cout << c[100] << std::endl; // 110
auto c3 = c + 3;
std::cout << c3[0] << std::endl;  // 13
```

c[0]	c[1]	c[2]	c[3]
10	11	12	13

Fancy Iterators

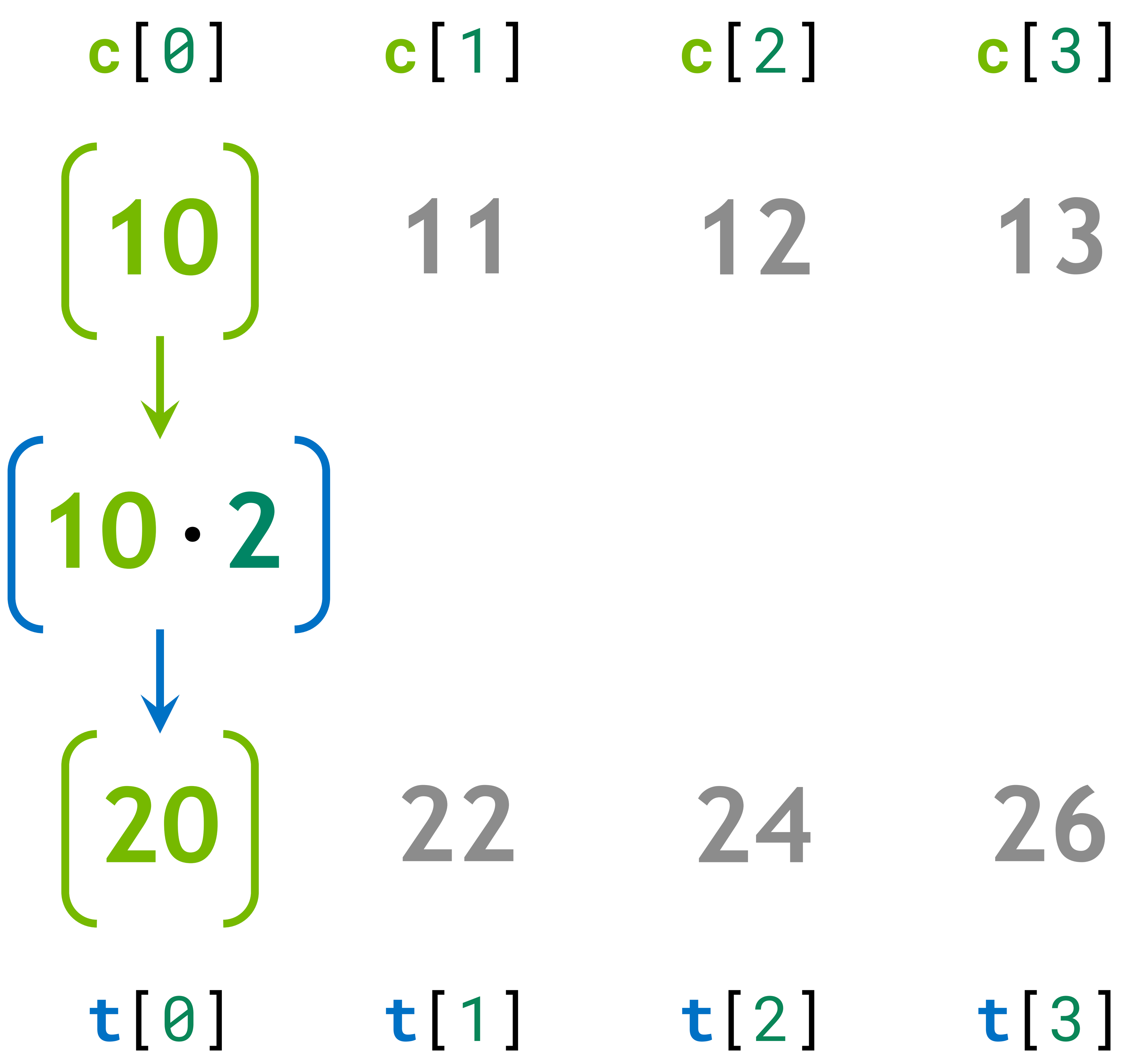
current

```
__global__ void unpermute_kernel(float* inp, float *out,
                                 int B, int N, int NH, int d) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < B * NH * N * d) {
        int b = idx / (NH * N * d);
        int rest = idx % (NH * N * d);
        int nh_ = rest / (N * d);
        rest = rest % (N * d);
}

num_
unpe
```

alternative

```
auto map = thrust::make_transform_iterator(
  thrust::make_counting_iterator(0), [=] __host__ __device__(int idx) {
    auto [b, n, nh_, d_] = i2n(idx, NH, T, HS);
    return (b * NH * T * HS) + (n * NH * HS) + (nh_ * HS) + d_;
  });
cub::CacheModifiedInputIterator<cub::LOAD_CS, float> vaccumcs(vaccum);
thrust::scatter(thrust::device, vaccumcs, vaccumcs + B * T * C, map, out);
```

```
auto c = thrust::make_counting_iterator(10);

std::cout << c[0]   << std::endl; // 10
std::cout << c[1]   << std::endl; // 11
std::cout << c[100] << std::endl; // 110
auto c3 = c + 3;
std::cout << c3[0] << std::endl;  // 13

auto t = thrust::make_transform_iterator(
           c, [] __host__ __device__(int i) {
                return i * 2;
              });

std::cout << t[0] << std::endl;   // 20
std::cout << t[1] << std::endl;   // 22
std::cout << t[100] << std::endl; // 220
auto t3 = t + 3;
std::cout << t3[0] << std::endl;  // 26
```



Fancy Iterators

current

```
__global__ void unpermute_kernel(float* inp, float *out,
                                 int B, int N, int NH, int d) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < B * NH * N * d) {
        int b = idx / (NH * N * d);
        int rest = idx % (NH * N * d);
        int nh_ = rest / (N * d);
        rest = rest % (N * d);
        int n = rest / d;
        int d_ = rest % d;

        int other_idx = (b * NH * N * d) + (n * NH * d) + (nh_ * d) + d_;
        out[other_idx] = __ldcs(&inp[idx]);
    }
}

num_blocks = CEIL_DIV(B * T * C, block_size);
unpermute_kernel<<<num_blocks, block_size>>>(vaccum, out, B, T, NH, HS);
```

alternative

```
auto map = thrust::make_transform_iterator(
  thrust::make_counting_iterator(0), [=] __host__ __device__(int idx) {
    auto [b, n, nh_, d_] = i2n(idx, NH, T, HS);
    return (b * NH * T * HS) + (n * NH * HS) + (nh_ * HS) + d_;
  });
cub::CacheModifiedInputIterator<cub::LOAD_CS, float> vaccumcs(vaccum);
thrust::scatter(thrust::device, vaccumcs, vaccumcs + B * T * C, map, out);
```

Fancy Iterators

current

```
__global__ void unpermute_kernel(float* inp, float *out,
                                 int B, int N, int NH, int d) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < B * NH * N * d) {
        int b = idx / (NH * N * d);
        int rest = idx % (NH * N * d);
        int nh_ = rest / (N * d);
        rest = rest % (N * d);
        int n = rest / d;
        int d_ = rest % d;

        int other_idx = (b * NH * N * d) + (n * NH * d) + (nh_ * d) + d_;
        out[other_idx] = __ldcs(&inp[idx]);
    }
}

num_blocks = CEIL_DIV(B * T * C, block_size);
unpermute_kernel<<<num_blocks, block_size>>>(vaccum, out, B, T, NH, HS);
```

alternative

```
auto map = thrust::make_transform_iterator(
  thrust::make_counting_iterator(0), [=] __host__ __device__(int idx) {
    auto [b, n, nh_, d_] = i2n(idx, NH, T, HS);
    return (b * NH * T * HS) + (n * NH * HS) + (nh_ * HS) + d_;
  });
cub::CacheModifiedInputIterator<cub::LOAD_CS, float> vaccumcs(vaccum);
thrust::scatter(thrust::device, vaccumcs, vaccumcs + B * T * C, map, out);
```

MDSpan

current

```
__global__ void permute_kernel(float* q, float* k, float* v,
                               const float* inp, int B, int N, int NH, int d) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Q[b][nh_][n][d_] = inp[b][n][0][nh_][d_]

    if (idx < B * NH * N * d) {
        int b = idx / (NH * N * d);
        int rest = idx % (NH * N * d);
        int nh_ = rest / (N * d);
        rest = rest % (N * d);
        int n = rest / d;
        int d_ = rest % d;

        int inp_idx = \
            (b * N * 3 * NH * d)
            +   (n * 3 * NH * d)
            +       (0 * NH * d)
            +          (nh_ * d)
            +                d_;

        q[idx] = __ldcs(&inp[inp_idx]);
        k[idx] = __ldcs(&inp[inp_idx + NH * d]);
        v[idx] = __ldcs(&inp[inp_idx + 2 * (NH * d)]);
    }
}

void attention_forward(float* out, float* vaccum, float* qkvr, float* preatt, float* att,
                       float* inp, int B, int T, int C, int NH) {
    const int block_size = 256;
    const int softmax_block_size = 256;

    int HS = C / NH; // head size

    float *q, *k, *v;
    q = qkvr + 0 * B * T * C;
    k = qkvr + 1 * B * T * C;
    v = qkvr + 2 * B * T * C;
    int total_threads = B * NH * T * HS;
    int num_blocks = CEIL_DIV(total_threads, block_size);
    permute_kernel<<<num_blocks, block_size>>>(q, k, v, inp, B, T, NH, HS);
```

alternative

```
void attention_forward(float* out, float* vaccum, float* qkvr, float* preatt, float* att,
                       float* inp, int B, int T, int C, int NH) {
    const int block_size = 256;
    const int softmax_block_size = 256;

    int HS = C / NH; // head size

    float *q, *k, *v;
    q = qkvr + 0 * B * T * C;
    k = qkvr + 1 * B * T * C;
    v = qkvr + 2 * B * T * C;

    constexpr auto dyn = cuda::std::dynamic_extent;
    using ext_t = cuda::std::extents<int, dyn, dyn, 3, dyn, dyn>;
    using mds_t = cuda::std::mdspan<const float, ext_t>;

    ext_t extents{B, T, NH, HS};
    mds_t inp_md{inp, extents};

    auto begin = thrust::make_counting_iterator(0);
    auto end = begin + B * NH * T * HS;

    // Q[b][nh_][n][d_] = inp[b][n][0][nh_][d_]
    thrust::for_each(thrust::cuda::par_nosync, begin, end,
                     [=] __device__(int idx) {
                        auto [b, t, nh_, hs] = i2n(idx, NH, T, HS);
                        q[idx] = inp_md(b, t, 0, nh_, hs);
                        k[idx] = inp_md(b, t, 1, nh_, hs);
                        v[idx] = inp_md(b, t, 2, nh_, hs);
                     });
```

- Turn `// Q[b][nh_][n][d_] = inp[b][n][0][nh_][d_]` into actual code
- Preserve compile-time information

MDSpan

current

```
__global__ void permute_kernel(float* q, float* k, float* v,
                               const float* inp, int B, int N, int NH, int d) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Q[b][nh_][n][d_] = inp[b][n][0][nh_][d_]

    if (idx < B * NH * N * d) {
        int b = idx / (NH * N * d);
        int rest = idx % (NH * N * d);
        int nh_ = rest / (N * d);
        rest = rest % (N * d);
        int n = rest / d;
        int d_ = rest % d;

        int inp_idx = \
            (b * N * 3 * NH * d)
            +   (n * 3 * NH * d)
            +       (0 * NH * d)
            +          (nh_ * d)
            +                d_;

        q[idx] = __ldcs(&inp[inp_idx]);
        k[idx] = __ldcs(&inp[inp_idx + NH * d]);
        v[idx] = __ldcs(&inp[inp_idx + 2 * (NH * d)])
    }
}

void attention_forward(float* out, float* vaccum, flo
                       float* inp, int B, int T, int
    const int block_size = 256;
    const int softmax_block_size = 256;

    int HS = C / NH; // head size

    // permute and separate inp from (B, T, 3, NH, HS) to 3X (B, NH, T, HS)
    float *q, *k, *v;
    q = qkvr + 0 * B * T * C;
    k = qkvr + 1 * B * T * C;
    v = qkvr + 2 * B * T * C;
    int total_threads = B * NH * T * HS;
    int num_blocks = CEIL_DIV(total_threads, block_size);
    permute_kernel<<<num_blocks, block_size>>>(q, k, v, inp, B, T, NH, HS);
```

alternative

```
void attention_forward(float* out, float* vaccum, float* qkvr, float* preatt, float* att,
                       float* inp, int B, int T, int C, int NH) {
    const int block_size = 256;
    const int softmax_block_size = 256;

    int HS = C / NH; // head size

    float *q, *k, *v;
    q = qkvr + 0 * B * T * C;
    k = qkvr + 1 * B * T * C;
    v = qkvr + 2 * B * T * C;
dyn>;
iterator(0),
* T * HS),
T, HS);
});
```

```
cuda::std::array<int, 9> linearized = {
  0, 1, 2,
  3, 4, 5,
  6, 7, 8
};

cuda::std::mdspan<int, cuda::std::extents<int, 3, 3>> md(linearized.data());

std::cout << md(0, 0) << std::endl; // 0
std::cout << md(1, 1) << std::endl; // 4
std::cout << md(1, 2) << std::endl; // 5
```

- Turn `// Q[b][nh_][n][d_] = inp[b][n][0][nh_][d_]` into actual code
- Preserve compile-time information

MDSpan

current

```
__global__ void permute_kernel(float* q, float* k, float* v,
                               const float* inp, int B, int N, int NH, int d) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Q[b][nh_][n][d_] = inp[b][n][0][nh_][d_]

    if (idx < B * NH * N * d) {
        int b = idx / (NH * N * d);
        int rest = idx % (NH * N * d);
        int nh_ = rest / (N * d);
        rest = rest % (N * d);
        int n = rest / d;
        int d_ = rest % d;

        int inp_idx = \
            (b * N * 3 * NH * d)
            +   (n * 3 * NH * d)
            +       (0 * NH * d)
            +          (nh_ * d)
            +                d_;

        q[idx] = __ldcs(&inp[inp_idx]);
        k[idx] = __ldcs(&inp[inp_idx + NH * d]);
        v[idx] = __ldcs(&inp[inp_idx + 2 * (NH * d)]);
    }
}

void attention_forward(float* out, float* vaccum, float* qkvr, float* preatt, float* att,
                       float* inp, int B, int T, int C, int NH) {
    const int block_size = 256;
    const int softmax_block_size = 256;

    int HS = C / NH; // head size

    // permute and separate inp from (B, T, 3, NH, HS) to 3X (B, NH, T, HS)
    float *q, *k, *v;
    q = qkvr + 0 * B * T * C;
    k = qkvr + 1 * B * T * C;
    v = qkvr + 2 * B * T * C;
    int total_threads = B * NH * T * HS;
    int num_blocks = CEIL_DIV(total_threads, block_size);
    permute_kernel<<<num_blocks, block_size>>>(q, k, v, inp, B, T, NH, HS);
```

alternative

```
void attention_forward(float* out, float* vaccum, float* qkvr, float* preatt, float* att,
                       float* inp, int B, int T, int C, int NH) {
    const int block_size = 256;
    const int softmax_block_size = 256;

    int HS = C / NH; // head size

    float *q, *k, *v;
    q = qkvr + 0 * B * T * C;
    k = qkvr + 1 * B * T * C;
    v = qkvr + 2 * B * T * C;

    constexpr auto dyn = cuda::std::dynamic_extent;
    using ext_t = cuda::std::extents<int, dyn, dyn, 3, dyn, dyn>;
    using mds_t = cuda::std::mdspan<const float, ext_t>;

    ext_t extents{B, T, NH, HS};
    mds_t inp_md{inp, extents};

    // Q[b][nh_][n][d_] = inp[b][n][0][nh_][d_]
    thrust::for_each(thrust::device, thrust::make_counting_iterator(0),
                     thrust::make_counting_iterator(B * NH * T * HS),
                     [=] __device__(int idx) {
                        auto [b, t, nh_, hs] = i2n(idx, NH, T, HS);
                        q[idx] = inp_md(b, t, 0, nh_, hs);
                        k[idx] = inp_md(b, t, 1, nh_, hs);
                        v[idx] = inp_md(b, t, 2, nh_, hs);
                     });
```

- Turn `// Q[b][nh_][n][d_] = inp[b][n][0][nh_][d_]` into actual code
- Preserve compile-time information

MDSpan Customization

```
template <class T> struct streaming_accessor {
    using offset_policy = streaming_accessor;
    using element_type = T;
    using data_handle_type = const T *;
    using reference = const T;

    inline __host__ __device__
    reference access(data_handle_type p, size_t i) const {
        NV_IF_TARGET(NV_IS_DEVICE,
                     (return __ldcs(p + i);),
                     (return p[i];));
    }

    inline __host__ __device__
    data_handle_type offset(data_handle_type p, size_t i) const {
        return p + i;
    }
};
```

```
void attention_forward(float* out, float* vaccum, float* qkvr, float* preatt,
                       float* att, float* inp, int B, int T, int C, int NH) {
    const int block_size = 256;
    const int softmax_block_size = 256;

    int HS = C / NH; // head size

    float *q, *k, *v;
    q = qkvr + 0 * B * T * C;
    k = qkvr + 1 * B * T * C;
    v = qkvr + 2 * B * T * C;

    constexpr auto dyn = cuda::std::dynamic_extent;
    using ext_t = cuda::std::extents<int, dyn, dyn, 3, dyn, dyn>;
    using mds_t = cuda::std::mdspan<const float, ext_t>;
    using mds_t = cuda::std::mdspan<const float, ext_t, cuda::std::layout_right,
                                    streaming_accessor<float>>;
```

- High-level abstractions do not imply abandoning low-level features

MDSpan

Domain-specific types

current

```
__global__ void encoder_forward_kernel2(float* out,
                                        int* inp, float* wte, float* wpe,
                                        int B, int T, int C) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int N = B * T * C;

    if (idx < N) {
        int bt = idx / C;
        int b = bt / T;
        int t = bt % T;
        int c = idx % C;

        int ix = inp[b * T + t];

        float* out_btc = out + b * T * C + t * C + c;
        float* wte_ix = wte + ix * C + c;
        float* wpe_tc = wpe + t * C + c;
        *out_btc = *wte_ix + *wpe_tc;
    }
}

void encoder_forward(float* out,
                     int* inp, float* wte, float* wpe,
                     int B, int T, int C) {
    const int N = B * T * C;
    const int block_size = 256;
    const int grid_size = CEIL_DIV(N, block_size);
    encoder_forward_kernel2<<<grid_size, block_size>>>(
      out, inp, wte, wpe, B, T, C);
    cudaCheck(cudaGetLastError());
}
```

alternative

```
using float_3d_mds = cuda::std::mdspan<float, cuda::std::dextents<int, 3>>;
using float_2d_mds = cuda::std::mdspan<float, cuda::std::dextents<int, 2>>;
using const_int_2d_mds = cuda::std::mdspan<const int, cuda::std::dextents<int, 2>>;

using output_tensor = float_3d_mds;
using weight_embed_tensor = float_2d_mds;
using position_embed_tensor = float_2d_mds;
using input_matrix = const_int_2d_mds;

void encoder_forward(float* out, const thrust::device_vector<int>& inpv,
                     float* wte, float* wpe,
                     int B, int T, int C, int V) {
  output_tensor out_md(out, B, T, C);
  weight_embed_tensor wte_md(wte, V, C);
  position_embed_tensor wpe_md(wpe, T, C);
  input_matrix inp_md(thrust::raw_pointer_cast(inpv.data()), B, T);

  cudaCheck(cub::DeviceFor::Bulk(B * T * C, [=] __device__(int idx) {
    auto [b, t, c] = i2n(idx, C, T);
    out_md(b, t, c) = wte_md(inp_md(b, t), c) + wpe_md(t, c);
  }));
}
```

Kernel Fusion

current

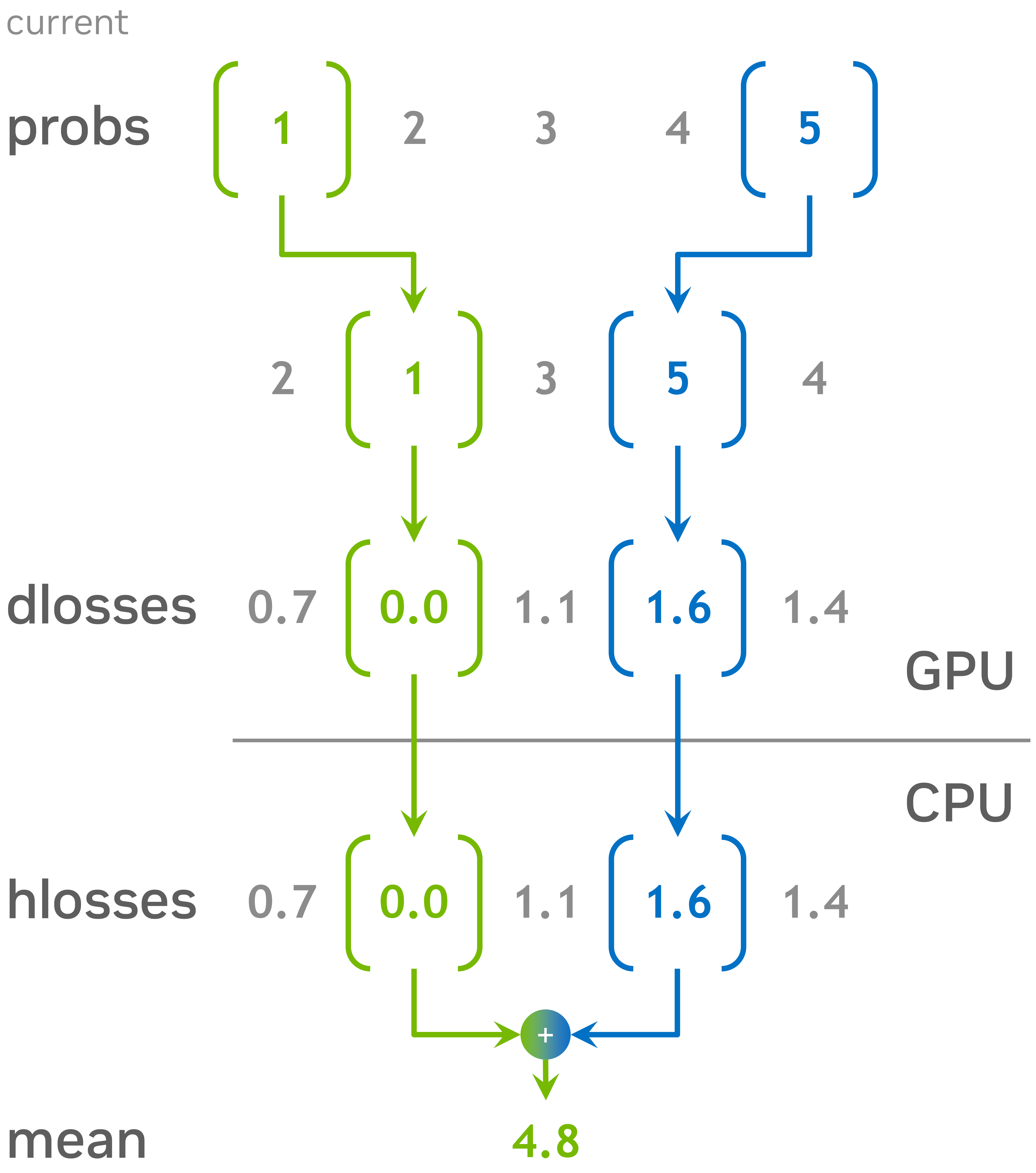
```
__global__ void crossentropy_forward_kernel1(float* losses,
                            float* probs, int* targets,
                            int B, int T, int V) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < B * T) {
        int b = i / T;
        int t = i % T;
        float* probs_bt = probs + b * T * V + t * V;
        int ix = targets[b * T + t];
        losses[b * T + t] = -logf(probs_bt[ix]);
    }
}


void crossentropy_forward(float* losses,
                            float* probs, int* targets,
                            int B, int T, int V) {
    const int block_size = 128;
    const int N = B * T;
    const int grid_size = CEIL_DIV(N, block_size);
    crossentropy_forward_kernel1<<<grid_size, block_size>>>(
      losses, probs, targets, B, T, V);
    cudaCheck(cudaGetLastError());
}

crossentropy_forward(acts.losses, acts.probs, model->targets, B, T, V);
cudaCheck(cudaMemcpy(model->cpu_losses, acts.losses, B * T * sizeof(float),
cudaMemcpyDeviceToHost));
float mean_loss = 0.0f;
for (int i=0; i<B*T; i++) { mean_loss += model->cpu_losses[i]; }
mean_loss /= B*T;
model->mean_loss = mean_loss;
```

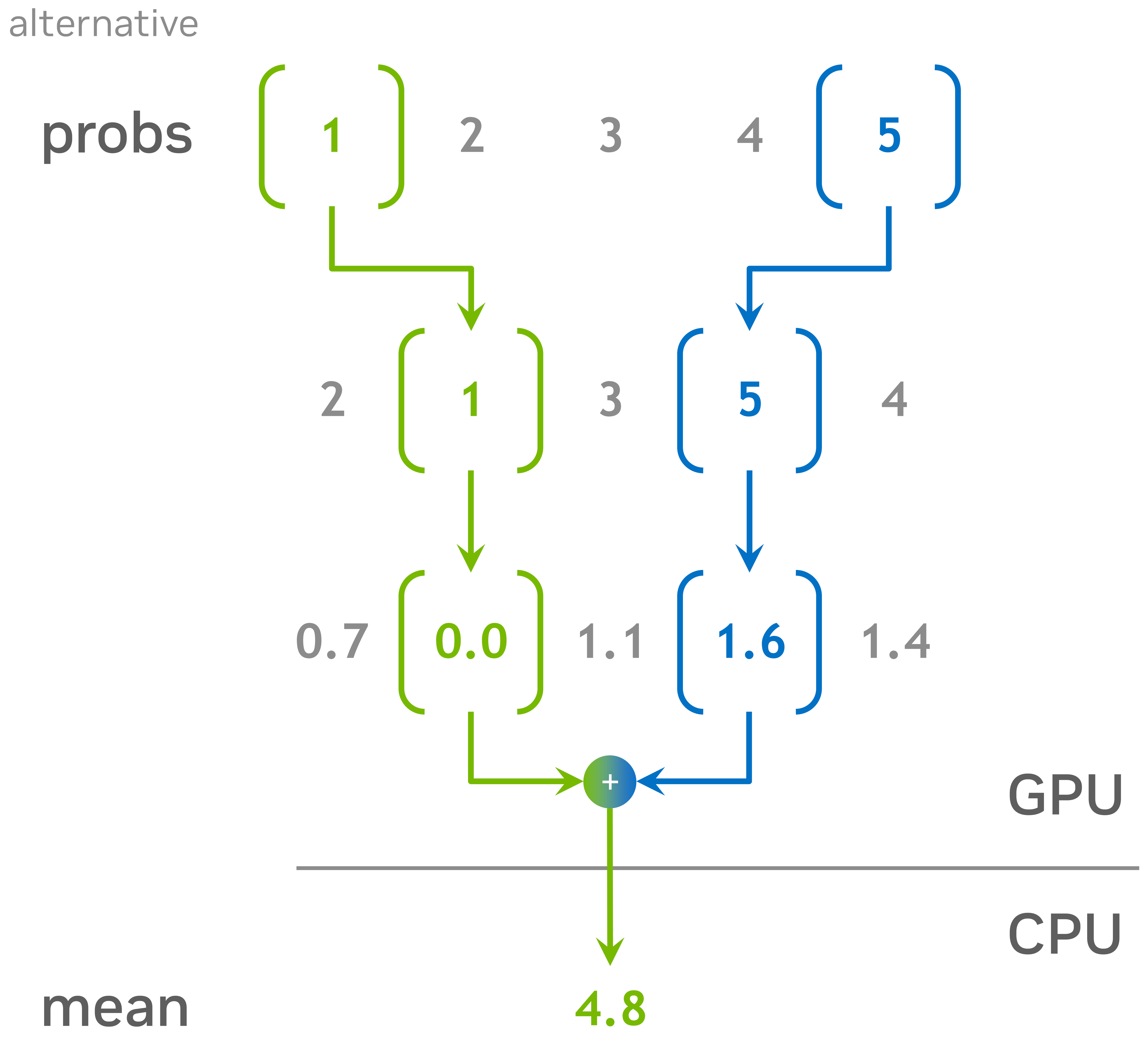
alternative

```
target_matrix targets_md(thrust::raw_pointer_cast(model.targets.data()), B, T);

auto map = thrust::make_transform_iterator(
             thrust::make_counting_iterator(0),
                [=] __device__(int i) -> int {
                  int b = i / T;
                  int t = i % T;
                  return targets_md(b, t) + b * T * V + t * V;
                });
auto permutation = thrust::make_permutation_iterator(acts.probs, map);
auto losses = thrust::make_transform_iterator(
    permutation, [] __device__(float prob) -> float { return -logf(prob); });

model.mean_loss = thrust::reduce(thrust::device, losses, losses + B * T, 0.0) / (B * T);
```

- Only four bytes pass PCIe
- Permutation is left for illustration purposes

Kernel Fusion





- B * T times fewer bytes crossing PCIe
- Permutation is left for illustration purposes

atomic

current

```
atomicAdd(dwte_ix, *dout_btc);
atomicAdd(dwpe_tc, *dout_btc);
```

Without looking into docs, it's hard to tell:

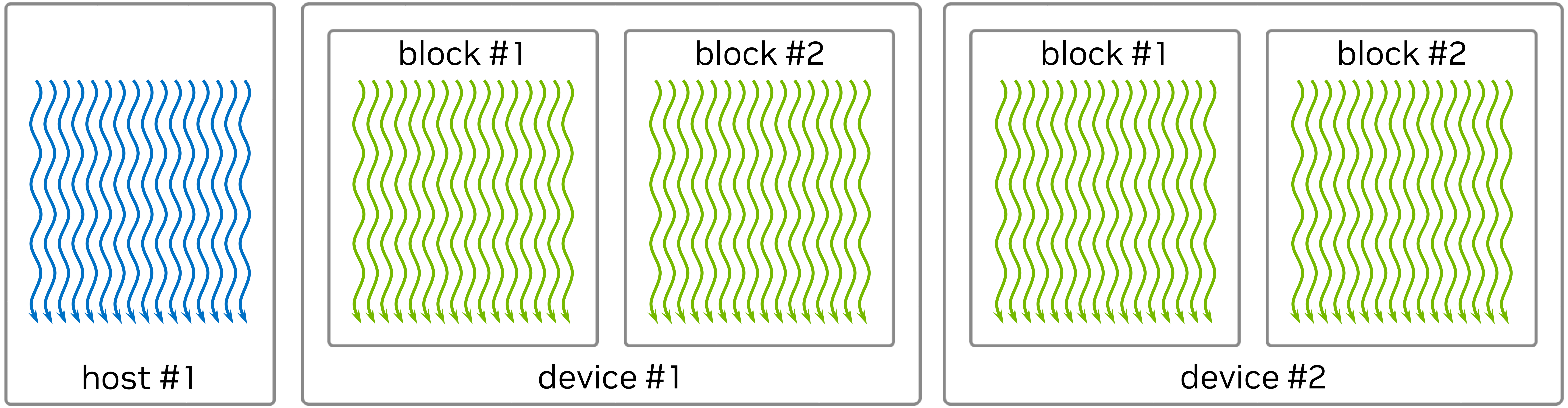
- what's the thread scope
- what's the memory order

alternative

```
cuda::atomic_ref<float, cuda::thread_scope_device> dwte_ix_ref(*dwte_ix);
cuda::atomic_ref<float, cuda::thread_scope_device> dwte_tc_ref(*dwpe_tc);

dwte_ix_ref.fetch_add(*dout_btc, cuda::memory_order_relaxed);
dwte_tc_ref.fetch_add(*dout_btc, cuda::memory_order_relaxed);
```

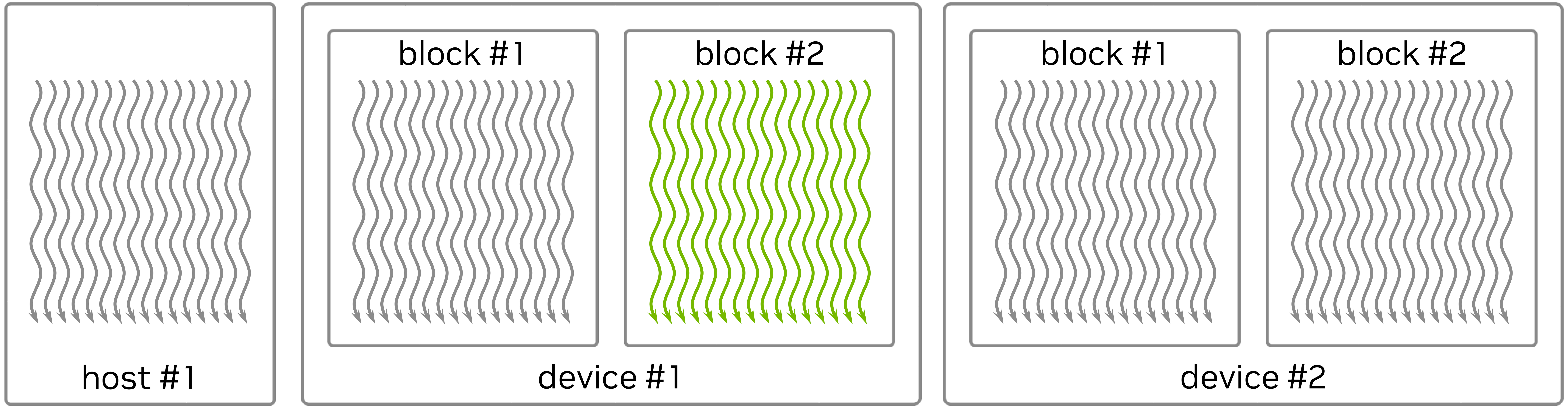
- Generic API not limited to built-in types

Thread Scope



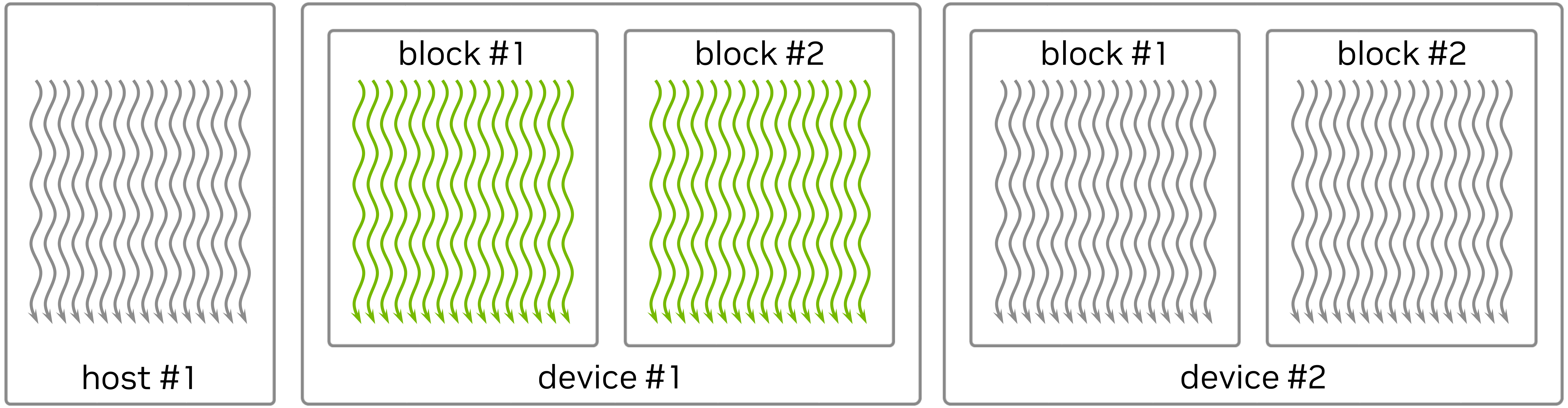
Scope is a set of threads that may interact directly with given operation and establish relations described in the memory consistency model

Thread Scope



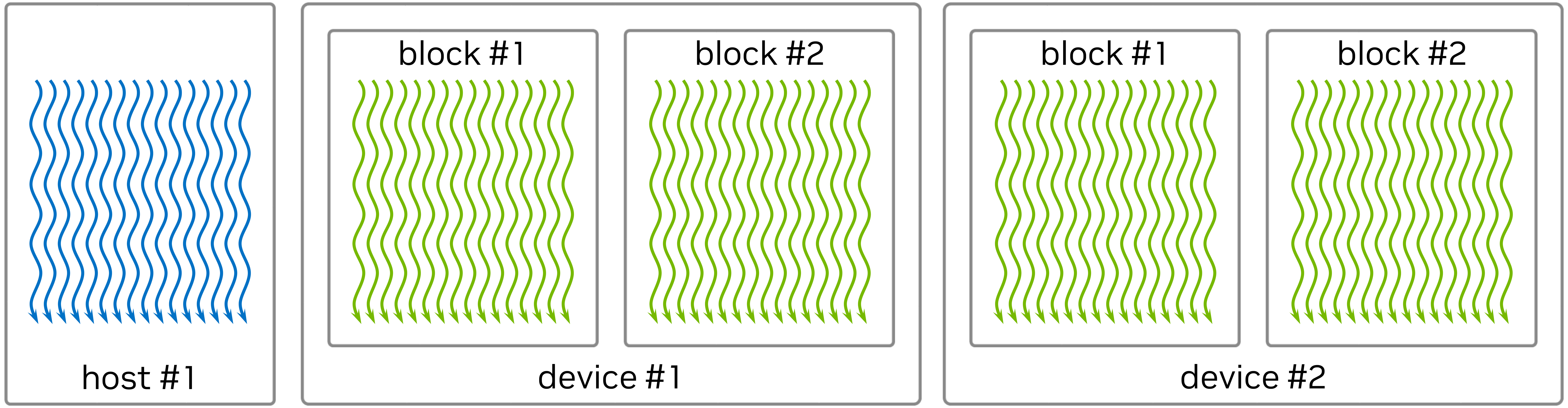
cuda::thread_scope_block is a set of threads of a given thread block

Thread Scope



cuda::thread_scope_device is a set of threads of a given device

Thread Scope



cuda::thread_scope_system is a set of threads of a given system

CUB

Block-level algorithms

current

```
__global__
void layernorm_forward_kernel3(float* out, float* mean, float* rstd,
                               const float* inp, const float* weight,
                               const float* bias, int N, int C) {
    cg::thread_block block = cg::this_thread_block();
    cg::thread_block_tile<32> warp = cg::tiled_partition<32>(block);
    int idx = blockIdx.x * warp.meta_group_size() + warp.meta_group_rank();

    const float* x = inp + idx * C;

    // mean
    float sum = 0.0f;
    for (int i = warp.thread_rank(); i < C; i += warp.size()) {
        sum += x[i];
    }

    sum = cg::reduce(warp, sum, cg::plus<float>{});

    float m = sum / C;
    if(warp.thread_rank() == 0 && mean != nullptr) {
        __stcs(mean + idx, m);
    }
```

alternative

```
constexpr int block_size = 64;
__global__ __launch_bounds__(block_size)
void layernorm_forward_kernel3(float* out, float* mean, float* rstd,
                               const float* inp, const float* weight,
                               const float* bias, int N, int C) {
    int tid = threadIdx.x;
    int idx = blockIdx.x;

    const float* x = inp + idx * C;

    // mean
    float sum = 0.0;
    for (int i = tid; i < C; i += block_size) {
        sum += x[i];
    }


    sum = cub::BlockReduce<float, block_size>().Sum(sum);

    __shared__ float shared_mean;
    if(tid == 0 && mean != nullptr) {
        float m = sum / C;
        shared_mean = m;
        __stcs(mean + idx, m);
    }
    __syncthreads();
    const float m = shared_mean;
```

- Block-level algorithm reduces items-per-thread ratio

NVBench

```
void kernel3(nvbench::state &state) {
  int B = 8;
  int T = 1024;
  int C = 768;

  thrust::host_vector<float> h_inp(B * T * C);
  thrust::host_vector<float> h_weight(C);
  thrust::host_vector<float> h_bias(C);

  thrust::default_random_engine gen(42);
  thrust::uniform_real_distribution<float> dis(-1.0f, 1.0f);
  thrust::generate(h_inp.begin(), h_inp.end(), [&] { return dis(gen); });
  thrust::generate(h_weight.begin(), h_weight.end(), [&] { return dis(gen); });
  thrust::generate(h_bias.begin(), h_bias.end(), [&] { return dis(gen); });

  thrust::device_vector<float> d_out(B * T * C);
  thrust::device_vector<float> d_mean(B * T);
  thrust::device_vector<float> d_rstd(B * T);
  thrust::device_vector<float> d_inp(h_inp);
  thrust::device_vector<float> d_weight(h_weight);
  thrust::device_vector<float> d_bias(h_bias);

  const int N = B * T;
  const int block_size = state.get_int64("block_size");
  const int grid_size = (N * 32 + block_size - 1) / block_size;

  state.add_global_memory_reads<float>(d_inp.size() + d_weight.size() + d_bias.size());
  state.add_global_memory_writes<float>(d_out.size() + d_mean.size() + d_rstd.size());

  state.exec([&](nvbench::launch &launch) {
    cudaStream_t stream = launch.get_stream();
    layernorm_forward_kernel3<<<grid_size, block_size, 0, stream>>>(
      thrust::raw_pointer_cast(d_out.data()),
      thrust::raw_pointer_cast(d_mean.data()),
      thrust::raw_pointer_cast(d_rstd.data()),
      thrust::raw_pointer_cast(d_inp.data()),
      thrust::raw_pointer_cast(d_weight.data()),
      thrust::raw_pointer_cast(d_bias.data()),
  });
}
NVBENCH_BENCH(kernel3).add_int64_axis("block_size", {32, 64, 128, 256, 512, 1024});
```

NVBench

```
void kernel3(nvbench::state &state) {
  state.add_global_memory_reads<float>(d_inp.size() + d_weight.size() + d_bias.size());
  state.add_global_memory_writes<float>(d_out.size() + d_mean.size() + d_rstd.size());

  state.exec([&](nvbench::launch &launch) {
    cudaStream_t stream = launch.get_stream();
    layernorm_forward_kernel3<<<grid_size, block_size, 0, stream>>>(/* ... */);
  });
}
NVBENCH_BENCH(kernel3).add_int64_axis("block_size", {32, 64, 128, 256, 512, 1024});
```

NVBench

Registering benchmark

```
void kernel3(nvbench::state &state) {
  state.add_global_memory_reads<float>(d_inp.size() + d_weight.size() + d_bias.size());
  state.add_global_memory_writes<float>(d_out.size() + d_mean.size() + d_rstd.size());

  state.exec([&](nvbench::launch &launch) {
    cudaStream_t stream = launch.get_stream();
    layernorm_forward_kernel3<<<grid_size, block_size, 0, stream>>>(/* ... */);
  });
}
NVBENCH_BENCH(kernel3).add_int64_axis("block_size", {32, 64, 128, 256, 512, 1024});
```

NVBench

Reporting bandwidth

```
void kernel3(nvbench::state &state) {
  state.add_global_memory_reads<float>(d_inp.size() + d_weight.size() + d_bias.size());
  state.add_global_memory_writes<float>(d_out.size() + d_mean.size() + d_rstd.size());

  state.exec([&](nvbench::launch &launch) {
    cudaStream_t stream = launch.get_stream();
    layernorm_forward_kernel3<<<grid_size, block_size, 0, stream>>>(/* ... */);
  });
}
NVBENCH_BENCH(kernel3).add_int64_axis("block_size", {32, 64, 128, 256, 512, 1024});
```

NVBench

Executing benchmark

```
void kernel3(nvbench::state &state) {
  state.add_global_memory_reads<float>(d_inp.size() + d_weight.size() + d_bias.size());
  state.add_global_memory_writes<float>(d_out.size() + d_mean.size() + d_rstd.size());

  state.exec([&](nvbench::launch &launch) {
    cudaStream_t stream = launch.get_stream();
    layernorm_forward_kernel3<<<grid_size, block_size, 0, stream>>>(/* ... */);
  });
}
NVBENCH_BENCH(kernel3).add_int64_axis("block_size", {32, 64, 128, 256, 512, 1024});
```

block_size	Samples	CPU Time	Noise	GPU Time	Noise	GlobalMem BW	BWUtil	Samples	Batch GPU
32	320x	103.262 us	45.20%	74.645 us	3.58%	675.240 GB/s	70.33%	22346x	26.997 us
64	328x	106.501 us	37.29%	79.605 us	2.79%	633.169 GB/s	65.95%	19768x	30.325 us
128	466x	105.126 us	40.63%	78.102 us	3.08%	645.352 GB/s	67.22%	16580x	30.927 us
256	462x	103.782 us	38.78%	77.250 us	3.20%	652.469 GB/s	67.96%	19419x	25.748 us
512	320x	104.687 us	44.59%	76.679 us	3.10%	657.328 GB/s	68.46%	22346x	28.668 us
1024	318x	99.537 us	50.54%	70.749 us	4.02%	712.424 GB/s	74.20%	21415x	25.835 us

Takeaways

Don't use raw allocations

- Explicit `cudaMalloc/cudaFree` calls are tedious and error-prone
- Use containers like `thrust::device_vector`

When authoring kernels

- Use CUB block/warp algorithms for speed-of-light building blocks
- Use `cuda::atomic_ref`, not `atomicAdd`
- Use familiar types like `cuda::std::array`, `cuda::std::variant`, `cuda::std::tuple`, `cuda::std::optional`

Before writing custom kernels

- Consider using Thrust/CUB algorithms
  - Use Thrust for higher-level abstraction and CPU/GPU support
  - Use CUB for lower-level, CUDA-specific control
- Use fancy iterators to expand the power of algorithms

General Advice

- Use CMake for a convenient and robust CUDA C++ build system
- Use NVBench for statistically sound CUDA benchmarking
- Use `cuda::std::span` instead of raw pointers
- Use `cuda::std::mdspan` for multi-dimensional data

Call To Action

TL;DR: Anyone writing CUDA C++ should be using CCCL

- Think “Can I solve this with CCCL?” first
- If we are missing something, let us know!

We are Open Source! Come ask questions, collaborate, and contribute!

GitHub

github.com/NVIDIA/cccl

Discord

discord.gg/nvidiadeveloper

#cuda-cpp-core-libraries

NVIDIA