



Praxis

Hello World mit Data Parallel C++

Ein kurzes Tutorial für Kenner von C++ als Einführung in das Programmieren der Zukunft

von Jeff Hammond



Mit Hardwarebeschleunigern für HPC- und KI-Systeme wird das Programmieren in Zukunft immer paralleler werden. Spannende Zeiten für Entwickler, die heute mit C++ arbeiten. Eine der grundlegenden Neuerungen, die oneAPI für sie bringt, ist das Programmiermodell Data Parallel C++, kurz DPC++. Dabei handelt es sich um ein modernes, paralleles C++ für heterogene Architekturen. Die Basis bildet Khronos SYCL. Ein kurzes Tutorial verschafft Kennern von C++ Klarheit. „Hello World“ macht in einem Programmiermodell, das viele Dinge parallel macht, nicht viel Sinn. Wir starten deshalb mit der Vektoraddition als dem „Hello World“ der parallelen Programmierung. Die Operation, die wir implementieren möchten, ist SAXPY. Das steht kurz für A-mal X plus Y mit einfacher Genauigkeit. Diese Operation kann in C oder C++ wie folgt implementiert werden:

```
for (size_t i = 0; i < length; ++i) {
    Z[i] += A * X[i] + Y[i];
}
```

Es gibt viele weitere Möglichkeiten, diese Operation in C++ zu formulieren. Zum Beispiel könnten wir Bereiche verwenden, die den Code etwas mehr wie die folgende SYCL-Version aussehen lassen.

Hier ist dieselbe Schleife, diesmal programmiert in SYCL; die Erklärung folgt dann schrittweise:

```
h.parallel_for<class saxpy> (sycl::range<1>{length}, [=] (sycl::id<1> it) {
    const int i = it[0];
    Z[i] += A * X[i] + Y[i];
});
```

Wie Sie vielleicht erraten, ist mit `parallel_for` eine parallel ausführbare for-Schleife gemeint. Der Schleifenkörper wird Lambda genannt; Lambda ist hier also der Code, der wie `[..] {..}` aussieht.

Der Schleifeniterator wird als `sycl::range` und `sycl::id` ausgedrückt. In unserem einfachen Beispiel sind beide eindimensional, wie durch `<1>` angegeben. SYCL-Ranges und -IDs können ein-, zwei- oder dreidimensional sein; bei OpenCL und CUDA gibt es die gleiche Einschränkung.

Es mag etwas ungewohnt sein, Schleifen in dieser Weise zu schreiben, aber es stimmt mit der Funktionsweise von Lambda-Ausdrücken überein. Wer jemals in **parallel STL**, **TBB**, **Kokkos** oder **RAJA** programmiert hat, wird dieses Muster kennen.

Möglicherweise wundern Sie sich über das Template-Argument `<class saxpy>` für den Befehl `parallel_for`. Dies ist nur eine der Möglichkeiten, den Kernel zu benennen, was wiederum erforderlich ist, da SYCL möglicherweise nicht nur mit dem lokalen Gerätecompiler, sondern auch mit dem C++-Compiler eines anderen Hosts verwendet werden soll. In diesem Fall benötigen die beiden Compiler eine Möglichkeit, sich auf den Kernelnamen zu einigen. Bei vielen SYCL-Compilern, wie zum Beispiel auch bei Intel DPC++, ist dies nicht erforderlich. Mit der Option `fsycl-unnamed-lambda` können wir dem Compiler auch sagen, dass er sich keine „Gedanken“ über die Suche nach Namen machen soll. Wir werden an dieser Stelle nicht den Versuch einer Erklärung des `h` in `h.parallel_for` wagen, sondern das Thema später behandeln.

Herausforderungen der heterogenen Programmierung

Zu den Herausforderungen der heterogenen Programmierung zählen die verschiedenen Arten von Verarbeitungselementen – und häufig auch unterschiedliche Speichertypen. Diese Dinge machen Compiler und Runtimes komplizierter. Das SYCL-Programmiermodell erlaubt solch eine heterogene Ausführung, allerdings auf einer viel höheren Abstraktionsebene als OpenCL. Auch ist nicht alles explizit. Im Gegensatz zu anderen gängigen GPU-Programmiermodellen können die SYCL-Kernels in den Host-Programmfluss integriert werden, was die Lesbarkeit verbessert.

Bei DPC++ gibt es eine weitere elementare Voraussetzung: Wann immer wir auf einem Gerät rechnen möchten, müssen wir eine Arbeitswarteschlange erstellen:

```
sycl::queue q(sycl::default_selector{});
```

Der Standard-Selector bevorzugt eine GPU (falls vorhanden), andernfalls eine CPU. Wir können Warteschlangen erstellen, die bestimmten Gerätetypen zugeordnet sind:

```

sycl::queue q(sycl::host_selector{}); // run on the CPU without a runtime (i.e., no OpenCL)
sycl::queue q(sycl::cpu_selector{}); // run on the CPU with a runtime (e.g., OpenCL)
sycl::queue q(sycl::gpu_selector{}); // run on the GPU
sycl::queue q(sycl::accelerator_selector{}); // run on an FPGA or other accelerator

```

Die Host- und CPU-Selektoren können selbst dann zu erheblich unterschiedlichen Ergebnissen führen, wenn sie auf dieselbe Hardware abzielen. Das kann daran liegen, dass der Host-Selector möglicherweise eine für das Debuggen optimierte sequenzielle Implementierung verwendet, während der CPU-Selector die OpenCL-Runtime verwendet und über alle Kerne läuft. Außerdem kann der OpenCL Just-in-Time Compiler (JIT) möglicherweise anderen Code generieren, weil er einen ganz anderen Compiler verwendet. Gehen Sie also nicht davon aus, dass Host und CPU in SYCL dasselbe bedeuten, nur weil der Host eine CPU ist.

Verwalten von Daten in SYCL

Die kanonische Methode zum Verwalten von Daten in SYCL arbeitet mit Puffern. Ein SYCL-Buffer ist ein „undurchsichtiger“ Container. Dies ist zwar ein elegantes Design, doch einige Anwendungen arbeiten mit Pointern, die von der Erweiterung „Unified Shared Memory“ (USM) bereitgestellt werden (die wird später erläutert).

```

// T is a data type, e.g., float
std::vector<T> h_X(length,xval);
sycl::buffer<T,1> d_X { h_X.data(), sycl::range<1>(h_X.size()) };

```

Im gerade vorgestellten Beispiel weist der Benutzer einen C++-Container auf dem Host zu und übergibt ihn dann an SYCL. Bis der Destructor des SYCL-Puffers aufgerufen wird, kann der Benutzer einzig und allein über einen SYCL-Mechanismus auf die Daten zugreifen. SYCL-Zugriffsfunktionen sind der wichtigste Aspekt der SYCL-Datenverwaltung mit Puffern, die unten erläutert werden.

Da für Geräte-Code möglicherweise ein anderer Compiler oder Generierungsmechanismus als für den Host erforderlich ist, müssen Abschnitte des Geräte-Codes eindeutig identifiziert werden. Unten sehen wir, wie dies in SYCL 1.2.1 aussieht. Wir verwenden die Methode `submit`, um die Arbeit in die Gerätewarteschlange `q` zu stellen. Diese Methode gibt einen opaken Handler zurück, gegen den wir die Kernels ausführen, in diesem Fall via `parallel_for`.

```

q.submit([&](sycl::handler& h) {
    ...
    h.parallel_for<class nstream> (sycl::range<1>{length}, [=] (sycl::id<1> i) {
        ...
    });
});
q.wait();

```

Wir können die Ausführung des Codes auf den Geräten mit der Methode `wait()` synchronisieren. Es gibt granularere Methoden für das Synchronisieren – aber wir beginnen mit der einfachsten, der Hammermethode.

Einige Benutzer finden den obigen Code möglicherweise etwas weitschweifig, insbesondere im Vergleich zu Kokkos. Der Intel-Compiler für DPC++ unterstützt eine knappe Syntax, die wir im Folgenden erläutern.

Das letzte Puzzle-teil

```

q.submit([&](sycl::handler& h) {

    auto X = d_X.template get_access<sycl::access::mode::read>(h);
    auto Y = d_Y.template get_access<sycl::access::mode::read>(h);
    auto Z = d_Z.template get_access<sycl::access::mode::read_write>(h);

    h.parallel_for<class nstream> (sycl::range<1>{length}, [=] (sycl::id<1> i) {
        ...
    });
});

```

Kommen wir auf die SYCL-Zugriffsfunktionen zurück, das letzte Puzzleteil für unser erstes SYCL-Programm. Diese „Accessoren“ sind GPU-Programmierern vielleicht nicht vertraut, haben aber im Vergleich zu anderen Methoden einige schöne Eigenschaften. Während SYCL dem Programmierer gestattet, Daten explizit zu verschieben, indem er beispielsweise die Methode `copy()` verwendet, ist das bei den Accessor-Methoden nicht nötig. Sie können ein Datenflussdiagramm generieren, das der Compiler und die Runtime verwenden können, um Daten zum richtigen Zeitpunkt zu verschieben. Dies ist besonders effektiv, sobald mehrere Kernels sequenziell aufgerufen werden.

In diesem Fall leitet die SYCL-Implementierung ab, dass Daten wiederverwendet werden, und kopiert sie nicht unnötig auf den Host zurück. Wir können die Datenbewegung auch asynchron planen (d. h. überlappend mit der Code-Ausführung auf dem Gerät). Während erfahrene GPU-Programmierer dies manuell tun können, stellen wir häufig fest, dass SYCL-Accessoren zu einer besseren Performance führen als OpenCL-Programme, bei denen Programmierer Daten explizit verschieben müssen.

Weil Programmiermodelle, bei denen Pointer den Speicher handhaben, Schwierigkeiten mit SYCL-Accessoren haben, macht die USM-Erweiterung diese Accessoren überflüssig. USM fordert zwar den Programmierer mit Blick auf Datenverschiebung und Synchronisierung, verbessert jedoch die Kompatibilität von Legacy-Code, der Pointer verwendet.

Unser erstes SYCL-Programm

Hier sind alle Komponenten unseres SAXPY-Programms in SYCL:

```

std::vector<float> h_X(length,xval);
std::vector<float> h_Y(length,yval);
std::vector<float> h_Z(length,zval);

try {

    sycl::queue q(sycl::default_selector{});

    const float A(aval);

    sycl::buffer<float,1> d_X { h_X.data(), sycl::range<1>(h_X.size()) };
    sycl::buffer<float,1> d_Y { h_Y.data(), sycl::range<1>(h_Y.size()) };
    sycl::buffer<float,1> d_Z { h_Z.data(), sycl::range<1>(h_Z.size()) };

    q.submit([&](sycl::handler& h) {

        auto X = d_X.template get_access<sycl::access::mode::read>(h);
        auto Y = d_Y.template get_access<sycl::access::mode::read>(h);
        auto Z = d_Z.template get_access<sycl::access::mode::read_write>(h);

        h.parallel_for<class nstream>( sycl::range<1>(length), [=] (sycl::id<1> it) {
            const int i = it[0];
            Z[i] += A * X[i] + Y[i];
        });
    });
    q.wait();
}
catch (sycl::exception & e) {
    std::cout << e.what() << std::endl;
    return 1;
}

```

Der vollständige Quellcode für dieses Beispiel ist in diesem GitHub-Repository verfügbar:

<https://github.com/jeffhammond/dpcpp-tutorial>

Während dieses Programm perfekt funktioniert und auf vielen Plattformen implementiert werden kann, werden es einige Benutzer ziemlich „geschwätzig“ finden. Darüber hinaus ist es nicht kompatibel mit Bibliotheken und Frameworks, die den Speicher mithilfe von Pointern verwalten müssen. Um dieses Problem mit SYCL 1.2.1 zu beheben, hat Intel in DPC++ die USM-Erweiterung, die eine pointerbasierte Speicherverwaltung unterstützt.

USM unterstützt zwei wichtige Nutzungsmodelle, die wir im Folgenden darstellen. Das erste unterstützt die automatische Datenübertragung zwischen Host und Gerät. Das zweite dient zum expliziten Verschieben der Daten zu und von Geräten.

Die Details sind in der vorläufigen SYCL-2020-Spezifikation enthalten. Zu Beginn müssen Sie nur Nachfolgendes wissen. Das Argument `q` ist die Warteschlange, die dem Gerät zugeordnet ist, auf dem die zugewiesenen Daten gespeichert sind (entweder permanent oder temporär):

```

//shared allocation (can migrate between host and device)
auto d_X = sycl::malloc_shared<float>(length, q);

//device allocation (does not migrate)
auto d_X = sycl::malloc_device<float>(length, q);

// deallocation (works with any allocation type)
sycl::free(d_X, q);

```

Wenn wir die Gerätezuordnung verwenden, müssen Daten explizit verschoben werden (z. B. mithilfe der SYCL-

Methode `memcpy`). Dies verhält sich genauso wie bei `std::memcpy` (z. B. befindet sich das Ziel links):

```
//shared allocation (can migrate between host and device)
auto d_X = sycl::malloc_shared<float>(length, q);

//device allocation (does not migrate)
auto d_X = sycl::malloc_device<float>(length, q);

// deallocation (works with any allocation type)
sycl::free(d_X, q);
```

Wenn wir USM verwenden, sind keine Accessoren mehr erforderlich, was bedeutet, dass wir den obigen Kernel-Code vereinfachen können:

```
q.submit([&](sycl::handler& h) {
    h.parallel_for<class saxpy>(sycl::range<1>(length), ::id<1> i)
        d_Z[i] += A * d_X[i] + d_Y[i];
    });
```

Die vollständigen Arbeitsbeispiele für beide Versionen von USM finden Sie in diesem Repo mit den Namen `saxpy-usm.cc` bzw. `saxpy-usm2.cc`.

Falls Sie sich zwischenzeitlich gefragt haben, warum der opake Handler `h` in jedem dieser Programme erforderlich war: Es stellt sich heraus, dass er letztendlich doch nicht erforderlich ist. Das Folgende ist eine äquivalente Implementierung, die mit der vorläufigen Spezifikation SYCL 2020 möglich ist. Zudem können wir ausnutzen, dass Lambda-Namen in der vorläufigen Spezifikation SYCL 2020 nur optional sind. Zusammen sorgen diese beiden kleinen Änderungen dafür, dass die SYCL-Kernels dieselbe Länge haben wie die ursprüngliche Schleife in C++, die zu Beginn dieses Tutorials aufgeführt ist:

```
q.parallel_for(sycl::range<1>(length), [=](sycl::id<1> i) {
    d_Z[i] += A * d_X[i] + d_Y[i];
});
```

Wir haben mit drei Codezeilen begonnen, die sequenziell auf einer CPU ausgeführt werden. Wir enden mit drei Codezeilen, die parallel auf CPUs, GPUs, FPGAs und anderen Geräten ausgeführt werden.

Natürlich wird nicht alles so einfach sein wie SAXPY, aber zumindest wissen Sie jetzt, dass SYCL einfache Dinge nicht schwer machen wird, und es baut auf einer Reihe moderner C++-Funktionen und universeller Konzepte wie „parallel for“ auf, anstatt neue Dinge einzuführen, die erst erlernt werden müssen.

Literatur & Links

[1] Spezifikation von Khronos SYCL 1.2.1, www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf

[2] DPC++ language extensions, <https://github.com/intel/llvm/tree/sycl/sycl/doc/extensions>

Jeff Hammond

ist „Principal Engineer“ der Intel Data Center Group.

Bildnachweise

Intel

[AI Trendletter](#)

[Impressum](#)

|
[Kontakt & Anfrage](#)