Bringing Existing Code to CUDA Using constexpr and std::pmr

BOWIE OWENS

Cppcon
The C++ Conference
2021
October 24-29

Outline

- Introduction
- Memory
- Host vs Device Functions
- Return on Investment
- Concluding remarks

Introduction

- I work the RiskLab team at CSIRO on applied mathematics for Financial Risk.
- The aim of this talk is to:
  - Document some of the challenges in applying the principles from introductory CUDA examples to an existing project that has a meaningful amount of non-trivial code.
  - Provide some guidance to people about to embark on using CUDA to speed up existing software.

An Even Easier Introduction to CUDA

```
void add_cpu(int n, float* x, float* y) {
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}
TEST_CASE("cppcon-0", "[CUDA]") {
    int N = 1 << 20;
    float* x = new float[N];
    float* y = new float[N];

    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    add_cpu(N, x, y);

    delete[] x;
    delete[] y;
}
```

An Even Easier Introduction to CUDA

```
TEST_CASE("cppcon-1", "[CUDA]") {
    int N = 1 << 20;
    float* x;
    float* y;
    cudaMallocManaged(&x, N*sizeof(float));
    cudaMallocManaged(&y, N*sizeof(float));

    // …

    cudaFree(x);
    cudaFree(y);
}
```

An Even Easier Introduction to CUDA

```
__global__ void add_gpu(int n, float* x, float* y)
{
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}
TEST_CASE("cppcon-1", "[CUDA]") {
    // …
}
```

An Even Easier Introduction to CUDA

```
__global__ void add_gpu(int n, float* x, float* y) {
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}
TEST_CASE("cppcon-1", "[CUDA]") {
    int N = 1 << 20;
    float* x;
    float* y;
    // …

    add_gpu<<<1, 1>>>(N, x, y);

    // …
}
```

Questions About the Introductory Example?

```
__global__ void add_gpu(int n, float* x, float* y) {
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}
TEST_CASE("cppcon-1", "[CUDA]") {
    int N = 1 << 20;
    float* x;
    float* y;
    cudaMallocManaged(&x, N*sizeof(float));
    cudaMallocManaged(&y, N*sizeof(float));

    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    add_gpu<<<1, 1>>>(N, x, y);

    cudaFree(x);
    cudaFree(y);
}
```

Ok, about the kernel parameters

```
TEST_CASE("cppcon-1", "[CUDA]") {
    int N = 1 << 20;
    float* x;
    float* y;
    // …

    int block_size = 256;
    int grid_size  = (N + block_size - 1) / block_size;
    add_gpu<<<grid_size, block_size>>>(N, x, y);

    // …
}
```

Ok, about the kernel parameters

```
__global__ void add_gpu(int n, float* x, float* y) {
    int i0 = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = i0; i < n; i += stride)
        y[i] = x[i] + y[i];
}
TEST_CASE("cppcon-1", "[CUDA]") {
    // …
    add_gpu<<<grid_size, block_size>>>(N, x, y);
  // …
}
```

Memory

CPU vs GPU Memory

"In a typical PC or cluster node today, **the memories of the CPU and GPU are physically distinct and separated by the PCI-Express bus.**" -- https://developer.nvidia.com/blog/unified-memory-in-cuda-6/

System Memory

GPU Memory

Unified Memory

“Unified Memory creates a pool of managed memory that is shared between the CPU and GPU, bridging the CPU-GPU divide. Managed memory is accessible to both the CPU and GPU using a single pointer. The key is that the system automatically *migrates* data allocated in Unified Memory between host and device…” -- https://developer.nvidia.com/blog/unified-memory-in-cuda-6/

System Memory GPU Memory

Memory Allocation

```
// cpu
float* x = new float[N];
float* y = new float[N];




// …

delete[] x;
delete[] y;
```

```
// gpu
float* x;
float* y;

cudaMallocManaged(
  &x, N*sizeof(float));
cudaMallocManaged(
  &y, N*sizeof(float));

// …

cudaFree(x);
cudaFree(y);
```

Memory Allocation

```
// cpu                        // gpu
std::vector<float> x(N);      // ???
std::vector<float> y(N);      // ???

// ...                        // ...
```

std::pmr

- Added in C++17:
  - `std::pmr::memory_resource`
  - `std::pmr::polymorphic_allocator`
  - `std::pmr::vector`
  - `std::pmr::monotonic_buffer_resource`
  - …

Memory Allocation

```
// gpu
unified_memory_resource mem;

std::pmr::vector<float> x(N, &mem);
std::pmr::vector<float> y(N, &mem);

// ...
```

A Unified Memory Resource

```
struct unified_memory_resource :
  std::pmr::memory_resource {
    void* do_allocate(std::size_t, std::size_t);

    void do_deallocate(
        void* p, std::size_t, std::size_t);

    bool do_is_equal(
      const std::pmr::memory_resource& other)
        const noexcept;
};
```

A Unified Memory Resource

```
struct unified_memory_resource :
  std::pmr::memory_resource {
    void* do_allocate(std::size_t, std::size_t) final;

    void do_deallocate(
        void* p, std::size_t, std::size_t) final;

    bool do_is_equal(
      const std::pmr::memory_resource& other)
        const noexcept final;
};
```

A Unified Memory Resource

```
void* unified_memory_resource::do_allocate(
    std::size_t bytes, std::size_t /*alignment*/) {
    void* x      = nullptr;
    auto const r = cudaMallocManaged(&x, bytes);
    if (r != cudaError_t::cudaSuccess) {
        throw std::bad_alloc();
    }

    return x;
}

void unified_memory_resource::do_deallocate(
    void* p, std::size_t, std::size_t) {
    cudaFree(p);
}
```

Memory Allocation

```
// gpu
unified_memory_resource mem;

std::pmr::vector<float> x(N, &mem);
std::pmr::vector<float> y(N, &mem);

// ...
```

Avoid References to Objects Not in Unified Memory

```
void add_cpu(
  int n,
  vector<float> const& x,
  vector<float>& y) {
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}
```

```
__global__
void add_gpu(
  int n,
  gsl::span<float const> x,
  gsl::span<float> y) {
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}
```

Avoid References to Objects Not in Unified Memory

```
template <class callable>
void add(int n, callable f, gsl::span<float> y)
{  for (int i = 0; i < n; i++) y[i] += f(i); }
```

```
struct add_ref {
   std::vector<double> const& a_;
   std::vector<double> const& b_;

   auto operator()(index i) const
   { return a_[i] + b_[i]; }
};
```

```
struct add_span {
   gsl::span<double const> a_;
   gsl::span<double const> b_;

   auto operator()(index i) const
   { return a_[i] + b_[i]; }
};
```

Unified Memory

"Unified Memory creates a pool of managed memory that is shared between the CPU and GPU, bridging the CPU-GPU divide. Managed memory is accessible to both the CPU and GPU using a single pointer. The key is that **the system automatically *migrates* data** allocated in Unified Memory between host and device…." -- https://developer.nvidia.com/blog/unified-memory-in-cuda-6/

System Memory

GPU Memory

A Device Only Memory Resource

```
struct device_memory_resource :
    std::pmr::memory_resource {
    void* do_allocate(
        std::size_t bytes, std::size_t);

    void do_deallocate(
        void* p, std::size_t, std::size_t);

    bool
    do_is_equal(const std::pmr::memory_resource&)
       const noexcept;
};
```

A Device Only Memory Resource

```
void* device_memory_resource::do_allocate(
    std::size_t bytes, std::size_t) {
    void* x       = nullptr;
    auto const r = cudaMalloc(&x, bytes);
    if (r != cudaError_t::cudaSuccess) {
        throw std::bad_alloc();
    }

    return x;
}
void device_memory_resource::do_deallocate(
    void* p, std::size_t, std::size_t) {
    cudaFree(p);
}
```

Device Only Memory Allocation

```
device_memory_resource mem;

std::pmr::vector<float> x(N, &mem);
std::pmr::vector<float> y(N, &mem);

// ...
```

A Non-Initialising Allocator

```
template <class T>
struct no_init_allocator : std::pmr::polymorphic_allocator<T> {
  // constructors

  template <typename U> void construct(U* p) {
    ::new (static_cast<void*>(p)) U;
  }
};

template <class T>
using vector = std::vector<T, no_init_allocator<T>>;
```

A Non-Initialising Allocator

```
template <class T>
struct no_init_allocator : std::pmr::polymorphic_allocator<T> {
  // constructors

  template <typename U> void construct(U* p) {
    ::new (static_cast<void*>(p)) U;
  }
};

template <class T>
using vector = std::vector<T, no_init_allocator<T>>;
```

Device Only Memory Allocation

```
device_memory_resource mem;

vector<float> x(N, &mem);
vector<float> y(N, &mem);

// …
```

Intermediate Memory Resources

```
device_memory_resource mem;
std::pmr::monotonic_buffer_resource chunk(&mem);

vector<float> x(N, &chunk);
vector<float> y(N, &chunk);

// …
```

Memory Allocation Across Functions

```
std::pmr::vector<float>
get_x(std::pmr::memory_resource* mem);

TEST_CASE("cppcon-6", "[CUDA]") {
    unified_memory_resource unified_mem;
    device_memory_resource device_mem;

    auto const x = get_x(&unified_mem);
    auto const N = x.size();
    vector<float> y(N, &device_mem);
}
```

Questions About Memory Allocation?

Host vs Device Functions

Execution Space Specifiers

```
// cpu
void add_cpu(int n, float* x, float* y) {
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}

// gpu
__global__ void add_gpu(int n, float* x, float* y) {
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}
```

Execution Space Specifiers

The `__global__` execution space specifier declares a function as being a kernel. Such a function is:

- Executed on the device,
- Callable from the host.

The `__device__` execution space specifier declares a function that is:

- Executed on the device,
- Callable from the device only.

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#function-declaration-specifiers

Execution Space Specifiers

The **`__host__`** execution space specifier declares a function that is:

- Executed on the host,
- Callable from the host only.

It is equivalent to declare a function with only the __host__ execution space specifier or to declare it without any of the __host__, __device__, or __global__ execution space specifier; in either case the function is compiled for the host only.

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#host

Example Annotated Code

```
__host__ __device__ void _add(std::size_t n, float const* x, float* y) {
    for (std::size_t i = 0; i < n; ++i) {
        y[i] = x[i] + y[i];
    }
}

void add_cpu(std::size_t n, float const* x, float* y) {
    _add(n, x, y);
}

__global__ void add_gpu(int n, float const* x, float* y) {
    _add(n, x, y);
}
```

Libraries Not Under Your Control

```
template <std::size_t m> void _add(
    std::size_t n,
    std::array<float, m> const* x,
    std::array<float, m>* y) {
    for (std::size_t i = 0; i < n; ++i)
        for (std::size_t j = 0; j < m; ++j)
            y[i][j] = x[i][j] + y[i][j];
}
```

Using constexpr

4.2.3.17. --expt-relaxed-constexpr (-expt-relaxed-constexpr)

Experimental flag: **Allow host code to invoke __device__constexpr functions, and device code to invoke __host__constexpr functions.**

Note that the behavior of this flag may change in future compiler releases.

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#options-for-altering-compiler-linker-behavior-expt-relaxed-constexpr

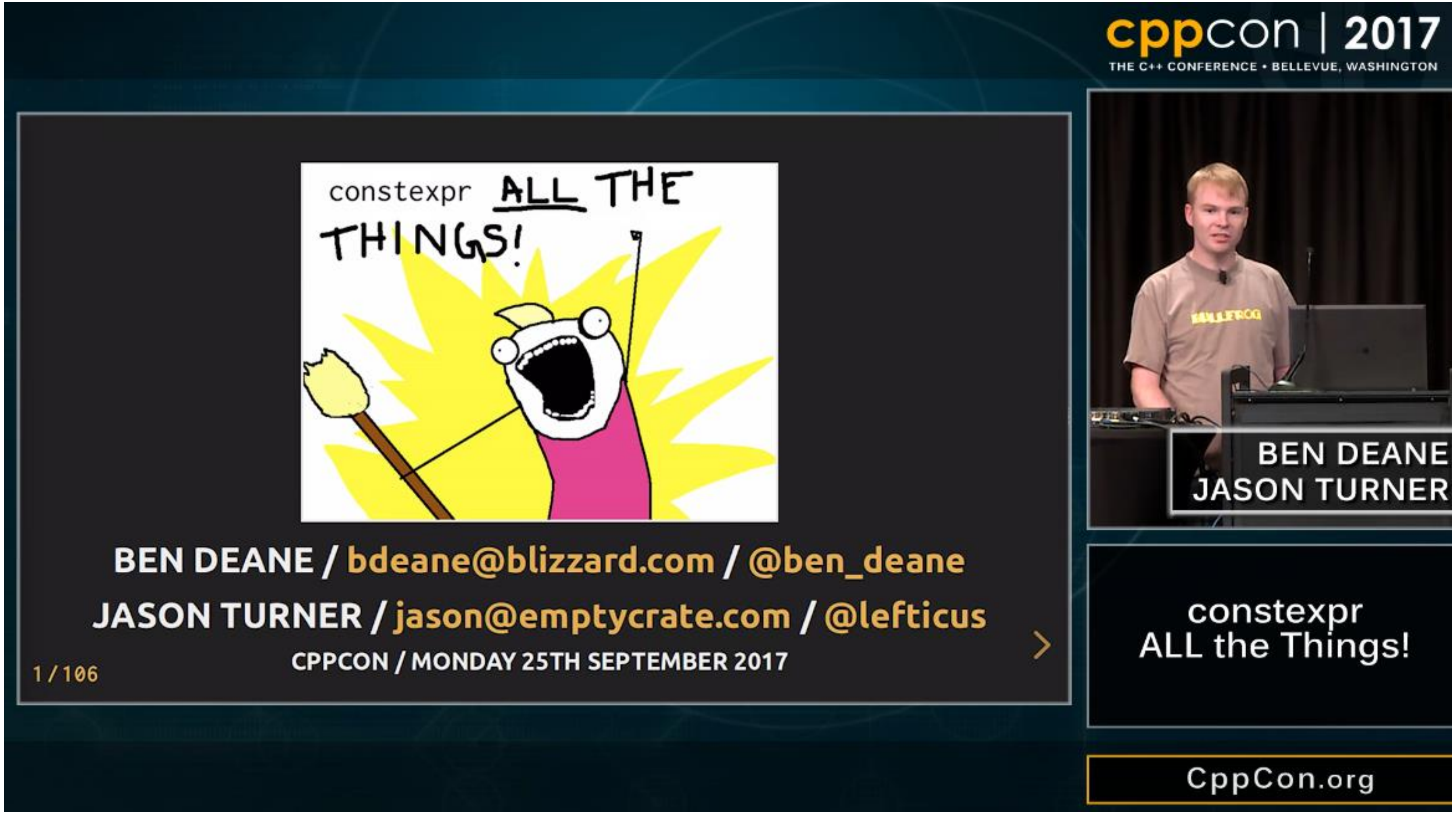
Using constexpr

```
constexpr void _add(std::size_t n, float const* x, float* y) {
    for (std::size_t i = 0; i < n; ++i) {
        y[i] = x[i] + y[i];
    }
}

void add_cpu(std::size_t n, float const* x, float* y) {
    _add(n, x, y);
}

__global__ void add_gpu(int n, float const* x, float* y) {
    _add(n, x, y);
}
```

Using constexpr

```
template <std::size_t m> constexpr void _add(
    std::size_t n,
    std::array<float, m> const* x,
    std::array<float, m>* y) {
    for (std::size_t i = 0; i < n; ++i)
        for (std::size_t j = 0; j < m; ++j)
            y[i][j] = x[i][j] + y[i][j];
}
```

constexpr ALL the Things!



constexpr Opens Up Third Party Libraries

- std library
- GSL Guidelines Support Library
- …

No Work is Less Work Than Some Work

- Your existing code should be tested and known to work.
- The fewer lines of code you add the less you need to debug.
- You do need to test that your code is appropriately marked constexpr.
  - Add tests that execute at compile time and fail to compile if the test fails. This ensures:
    - You do not call a non-constexpr function from a constexpr function.
    - That the behavior of your constexpr function matches your runtime behaviour (they can diverge).

Questions About constexpr/Execution Space Specifiers?

Return on Investment

std::pmr

- Allows control over allocation of memory necessary for access by GPU.
- May improve performance of your CPU code by:
  - Reducing the number of calls to the allocator.
  - Improving locality of objects.
  - Providing a way to instrument your code and identify inefficiencies.
- Requires your std library support std::pmr.

constexpr

- constexpr expands the set of functions available to the GPU.
- May improve performance by:
  - Moving work from runtime to compile time.
  - Making dimensions of vectors/matrices available at compile time.

Concluding Remarks

Questions?

Bowie Owens

bowie.owens@csiro.au