TECHNISCHE UNIVERSITÄT DRESDEN

HZDR HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF

DRESDEN concept

Jan Stephan
Fakultät Informatik // Institut für Technische Informatik
Seniorprofessor Dr.-Ing. habil. Rainer G. Spallek

Entwicklung eines SYCL-Backends für die Alpaka-Bibliothek und dessen Evaluation mit Schwerpunkt auf FPGAs

Verteidigung der Diplomarbeit // 17. Dezember 2019

Erstgutachter: Prof. Dr-Ing. habil. Rainer G. Spallek
Zweitgutachter: Prof. Dr. rer. nat. Ulrich Schramm

Gliederung

Motivation und Ziel

FPGAs als Beschleuniger
- Einsatzzwecke
- Programmierung

Die SYCL-Spezifikation

Die Alpaka-Bibliothek

Implementierung des SYCL-Backends
- Struktur
- Probleme

Ergebnisse
- Nutzbarkeit
- Performanz

Fazit

Motivation und Ziel

Motivation

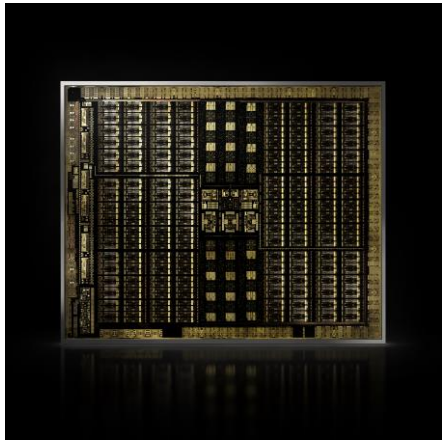
alpaka

OpenMP
Enabling HPC since 1997

NVIDIA
CUDA

OpenCL

SYCL™



Motivation

CPU / GPU	FPGA	ASIC
Allzweck-Hardware	Spezialisierte Hardware	Spezialisierte Hardware
Günstig	Günstig	Teuer
Nicht taktgenau	Taktgenau	Taktgenau
Nicht anpassbar	Anpassbar	Nicht anpassbar
Schnell (GHz)	Langsam (MHz)	Schnell (GHz)

→ Niedrige bis mittlere Stückzahlen
→ Latenzkritische Probleme

Ziel

Untersuchung des SYCL-Standards
- Existierende Implementierungen
- Nutzbarkeit
- FPGA-Schwerpunkt

Entwicklung eines Alpaka-SYCL-Backends
- Analyse der Schwierigkeiten und Unzulänglichkeiten
- Verifizierung durch reale Anwendung

FPGAs als Beschleuniger

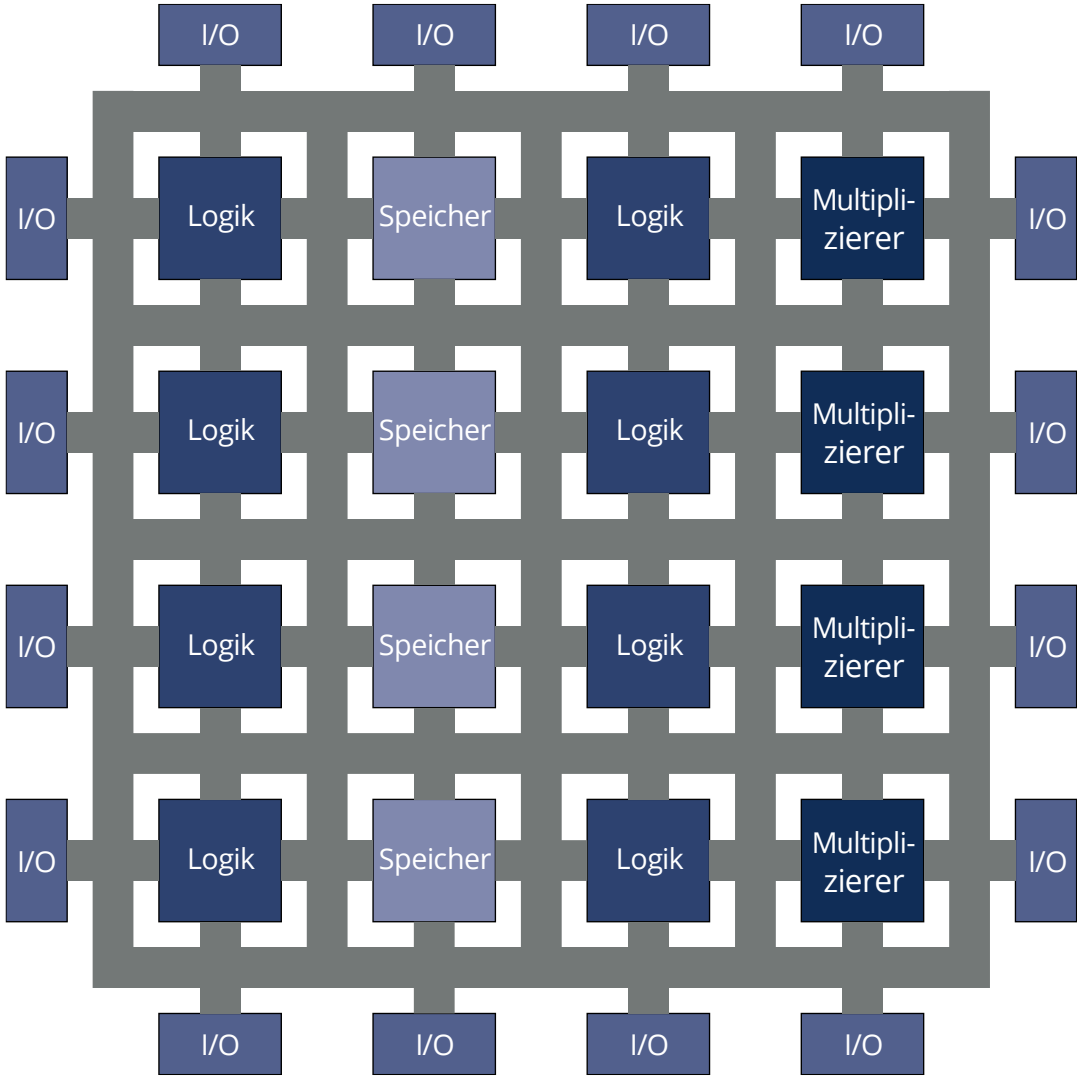
FPGAs als Beschleuniger

Aufbau

- Logikzellen mit geringer Komplexität
- Regelmäßige spaltenweise Feldstruktur
- Programmierbare Verdrahtungen
- Puffer für Ein- und Ausgabe (I/O)
- Weitere Elemente (Speicher, Multiplizierer, …)

Moderne FPGAs (Xilinx Virtex UltraScale+)

- *Configurable logic block* (CLB)
- *Input/output block* (IOB)
- Block RAM (36 kbit, paralleler Zugriff)
- UltraRAM (288 kbit, kein paralleler Zugriff)
- *Digital signal processor* (DSP)
- *Clock management tile* (CMT)



Nach [HS10], S. 10-14

FPGAs als Beschleuniger
Einsatzzwecke

- Schaltungsentwurf
  - *Rapid prototyping* für integrierte Schaltkreise
  - Einfache Fehlerbehebung
- Schaltkreise in kleiner Stückzahl
  - Einstiegskosten geringer als bei ASICs
  - Höhere Kosten pro Stück bei größeren Produktionsvolumen
- Microsoft: Inferenz tiefer neuraler Netzwerke [Fow+18, Chu+18]
  - Trainierte Netzwerke als Schaltung
- Microsoft: FPGAs als Netzwerkkarten [Fir+18]
- Amazon: FPGA-Cloud-Instanzen [Ama]
  - FPGAs als Beschleuniger [Di+17]

FPGAs als Beschleuniger
Programmierung // High-Level-Synthese

High-Level-Synthese

- Abstraktion der FPGA-Elemente
- Programmierung auf algorithmischer Ebene
- Festlegbare Randbedingungen
  - Ausrollen von Schleifen
  - Pipelining
  - Block RAM
  - etc.
- Hochsprachen
  - C / C++ / SystemC
  - OpenCL
  - **SYCL**

Die SYCL-Spezifikation

Die SYCL-Spezifikation

SYCL [KRH19]

- Offener Standard
  - Einheitliche Programmierschnittstelle
  - Implementierung durch Hardware-Hersteller oder Dritte
  - Herstellerspezifische Erweiterungen
- Basiert auf OpenCL
  - OpenCLs Konzepte und Portabilität
  - Keine Trennung zwischen Host- und Device-Quelltext
  - Moderne C++-Schnittstelle
- Implementierungen
  - ComputeCpp (Intel-CPUs, Intel-GPUs, NVIDIA-GPUs)
  - Intel (Intel-Hardware)
  - **Xilinx (FPGAs)**
  - hipSYCL (AMD-GPUs, NVIDIA-GPUs), sycl-gtx (OpenCL 1.2)

Die SYCL-Spezifikation

AXPY [Law+79]

$$\vec{y}' = a \cdot \vec{x} + \vec{y}$$

```
auto queue = sycl::queue{xilinx_selector{}};

const auto range = sycl::range<1>{1024};

auto buf_x = sycl::buffer<int, 1>{range};
auto buf_y = sycl::buffer<int, 1>{range};

/* Initialisierung auf Host-Seite */

queue.submit([&](sycl::handler& cgh)
{
    auto x = buf_x.get_access<sycl::access::mode::read>(cgh);
    auto y = buf_y.get_access<sycl::access::mode::read_write>(cgh);
    cgh.parallel_for<class axpy>(range,
    [=](sycl::item<1> work_item)
    {
        auto idx = work_item.get_id();
        y[idx] = a * x[idx] + y[idx];
    }
});
queue.wait(); // Synchronisierung
/* ab hier Zugriff durch Host möglich */
```

Die SYCL-Spezifikation

Xilinx-Erweiterungen

— Dataflow

```
sycl::xilinx::dataflow([&]()
{
    auto x = func_a();    // x ist ein Vektor
    func_b(x);
});
```

— Pipelining

```
sycl::xilinx::pipeline([&]()
{
        for(auto i = 0; i < 1024; ++i)
        {
            /* … */
        }
});
```

— Speicherzerlegung / Block RAM
- cyclic
- block
- complete

```
auto arr = sycl::xilinx::partition_array<int, 16,
                sycl::xilinx::partition::cyclic<4, 1>>{};
```

Die Alpaka-Bibliothek

Die Alpaka-Bibliothek

Alpaka: *Abstraction Library for Parallel Kernel Acceleration* [Wor15]

- quelloffene Abstraktionsbibliothek
  - Einheitliche Programmierschnittstelle
  - Implementierung durch *Helmholtz-Zentrum Dresden-Rossendorf / Institut für Strahlenphysik / Computergestützte Strahlenphysik*
  - Keine Trennung zwischen Host- und Device-Quelltext → inkompatibel zu OpenCL
  - C++-Bibliothek

- Strukturierung durch Konzepte
  - API gibt Konzept vor
  - Konzept wird durch hardware-spezifische API implementiert

- Implementierungen
  - NVIDIA-GPUs (CUDA, HIP)
  - AMD-GPUs (HIP)
  - CPUs (OpenMP, Threading Building Blocks, Boost Fiber, `std::thread`)

Die Alpaka-Bibliothek

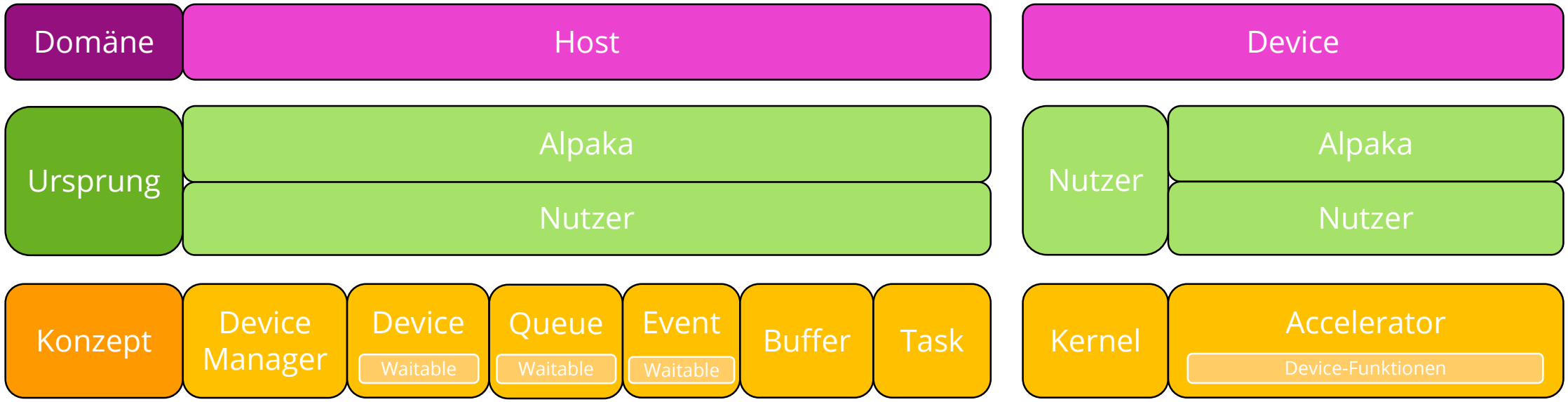
AXPY [Law+79]

$$\vec{y}' = a \cdot \vec{x} + \vec{y}$$

```
struct AxpyKernel
{
    template <typename TAcc, typename TIdx>
    ALPAKA_FN_ACC void operator()(
        const TAcc& acc, const TIdx& num_elements,
        const int a, const int* x, int* y) const
    {
        auto gt_idx = alpaka::idx::getIdx<
           alpaka::Grid, alpaka::Threads>(acc)[0u];
        auto te_ext = alpaka::workdiv::getWorkDiv<
           alpaka::Thread, alpaka::Elems>(acc)[0u];
        auto first = gt_idx * te_ext;

        if(first < num_elements)
        {
            auto last = first + te_ext;
            for(auto i = first; i < last; ++i)
                y[i] = a * x[i] + y[i];
        }
    }
};
```

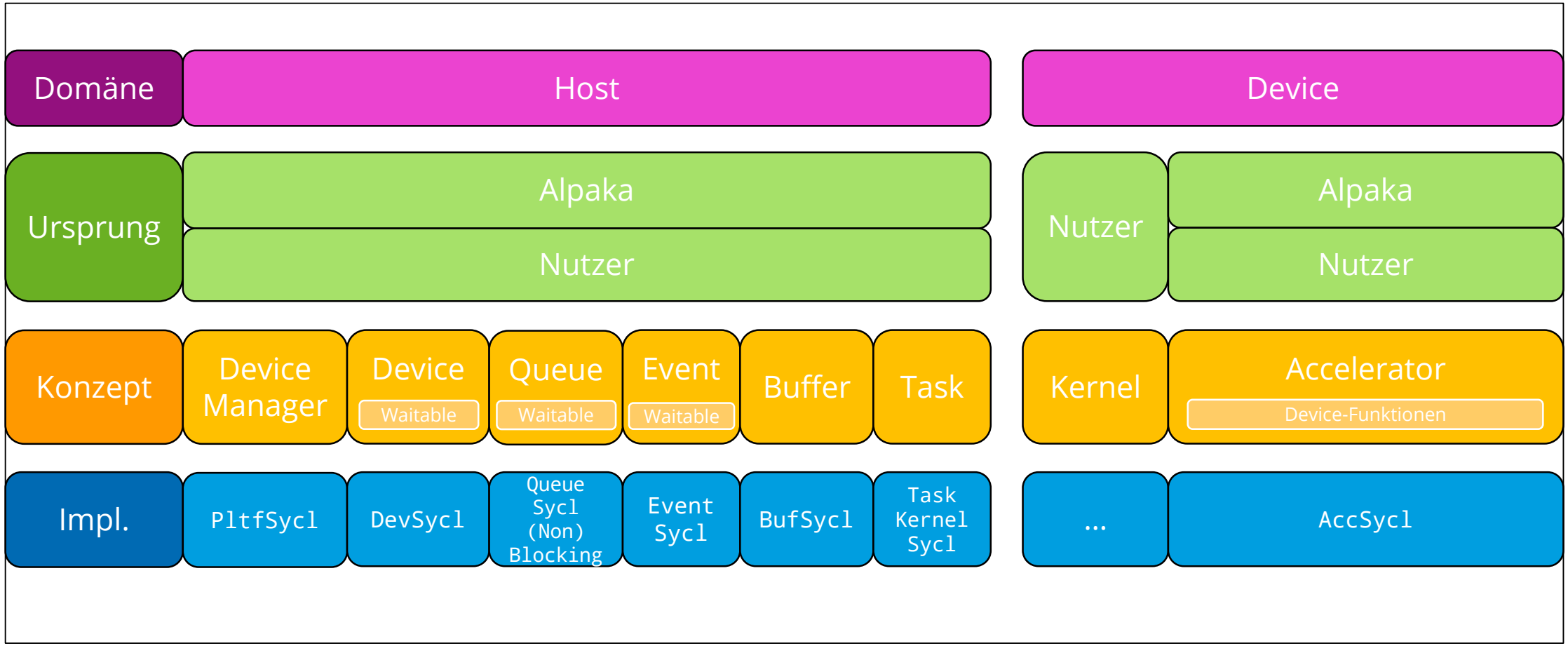
```
using Dim = alpaka::dim::DimInt<1u>;
using Idx = std::size_t;
using Acc = alpaka::acc::AccGpuCudaRt<Dim, Idx>;

auto host = alpaka::pltf::getDevByIdx<alpaka::Pltf::PltfCpu>(0u);
auto dev = alpaka::pltf::getDevByIdx<alpaka::Pltf::Pltf<Acc>>(0u);
auto queue = alpaka::queue::QueueCudaRtNonBlocking{dev};

const auto extent = alpaka::vec::Vec<Dim, Idx>{1024};
auto host_buf_x = alpaka::mem::buf::alloc<int, Idx>(host, extent);
auto host_buf_y = alpaka::mem::buf::alloc<int, Idx>(host, extent);
/* Initialisierung auf Host-Seite */
auto dev_buf_x = alpaka::mem::buf::alloc<int, Idx>(dev, extent);
auto dev_buf_y = alpaka::mem::buf::alloc<int, Idx>(dev, extent);

alpaka::mem::view::copy(queue, dev_buf_x, host_buf_x, extent);
alpaka::mem::view::copy(queue, dev_buf_y, host_buf_y, extent);

auto work_div = alpaka::workdiv::getValidWorkDiv<Acc>(dev, extent, Idx{1u});
auto task_kernel = alpaka::kernel::createTaskKernel<Acc>(work_div, AxpyKernel{},
                        1024, a, alpaka::mem::view::getPtrNative(dev_buf_x),
                        alpaka::mem::view::getPtrNative(dev_buf_y));

alpaka::queue::enqueue(queue, task_kernel);

alpaka::mem::view::copy(queue, host_buf_x, dev_buf_x, extent);
alpaka::mem::view::copy(queue, host_buf_y, dev_buf_y, extent);
alpaka::wait::wait(queue); // Synchronisierung
/* ab hier Zugriff durch Host möglich */
```

Die Alpaka-Bibliothek



Implementierung des SYCL-Backends

Implementierung des SYCL-Backends
Struktur



Implementierung des SYCL-Backends
Probleme // Events

Events

- Problem: Alpaka nutzt Events zur Synchronisierung und Abhängigkeitsverwaltung
  - `alpaka::wait::wait(event)`
  - `alpaka::wait::wait(queue, event)`
- SYCL verwaltet Abhängigkeiten selbst
  - `cl::sycl::queue` löst Abhängigkeiten auf
  - Abhängigkeit: Pufferverfügbarkeit, nicht Kernel-Ende!
  - `cl::sycl::queue` kann nicht auf `cl::sycl::event` warten

- Problem: Alpaka-Vorbild CUDA
  - Events werden erzeugt und vor/nach Kernel in Queue eingereiht
  - Erreichtes Event zeigt Kernel-Ende an
- SYCL-Queue erzeugt Events selbst
  - Für Profiling gedacht
  - Zeigt Kernel-Ende an, nicht Pufferverfügbarkeit!

Implementierung des SYCL-Backends
Probleme // Zeiger

Zeiger

- Problem: SYCL verbietet Zeiger-Kernel-Parameter
  - `alpaka::mem::view::getPtrNative()` für Kernel-Parameter nicht anwendbar
  - SYCL: `buffer.get_access()` → `accessor.get_pointer()` → `static_cast<T*>()`
  - Lösung: Pseudo-Zeiger + Template-Meta-Programmierung

- Problem: Zeiger verlieren Informationen
  - `buffer.get_access<access::mode::read_write, access::target::global_buffer>()`
  - Informationen werden benötigt (z.B. bei Atomics)

Implementierung des SYCL-Backends
Probleme // FPGA-Erweiterungen

FPGA-Erweiterungen

— In Alpaka integriert?
  – Separate Code-Pfade für FPGAs
  – Separate Code-Pfade für Hersteller

— Als Alpaka-Erweiterung?
  – Z.B. `alpaka::pipeline`
  – Nicht auf GPUs und CPUs anwendbar
  – Schränkt Portabilität ein

Ergebnisse

Ergebnisse
Nutzbarkeit // Übersicht

Implementierung	Nutzbarkeit mit Alpaka
ComputeCpp	Nicht nutzbar (fehlerhaft)
Intel	Nutzbar
Xilinx	Nicht nutzbar (fehlerhaft)
hipSYCL	Nicht nutzbar (unvollständig)
sycl-gtx	Nicht nutzbar (unvollständig)

Ergebnisse
Nutzbarkeit // ComputeCpp

Verwendete Version: ComputeCpp 1.1.5 Community Edition

Problem: Zeiger

```
template <typename T>
void f(T* ptr);

auto x = 42;
f(&x);  // void f(int* ptr);

auto ptr = sycl_accessor.get_pointer();
f(ptr); // void f(__global int* ptr);

std::is_same_v<int*, decltype(ptr)>;  // false
__global int* ptr2 = nullptr;         // Syntaxfehler
```

Ergebnisse
Nutzbarkeit // Xilinx

Verwendete Softwareversionen:

- github.com/triSYCL/sycl, `sycl/unified/next`-Zweig, Commit `#dfb95af`
- SDAccel 2019.1
- XRT 2.2
- `xilinx-u200-xdma` & `xilinx-u200-xdma-dev` für Ubuntu 18.04 (Version `201830.2-2580015`)

Problem: benutzerdefinierte Strukturen

```
struct coord {
    std::size_t x;
    std::size_t y;
};

struct kernel {
    void operator()()
    {
        auto c = coord{42, 42}; // Compiler-Absturz
    }
};
```

Ergebnisse
Performanz // Verifizierung

Verifizierung des Alpaka-Backends

- Alpaka-Programm: *jungfrau-photoncounter*
- Photonenzähler für JUNGFRAU-Detektor (Paul Scherrer Institut, PSI)
  - 16 Megapixel
  - 2 Byte pro Pixel
  - Derzeitige Frequenz: 100 Hz (3,2 GB/s)
  - Zielfrequenz: 2,2 kHz (74 GB/s)

$$N_\gamma = \frac{\text{ADC} - \text{Sockel}}{\text{Verstärkung} \cdot E_\gamma}$$

- N_γ: Anzahl der Photonen
- ADC: Messergebnis des Pixels
- Sockel: Grundrauschen des Pixels (engl. *pedestal*)
- Verstärkung: Signalverstärkung des Pixels (engl. *gain*)
- E_γ: Photonenenergie

Ergebnisse
Performanz // Verifizierung

Verwendete Software:

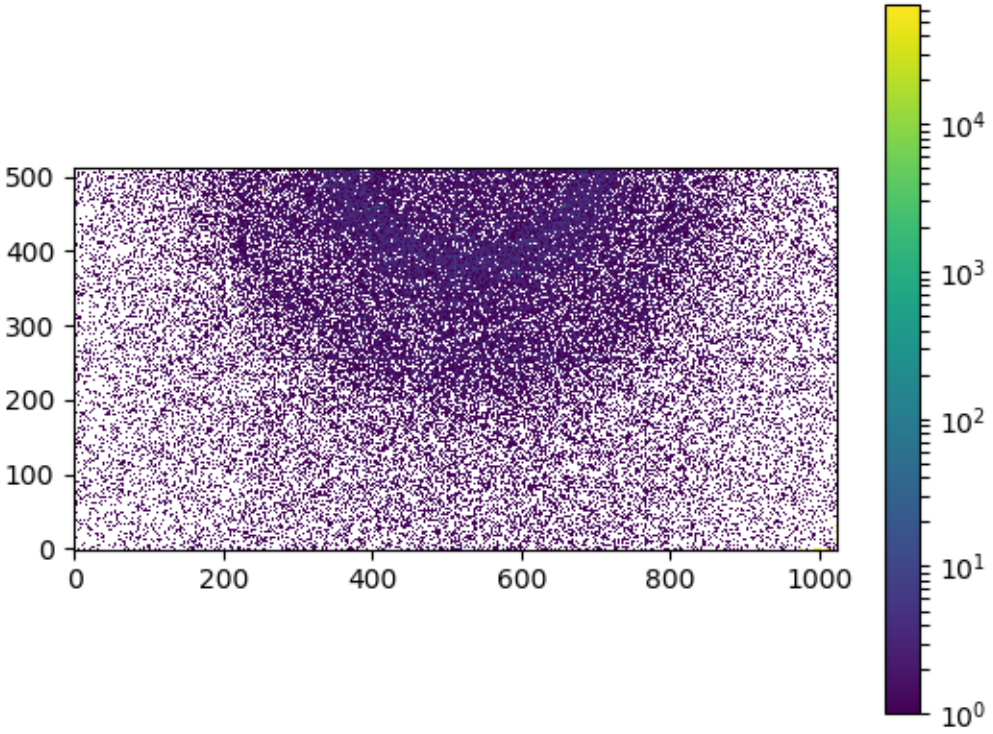
- Intel-SYCL-Implementierung: github.com/intel/llvm, `sycl`-Zweig, Commit 78d9957
- *Intel Graphics Compute Runtime for OpenCL*, Version 19.46.14807
- Ubuntu 19.10

Verwendete Hardware:

- Intel HD Graphics 520 (Skylake GT2)
- 6 GiB Speicher
- 100,8 GFLOPS (doppelte Präzision)
- Host: Intel Core i7-6500U (3,1 GHz), 8 GiB Speicher

Messung:

- Datensatz px_101016, auf 1000 Messungen reduziert
- Zeitbedarf: 48,372 s (ohne Laden, Kalibrierung etc.) → 20,6731 Messungen pro Sekunde



Ergebnisse
Performanz // Beispielalgorithmus FPGA

Boxfilter [NF17]

$$p'(x,y) = \frac{1}{9} \cdot \sum_{j=-1}^{1} \sum_{i=-1}^{1} p(x+i, y+j)$$

— Pixel außerhalb des Bildes → 0

Verwendete Software:

— Xilinx-Implementierung: github.com/triSYCL/sycl, `sycl/unified/next`-Zweig, Commit #`dfb95af`
— SDAccel 2019.1
— XRT 2.2
— `xilinx-u200-xdma` & `xilinx-u200-xdma-dev` für Ubuntu 18.04 (Version `201830.2-2580015`)

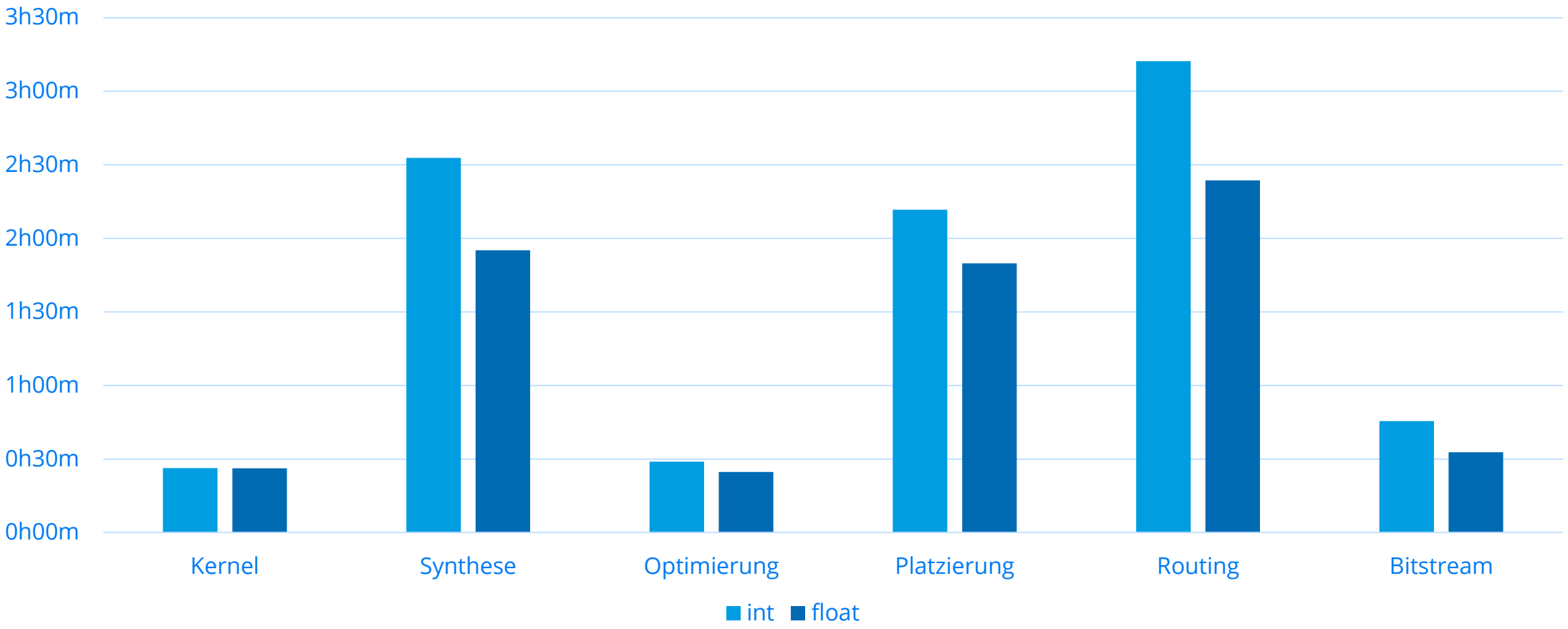
Verwendete Hardware:

— Xilinx Alveo U200 (Datacenter-FPGA, *Hemera*-Knoten h002)

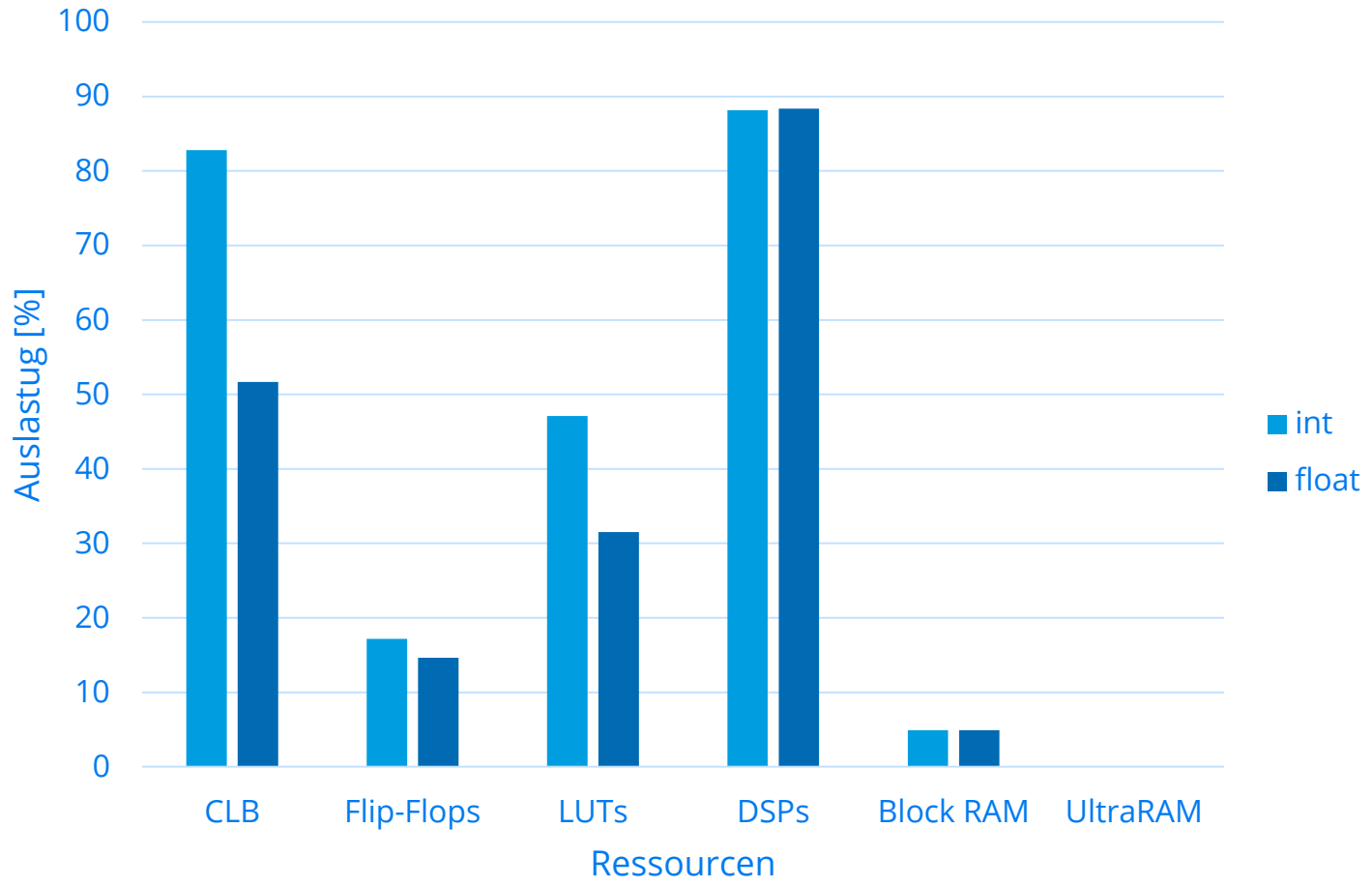
Methodik:

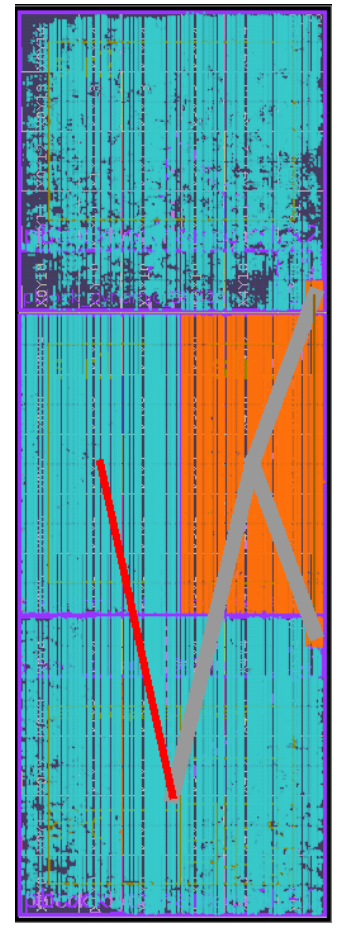
— 512 Bilder, 512 x 256 Pixel, `int` und `float`

Ergebnisse
Performanz // Compile-Zeit

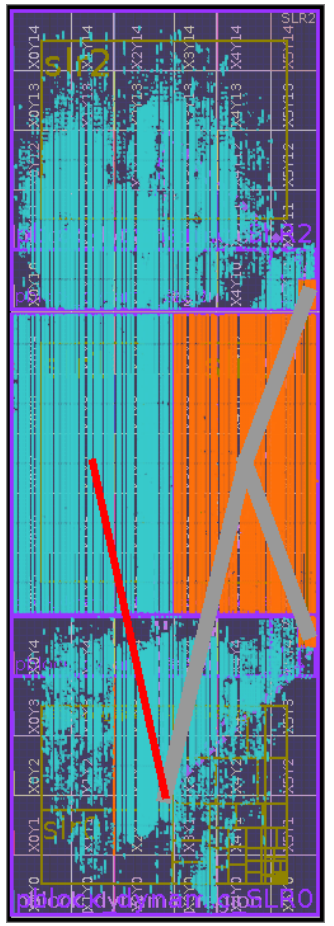


Ergebnisse
Performanz // Ressourcenverbrauch





int



float

Fazit

Fazit

- SYCL als Alpaka-Backend grundsätzlich geeignet
  - Konzeptionelle Unterschiede vorhanden
  - Entwicklung über Prototyp-Status hinaus nicht durchführbar
- Entwicklungsstand der SYCL-Implementierungen erschwert Alpaka-Nutzung
  - Nur Intel-Implementierung geeignet
  - Höherer Reifegrad der Implementierungen notwendig
- Alpaka sehr CUDA-nah
  - Abkehr von Zeigern sinnvoll
  - Höherer Abstraktionsgrad?
  - Erweiterungen?
- SYCL für FPGAs sehr attraktiv
  - Intuitive, moderne C++-Schnittstelle
  - Wichtiges Alpaka-Backend bei Weiterentwicklung der SYCL-Implementierungen

Vielen Dank!

Literatur

Literatur

[Ama] Amazon Web Services, Inc. *Amazon EC2 F1-Instances*. URL: https://aws.amazon.com/de/ec2/instance-types/f1/ (besucht am 22.11.2019)

[Chu+18] Eric Chung u.a. „Serving DNNs in Real Time at Datacenter Scale with Project Brainwave“. In: *IEEE Micro* Jahrgang 38. Ausgabe 2 (März 2018), S. 8 – 20. DOI: 10.1109/MM.2018.022071131

[Di+17] Lorenzo Di Tucci u.a. „The Role of CAD Frameworks in Heterogeneous FPGA-Based Cloud Systems“. In: IEEE 35th International Conference on Computer Design. Nov. 2017, S. 423 – 426. DOI: 10.1109/ICCD.2017.75

[Fir+18] Daniel Firestone u.a. „Azure Accelerated Networking: SmartNICs in the Public Cloud“. In: *15th USENIX Symposium on Networked Systems Design and Implementation*. Apr. 2018, S. 51 – 64.

[Fow+18] Jeremy Fowers u.a. „A Configurable Cloud-Scale DNN Processor for Real-Time AI“. In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. Juni 2018, S. 1 – 14. DOI: 10.1109/ISCA.2018.00012

[HS10] Charles Hawkins und Jaume Segura. *Introduction to Modern Digital Electronics.* Preliminary Edition. SciTech Publishing, Inc., 2010. ISBN: 978-1-891-12107-4

Literatur

[KRH19] Ronan Keryell, Maria Rovatsou und Lee Howes, Hrsg. *SYCL™ Specification*. 9450 SW Gemini Drive #45043, Beaverton, OR 97008-6018, Vereinigte Staaten von Amerika, April 2019.

[Law+79] Charles L. Lawson u.a. „Basic Linear Algebra Subprograms for Fortran Usage“. In: *ACM Transactions on Mathematical Software* Jahrgang 5. Ausgabe 3 (September 1979), S. 308 – 323. DOI: 10.1145/355841.355847

[NF17] Masahiro Nakamura und Norishige Fukushima. „Fast Implementation of Box Filtering“. In: *Proceedings of the International Workshop on Advanced Image Technology (IWAIT)*. Jan. 2017

[Wor15] Benjamin Worpitz. „Investigating performance portability of a highly scalable particle-in-cell simulation code on various multi-core architectures.“ Masterarbeit. Fakultät Informatik, Helmholtzstraße 10, 01069 Dresden: Technische Universität Dresden, Okt. 2015. DOI: 10.5281/zenodo.49768

[Xil19a] Xilinx, Inc. *Alveo U200 and U250 Data Center Accelerator Cards Data Sheet*. DS962 (v1.1). Xilinx, Inc. 2100 Logic Drive, San Jose, CA 95124, Vereinigte Staaten von Amerika, Juni 2019.

FPGAs als Beschleuniger
Programmierung // VHDL

Hardware-Modellierung mit VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY reg2 IS
    PORT(
        clk  : IN  std_logic;
        d0   : IN  std_logic;
        d1   : IN  std_logic;
        load : IN  std_logic;
        res  : IN  std_logic;
        q0   : OUT std_logic;
        q1   : OUT std_logic;
    );
END reg2;
```

```
ARCHITECTURE beh OF reg2 IS
    SIGNAL q0_s, q0_ns, q1_s, q1_ns : std_logic;
BEGIN
    reg: PROCESS (clk, res)
    BEGIN
        IF res = '1' THEN
            q0_s <= '0';
            q1_s <= '0';
        ELSIF clk'event AND clk = '1' THEN
            q0_s <= q0_ns;
            q1_s <= q1_ns;
        END IF;
    END PROCESS reg;

    q0 <= q0_s AFTER 2 ns;
    q1 <= q1_s AFTER 2 ns;

    mux: PROCESS (load, q0_s, q1_s, d0, d1)
    BEGIN
        IF load = '1' THEN
            q0_ns <= d0 AFTER 3 ns;
            q1_ns <= d1 AFTER 3 ns;
        ELSE
            q0_ns <= q0_s AFTER 4 ns;
            q1_ns <= q1_s AFTER 4 ns;
        END IF;
    END PROCESS mux;
END beh;
```