TECHNISCHE UNIVERSITÄT DRESDEN

Zentrum für Informationsdienste und Hochleistungsrechnen (ZIH)

Hauptseminar: Rechnerarchitektur und Programmierung

Neuerungen in C++11, 14 und 17 im Kontext Performance-Analyse von HPC-Anwendungen

Jan Stephan (jan.stephan@mailbox.tu-dresden.de)

Betreuer: Ronny Brendel (ronny.brendel@tu-dresden.de)

07. Juli 2016

ZIH
Zentrum für Informationsdienste und Hochleistungsrechnen

Gliederung

- Einleitung
  - Moderne C++-Philosophie
  - Messmethodik
- Beispiel: Vektoraddition mit CUDA
  - Spracherweiterungen: `auto`, `constexpr`, `using`, `enum class`
  - Bibliothekserweiterung: smart pointers
- Beispiel: Producer-Consumer-Queue
  - Bibliothekserweiterungen: `thread`, `mutex`, `atomic`
- Empfehlungen für die Unterstützung in Score-P und Vampir
- Ausblick auf C++17
- Weiterführende Literatur

Moderne C++-Philosophie

C++ feels like a new language. That is, I can express my ideas more clearly, more simply, and more directly in C++11 than I could in C++98. Furthermore, the resulting programs are better checked by the compiler and run faster.

- Bjarne Stroustrup[TCPP]

Moderne C++-Philosophie

Drücke Ideen direkt im Code aus!

[CPPCG](P.1)

Moderne C++-Philosophie → Suche in C++03

```
void do_something(vector<string>& v)
{
    string val;
    cin >> val;
    // …
    int index = -1; // unnötig kompliziert
    for(int i = 0; i < v.size(); ++i)
    {
        if(v[i] == val)
        {
            index = i;
            break;
        }
    }
    // …
}
```

Moderne C++-Philosophie → Suche in C++11

```
void do_something(vector<string>& v)
{
    string val;
    cin >> val;
    // …
    auto p = find(begin(v), end(v), val);
    // …
}
```

Moderne C++-Philosophie

Drücke Deine Absicht aus!

[CPPCG](P.3)

Moderne C++-Philosophie → Iteration in C++03

```
int i = 0;
while(i < v.size())
{
    // tue etwas mit v[i]…
}
```

Moderne C++-Philosophie → Iteration in C++11 / C++17

```
for(const auto& x : v) { /* tue etwas mit x */ }

for(auto&& x : v) { /* tue etwas mit x */ }

for_each(begin(v), end(v), [](int x) { /* tue etwas mit x */ });

for_each(execution::par, begin(v), end(v), [](int x) {
    /* tue etwas mit x */
});
```

Moderne C++-Philosophie

Erzeuge keine Ressourcenlecks!

[CPPCG](P.8)

Moderne C++-Philosophie → Ressourcenakquise in C

```
void f(char* name)
{
    FILE* input = fopen(name, "r");
    // …
    if(something) return;
    // …
    fclose(input);
}
```

Moderne C++-Philosophie → Ressourcenakquise in C++

```
void f(char* name)
{
    ifstream input(name);
    // …
    if(something) return;
    // …
}
```

→ RAII (Resource Acquisition Is Initialization)

Moderne C++-Philosophie

Benutze die Standardbibliothek!

[STC]

Moderne C++-Philosophie → Standardbibliothek

- `char*`-Strings → `std::string`
- `[]`-Arrays → `std::vector / std::array`
- Zeigerarithmetik → `<algorithm>`
- `new / delete` → `make_shared / make_unique`

Moderne C++-Philosophie → Folgen

- höherer Abstraktionsgrad
- weniger Quelltext
- messbar schnelleres Programm

[IPM]

Messmethodik

- System:
  - Betriebssystem: Arch Linux (x86_64)
  - Kernel 4.6.2
  - Compiler: gcc 6.1.1, clang 3.8.0
  - Prozessor: Intel Core i5-3230M (2,6GHz, Turbo Boost ausgeschaltet)
- Messung:
  - `clock_gettime(CLOCK_MONOTONIC, …);`

Median/Minimum/Maximum

Vektoraddition mit CUDA → Kernel (alt)

```
__global__ void vector_add(const int* __restrict__ a,
                           const int* __restrict__ b,
                           int* __restrict__ c,
                           size_t size)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if(i < size)
        c[i] = a[i] + b[i];
}
```

Vektoraddition mit CUDA → Kernel (neu)

```
__global__ void vector_add(const std::int32_t* __restrict__ a,
                           const std::int32_t* __restrict__ b,
                           std::int32_t* __restrict__ c,
                           std::size_t size)
{
    auto i = threadIdx.x + blockIdx.x * blockDim.x;
    if(i < size)
        c[i] = a[i] + b[i];
}
```

Neuerung in C++11: auto

- Spracherweiterung im C++11-Standard: `auto` [CPP11](§7.1.6.4)
- für Variablen:

```
auto x = 0; // wenn der Datenyp unwichtig ist
auto f = float(0); // bei Festlegung auf einen Datentypen
auto two = std::sqrt(4); // Herleitung aus Rückgabetypen
```

- für Funktionen:

```
auto foo(std::int32_t a, std::int32_t b) -> std::int32_t;

template <class T, class U>
auto bar(T a, U b) -> decltype(a + b);
```

Vektoraddition mit CUDA → Vektorerstellung (alt)

```
#define SIZE = 1000;

// …

for(size_t i = 0; i < SIZE; ++i)
{
    host_a[i] = std::rand();
    host_b[i] = std::rand();
}
```

Vektoraddition mit CUDA → Vektorerstellung (neu)

```
constexpr auto size = 1000;

// …

std::generate(a.get(), a.get() + size, std::rand);
std::generate(b.get(), b.get() + size, std::rand);
```

Neuerung in C++11: constexpr

- Spracherweiterung im C++11-Standard: `constexpr` [CPP11](§7.1.5)
- für Variablen:

```
constexpr auto i = 0; // Wert steht zur Compilezeit fest
```

- für Funktionen:

```
constexpr auto max(std::int32_t a, std::int32_t b) -> std::int32_t
{
    // Auswertung zur Compile-Zeit möglich
    return (a < b) ? a : b;
}
```

- kombiniert:

```
constexpr auto i = max(2, 3); // Auswertung zur Compile-Zeit

// Auswertung zur Compile-Zeit oder zur Laufzeit
auto j = max(x, y);
```

Vektoraddition mit CUDA → Speicherverwaltung (alt)

```
int* host_a = NULL;
cudaMallocHost((void**) &host_a, SIZE * sizeof(int));

int* dev_a = NULL;
cudaMalloc((void**) &dev_a, SIZE * sizeof(int));

// …

cudaFree(dev_a);
cudaFreeHost(host_a);
```

Vektoraddition mit CUDA → Speicherverwaltung (neu)

```
auto host_a = make_host_ptr<std::int32_t>(size);
auto dev_a = make_device_ptr<std::int32_t>(size);
```

Vektoraddition mit CUDA → Speicherverwaltung (neu)

```
template <class T>
auto make_device_ptr(std::size_t size) -> device_ptr<T>
{
    auto p = static_cast<T*>(nullptr);
    cudaMalloc(reinterpret_cast<void**>(&p), size * sizeof(T));
    return device_ptr<T>(p);
}

template <class T>
auto make_host_ptr(std::size_t size) -> host_ptr<T>
{
    auto p = static_cast<T*>(nullptr);
    cudaMallocHost(reinterpret_cast<void**>(&p), size * sizeof(T));
    return host_ptr(p);
}
```

Vektoraddition mit CUDA → Speicherverwaltung (neu)

```
struct device_deleter
{
    auto operator()(void* p) -> void { cudaFree(p); }
};

struct host_deleter
{
    auto operator()(void* p) -> void { cudaFreeHost(p); }
};

template <class T>
using device_ptr = std::unique_ptr<T[], device_deleter>;

template <class T>
using host_ptr = std::unique_ptr<T[], host_deleter>;
```

Neuerung in C++11: Smart pointers

- Bibliothekserweiterung im C++11-Standard: smart pointers [CPP11](§20.7)
  - `unique_ptr`
  - `shared_ptr`
  - `weak_ptr`
- veraltet: `auto_ptr`

Neuerungen in C++11: using

- Spracherweiterung in C++11: **using** [CPP11](§7.1.3/2)

```
typedef int my_type; // alt
using my_type = int; // neu

typedef void (*func_ptr)(double); // alt
using func_ptr = void (*)(double); // neu

template <class T>
using my_vec = std::vector<T, my_allocator>; // neu
```

Vektoraddition mit CUDA → Kopiervorgang (alt)

```
cudaMemcpy(dev_a, host_a, SIZE * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, host_b, SIZE * sizeof(int), cudaMemcpyHostToDevice);

// …

cudaMemcpy(host_c, dev_c, SIZE * sizeof(int), cudaMemcpyDeviceToHost);
```

Vektoraddition mit CUDA → Kopiervorgang (neu)

```
cuda_copy(dev_a, host_a, size);
cuda_copy(dev_b, host_b, size);

// …

cuda_copy(host_c, dev_c, size);
```

Vektoraddition mit CUDA → Kopiervorgang (neu)

```
template <class Dst, class Src>
auto cuda_copy(Dst& dst, const Src& src, std::size_t size) -> void
{
    cudaMemcpy(dst.get(), src.get(),
               size * sizeof(typename Dst::element_type),
               direction<Src::loc, Dst::loc>::value);
}
```

Vektoraddition mit CUDA → Kopiervorgang (neu)

```
enum class location { device, host };

template <location src, location dst> struct direction {};
template <> struct direction<location::host, location::device>
{
    static constexpr auto value = cudaMemcpyHostToDevice;
};

// etc.

template <class T, class Deleter, location l> class cuda_ptr
{
    public:
        static constexpr auto loc = location(l);
        using element_type = T;
    // Wrapper um unique_ptr<T[], Deleter> …
};

template <class T>
using device_ptr = cuda_ptr<T, device_deleter, location::device>;
```

Neuerungen in C++11: enum class

- Spracherweiterung in C++11: `enum class` [CPP11](§7.2)

```
enum color { red, green, blue, yellow };
enum traffic_light { red, yellow, green };
set_color(yellow); // oops…

enum class color { red, green, blue, yellow };
enum class traffic_light { red, yellow, green };
set_color(yellow); // Fehler: unbekannter Ausdruck
set_color(traffic_light::yellow); // Fehler: falscher Typ
set_color(color::yellow); // ok
```

Vektoraddition mit CUDA: Kernelaufruf (alt)

```
int block_size = 1024;
int grid_size = (int) std::ceil((float) SIZE / block_size);

vec_add<<<grid_size, block_size>>>(dev_a, dev_b, dev_c, SIZE);
```

Vektoraddition mit CUDA: Kernelaufruf (neu)

```
cuda_launch(vec_add, dev_a.get(), dev_b.get(), dev_c.get(), size);
```

Vektoraddition mit CUDA: Kernelaufruf (neu)

```
template <class... Args>
auto cuda_launch(void (*kernel)(Args...), Args... args) -> void
{

    // ermittle grid_size und block_size
    kernel<<<grid_size, block_size>>>(args...);
}
```

Neuerung in C++11: variadische Templates

- Spracherweiterung in C++11: variadische Templates [CPP11](§14.2.15)

Vektoraddition mit CUDA → Zusammenfassung

```
constexpr auto size = 1000;

auto host_a = make_host_ptr<std::int32_t>(size);
auto host_b = make_host_ptr<std::int32_t>(size);
auto host_c = make_host_ptr<std::int32_t>(size);

auto dev_a = make_device_ptr<std::int32_t>(size);
auto dev_b = make_device_ptr<std::int32_t>(size);
auto dev_c = make_device_ptr<std::int32_t>(size);

std::generate(host_a.get(), host_a.get() + size, std::rand);
std::generate(host_b.get(), host_b.get() + size, std::rand);

cuda_copy(dev_a, host_a, size);

cuda_copy(dev_b, host_b, size);

cuda_launch(vec_add, dev_a.get(), dev_b.get(), dev_c.get(), size);

cuda_copy(host_c, dev_c, size);
```

Fallbeispiel: Producer-Consumer-Queue

- Aufgabe: Implementierung einer Producer-Consumer-Queue[PPP]
- Probleme vor C++11:
  - kein plattformunabhängiger Weg (Pthreads, Win32-Threads, …)
  - kein nativer „C++-Weg“ (RAII [insbesondere Destruktoren], …), Wrapper erforderlich
  - keine plattformunabhängige Unterstützung für Atomics
- seit C++11:
  - `<thread>` zur Erzeugung von Threads
  - `<mutex>` und `<condition_variable>` zur Threadsynchronisierung
  - `<atomic>` für atomare Operationen
  - `<future>` für asynchrone Operationen (ohne explizite Threaderzeugung durch Programmierer, hier nicht gezeigt)

Producer-Consumer-Queue: Threaderzeugung

```
auto producer(queue<object>& q) -> void
{
    auto o = create_object();
    q.push(move(o));
}

auto consumer(queue<object>& q) -> void
{
    auto o = q.pop();
    do_something_with(o);
}

auto main() -> int
{
    auto q = queue();
    auto p = thread(&producer, ref(q)); // thread kopiert Parameter
    auto c = thread(&consumer, ref(q)); // ref erzwingt Referenz
    p.join();
    c.join();

    return 0;
}
```

Producer-Consumer-Queue: Implementierung mit Mutexes

```
template <class T>
class Queue {
        auto push(T t) -> void {
            auto lock = unique_lock<mutex>(mutex_);

            queue_.push(move(item));
            cv_.notify_one();
        }

        auto pop() -> T {
            auto lock = unique_lock<mutex>(mutex_);
            while(queue_.empty())
                cv_.wait(lock);

            auto ret = move(queue_.front());
            queue_.pop();

            return ret;
        }

        condition_variable cv_;
        mutex mutex_;
}
```

Producer-Consumer-Queue: Implementierung mit Atomics

```
template <class T> class Queue {
    auto push(T t) -> void {
        while(lock_)
            thread::this_thread::yield();

        lock_ = true;
        queue_.push(move(t));
        lock_ = false;
    }

    auto pop() -> T {
        while(lock_ || queue_.empty())
            thread::this_thread::yield();

        lock_ = true;
        auto ret = move(queue_.front());
        queue_.pop();
        lock_ = false;

        return ret;
    }

    atomic_bool lock_; // Initialisierung: false
};
```

Empfehlungen für die Unterstützung in Score-P und Vampir

- Probleme:
  - Inlining der meisten STL-Methoden
  - keine Unterstützung von Threads in gelinkten Bibliotheken
- Lösungsvorschläge:
  - Einrichtung eines Zählers, der die Aufrufe der STL-Methoden hochzählt
    - nur die vom Programmierer aufgerufenen Methoden!
  - nachträgliche Registrierung „externer“ Threads (statt Programmabsturz)

Ausblick auf C++17

- zusätzliche mathematische Funktionen (Polynomfunktionen, Betafunktion, elliptische Integrale, …)
  - gcc 6.1: `<tr1/cmath>`
  - Boost.Math
- Routinen zum Umgang mit dem Dateisystem (Dateien und Ordner)
  - gcc 5.3: `<experimental/filesystem>`
  - Boost.Filesystem
- parallelisierte Algorithmen (aus `<algorithm>`)
  - noch keine Compiler-Unterstützung

Weiterführende Literatur

- Bjarne Stroustrup: *A Tour of C++*, Addison-Wesley, 2014
- Bjarne Stroustrup: *The C++ Programming Language*, Addison-Wesley, 2013
- Scott Meyers: *Effective Modern C++*, O'Reilly, 2014
- Anthony Williams: *C++ Concurrency in Action*, Manning, 2012
- Stanley B. Lippmann, Josee Lajoie, Barbara E. Moo: *C++ Primer*, Addison-Wesley, 2012

- Bjarne Stroustrup: *C++11 FAQ*, http://www.stroustrup.com/C++11FAQ.html
- Herb Sutter: *Guru of the Week*, https://herbsutter.com/gotw/

Quellen

- [CPP11] ISO/IEC JTC1/SC22/WG21: *International Standard ISO/IEC 14882:2011(E) – Programming Language C++*, ISO/IEC, 2011
- [CPP14] ISO/IEC JTC1/SC22/WG21: *International Standard ISO/IEC 14882:2014(E) – Programming Language C++*, ISO/IEC, 2014
- [CPP17] ISO/IEC JTC1/SC22/WG21: *Working Draft, Standard for Programming Language C++*, ISO/IEC, Entwurf vom 30.05.2016
- [EP1] Project Euler: *Problem 1. Multiples of 3 and 5*, letzte Aktualisierung unbekannt, zuletzt abgerufen am 12.06.2016, https://projecteuler.net/problem=1
- [EP22] Project Euler: *Problem 22. Names scores*, letzte Aktualisierung unbekannt, zuletzt abgerufen am 12.06.2016, https://projecteuler.net/problem=22
- [MCS] Herb Sutter: *Back to Basics: Modern C++ Style*, Folien zum Vortrag auf der CppCon 2014, letzte Änderung am 13.09.2014, zuletzt abgerufen am 12.06.2016, https://github.com/CppCon/CppCon2014/tree/master/Presentations/Back%20to%20the%20Basics!%20Essentials%20of%20Modern%20C%2B%2B%20style

Quellen

- [TCPP] Bjarne Stroustrup: *A Tour of C++*, Addison-Wesley, 2014
- [CPPCG] Bjarne Stroustrup, Herb Sutter: *C++ Core Guidelines*, letzte Aktualisierung am 05. April 2016, zuletzt abgerufen am 22.06.16, http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
- [PPP] Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill: *Patterns for Parallel Programming*, Addison-Wesley, 2005
- [STC] Kate Gregory: *Stop Teaching C*, Folien zum Vortrag auf der CppCon 2015, letzte Änderung am 03.10.2015, zuletzt abgerufen am 29.06.16, https://github.com/CppCon/CppCon2015/tree/master/Presentations/Stop%20Teaching%20C
- [IPM] J. Daniel Garcia, Bjarne Stroustrup: *Improving performance and maintainability through refactoring in C++11*, August 2015

Vielen Dank!