

# Profilmodul Forschungsprojekt Grundlagen INF-PM-FPG

Innovative Spracherweiterungen für Beschleunigerkarten am  
Beispiel von SYCL, HC, HIP und CUDA: Untersuchung zu Nutzbarkeit  
und Performance

Jan Stephan

Betreuender Hochschullehrer: Prof. Dr. Wolfgang E. Nagel

Betreuer: Dr.-Ing. Bernd Trenkler

Matthias Werner, M.Sc.

18. März 2019

- Einleitung
  - Motivation
  - Ziel
- Funktionaler Vergleich
  - Anforderungen
  - CUDA
  - HIP
  - HC
  - SYCL
- Benchmark-Ergebnisse
  - NVIDIA-GPUs
  - AMD-GPUs
  - Übergreifender Vergleich
- Fazit & Ausblick

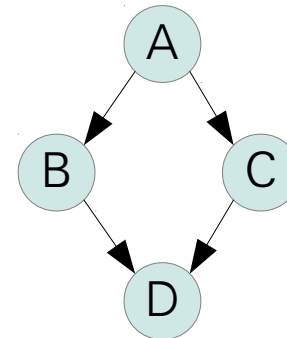
- NVIDIA CUDA ist dominante GPGPU-Plattform
  - TOP500 (November 2018): fünf der ersten zehn Plätze
  - TOP500 (November 2018): insgesamt 126 Mal
- Wenig Konkurrenz
  - TOP500 (November 2018): 30 Mal Intel Xeon Phi, kein AMD
  - verfügbare OpenCL-Unterstützung veraltet (OpenCL 1.2 von 2011)
- Neue Technologien:
  - AMD: HC & HIP als Teil von ROCm (*Radeon Open Compute*, 2016)
  - Khronos: SYCL (2014)

- Analyse und Vergleich der Programmiermodelle
- Performance-Benchmarks auf AMD- und NVIDIA-GPUs
- Empfehlungen für den zukünftigen Einsatz

---

# Funktionaler Vergleich

- Datensicht (nach M. Wong [1])
  - Datenbewegung
    - Explizit oder implizit?
  - Datenanordnung
    - Gekapselt?
  - Datenaffinität
    - Zuordnung zu bestimmter GPU
  - Datenlokalität
    - Nutzung der Speicherhierarchie
- Aufgabensicht
  - Aufgabengraphen



# CUDA - Einführung

---

- Single-source
- API: C
- Kernelsprache: C++-Dialekt
- Ausführung auf NVIDIA-GPUs möglich

```
__global__ void kernel(const float* a, const float* b, float* c,  
                      std::size_t dim)  
{  
    /* ... */  
}
```

- *Threads* sind kleinste Ausführungseinheit
- *Warps* bestehen aus 32 Threads, die synchron ausgeführt werden
  - Kommunikation über Intrinsiken
- *Blocks* bestehen aus bis zu 1.024 Threads
  - Ausführung auf einem Multiprozessor
  - Kommunikation über *shared memory*
  - Synchronisation über Barrieren
- *Grid* besteht aus allen Blocks
  - Kommunikation über *global memory*



- Explizite Datenbewegung

```
auto host_buf = new float[num_elems];  
auto dev_buf = static_cast<float*>(nullptr);  
cudaMalloc(&dev_buf, num_elems * sizeof(float));  
cudaMemcpy(dev_buf, host_buf, num_elems * sizeof(float),  
            cudaMemcpyDeviceToHost);
```

- Implizite Datenbewegung (CUDA 6)

```
auto buf = static_cast<float*>(nullptr);  
cudaMallocManaged(&buf, num_elems * sizeof(float));
```

- Implizite Datenbewegung (CUDA 8, Pascal)

```
auto buf = new float[num_elems];
```

- GPU-Speicheranordnung entspricht CPU-Speicheranordnung
  - Programmierer muss effiziente Anordnung / Zugriffe sicherstellen
- Ausnahmen
  - zwei-/dreidimensionale Arrays (`cudaMallocPitch` / `cudaMalloc3D`)
  - Textur-Caches (`cudaMallocArray`)

- Explizite Bewegung: Genau eine aktive GPU
- GPU-Wechsel erfolgt manuell (`cudaSetDevice`)

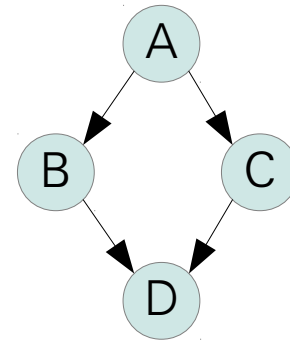
```
cudaSetDevice(0);  
cudaMemcpy(..., cudaMemcpyHostToDevice); // Ziel: GPU #0  
cudaSetDevice(1);  
cudaMemcpy(..., cudaMemcpyHostToDevice); // Ziel: GPU #1
```

- Implizite Bewegung (vor Kepler): wie explizite Bewegung
- Implizite Bewegung (ab Kepler): Speicher auf allen GPUs sichtbar

- Mehrere Ebenen der Speicherhierarchie
  - *Global memory*: Hauptspeicher der GPU, für alle Multiprozessoren sichtbar
    - Zugriff über L1- und L2-Caches
  - *Constant memory*: Hauptspeicher, spezieller Cache
  - Textur-Caches
  - *Shared memory*: Programmierbarer L1-Cache
  - Register
  - *Local memory*: „versteckter“ Hauptspeicher

- Herkömmlich: CUDA Streams und Events

```
A<<<..., stream1>>>();  
cudaEventRecord(eventA, stream1);  
  
B<<<..., stream1>>>();  
  
cudaStreamWaitEvent(stream2, eventA);  
C<<<..., stream2>>>();  
cudaEventRecord(eventC, stream2);  
  
cudaStreamWaitEvent(stream1, eventC);  
D<<<..., stream1>>>();
```



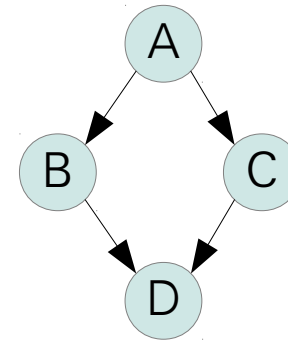
## ● Neu (CUDA 10): CUDA Graphs

```
cudaGraphCreate(&graph);
```

```
cudaGraphAddNode(graph, A, {}, ...);  
cudaGraphAddNode(graph, B, {A}, ...);  
cudaGraphAddNode(graph, C, {A}, ...);  
cudaGraphAddNode(graph, D, {B, C}, ...);
```

```
cudaGraphInstantiate(&instance, graph);
```

```
cudaGraphLaunch(instance, stream);
```



- *Heterogeneous-Computing Interface for Portability*
- CUDA-Klon
  - API: `cudaCmd` wird zu `hipCmd`
  - Kernelsprache weitestgehend identisch
- Ausführung auf AMD- und NVIDIA-GPUs möglich

- CUDAs Konzepte gelten grundsätzlich auch hier
- Keine implizite Datenbewegung (`hipMallocManaged`)
- Keine optimierte Datenanordnung (`hipMallocPitch` / `hipArrays`)
- Eingeschränkter Textur-Cache
- Keine HIP Graphs



- *Heterogeneous Compute*
- Basiert auf Microsofts C++AMP
- Single-source
- API: C++
- Kernelsprache: C++-Dialekt
- Ausführung auf AMD-GPUs möglich

```
void kernel(hc::tiled_index<1> idx,  
            hc::array_view<const float> a,  
            hc::array_view<const float> b,  
            hc::array_view<float> c, std::size_t dim) [[hc]]  
{  
    /* ... */  
}
```

- *Threads* sind kleinste Ausführungseinheit
- *Wavefronts* bestehen aus 64 Threads, die synchron ausgeführt werden
  - Kommunikation über Intrinsiken
- *Tiles* bestehen aus bis zu 1.024 Threads
  - Ausführung auf einem Multiprozessor
  - Kommunikation über *tile static memory*
  - Synchronisation über Barrieren
- Keine Entsprechung zu CUDA's *Grid*
  - Anzahl der Tiles wird durch globale Problemgröße bestimmt

## ● Explizite Datenbewegung

```
auto host_buf = std::vector<float>{};  
host_buf.resize(num_elems);
```

```
auto dev_buf = hc::array<float, 1>{hc::extent<1>{num_elems}};  
hc::copy(std::begin(host_buf), dev_buf);
```

## ● Implizite Datenbewegung (hc::array)

```
auto host_buf = std::vector<float>{};  
host_buf.resize(num_elems);
```

```
auto dev_buf = hc::array<float, 1>{hc::extent<1>{num_elems},  
                                std::begin(host_buf)};
```

## ● Implizite Datenbewegung (hc::array\_view)

```
auto host_buf = std::vector<float>{};  
host_buf.resize(num_elems);
```

```
auto view = hc::array_view<float, 1>{hc::extent<1>{num_elems},  
                                host_buf};
```

- GPU-Speicheranordnung vor Programmierer verborgen
  - `hc::array` und `hc::array_view` können bis zu drei Dimensionen abdecken
  - Zugriff auf Elemente über spezielle Datenstruktur `hc::index`
  - API unterscheidet zwischen Zeiger und linearisiertem Zeiger

- Explizite Bewegung: `hc::array` ist immer an bestimmte GPU gebunden
  - Bei fehlender Angabe entscheidet die Laufzeitumgebung

```
auto gpu0 = ...;  
auto gpu1 = ...;
```

```
auto gpu0_view = gpu0.get_default_view();  
auto gpu1_view = gpu1.get_default_view();
```

```
auto a = hc::array<float, 1>{..., gpu0_view};  
auto b = hc::array<float, 1>{..., gpu1_view};
```

```
hc::copy(a, b); // Quelle: GPU #0, Ziel: GPU #1
```

- Implizite Bewegung (`array_view`): keine GPU-Zuordnung
  - Kopien zwischen GPUs grundsätzlich versteckt

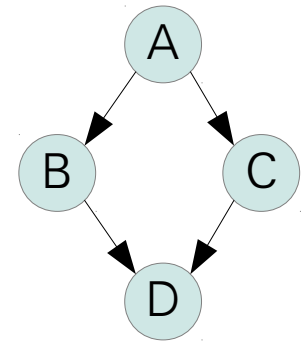
- Zwei Ebenen der Speicherhierarchie
  - *Global memory*: Hauptspeicher der GPU, für alle Multiprozessoren sichtbar
  - *Tile static memory*: Programmierbarer Multiprozessor-Cache

- hc::accelerator\_view ist *logische* Sicht auf GPU
  - Mehrere parallele accelerator\_views pro GPU möglich
  - Synchronisation zwischen verschiedenen accelerator\_views über completion\_future-Objekte
  - Entsprechen std::future, erweitert um then-Methode
  - Nachteil: then nur einmal ausführbar

```
auto futureA = hc::parallel_for_each(view0, /* A */);  
auto futureB = hc::parallel_for_each(view0, /* B */);
```

```
auto futureC = hc::completion_future{};  
futureA.then([&]() {  
    futureC = hc::parallel_for_each(view1, /* C */);  
});
```

```
futureC.then([&]() {  
    hc::parallel_for_each(view0, /* D */);  
})
```



# SYCL – Einführung

---

- Ursprünglich Compiler-Erweiterung für PlayStation3-Systeme
- Single-source; API und Kernelsprache kompatibel zum C++-Standard
  - Kernelsprache ohne dynamische Polymorphie
- Hardware-Unterstützung abhängig von konkreter Implementierung
  - ComputeCpp: Automotive, Embedded, Intel-CPUs, Intel-GPUs, NVIDIA-GPUs (experimentell)
  - triSYCL: CPUs, Xilinx-FPGAs (experimentell)
  - Intel-Prototyp: Intel-CPU, Intel-GPUs

```
struct kernel {  
    cl::sycl::accessor<float, 1, cl::sycl::access::mode::read> a;  
    cl::sycl::accessor<float, 1, cl::sycl::access::mode::read> b;  
    cl::sycl::accessor<float, 1,  
                        cl::sycl::access::mode::discard_write> c;  
    std::size_t dim;  
  
    void operator()(cl::sycl::nd_item<1> my_item) {  
        /* ... */  
    }  
}
```



- *Work-items* sind kleinste Ausführungseinheit
- Keine Entsprechung zu *Warps / Wavefronts*
- *Work-groups* bestehen aus mehreren *Work-items*
  - Kommunikation über *local memory*
  - Synchronisation über Barrieren
- Keine Entsprechung zu CUDA's *Grid*
  - Anzahl der *Work-groups* wird durch globale Problemgröße bestimmt

## ● Explizite Datenbewegung

```
auto host_buf = std::vector<float>{};  
host_buf.resize(num_elems);
```

```
auto dev_buf = cl::sycl::buffer<float>{cl::sycl::range<1>{num_elems}};  
queue.submit([&](cl::sycl::handler& cgh) {  
    auto acc = dev_buf.get_access<cl::sycl::access::mode::discard_write>();  
  
    cgh.copy(host_buf.data(), acc);  
});
```

## ● Implizite Datenbewegung

```
auto host_buf = std::vector<float>{};  
host_buf.resize(num_elems);
```

```
auto dev_buf = cl::sycl::buffer<float>{host_buf.data(),  
                                         cl::sycl::range<1>{num_elems}};
```

- GPU-Speicheranordnung vor Programmierer verborgen
  - Zeigerklassen sind *implementation defined*
  - SYCL umfasst neben `cl::sycl::buffer` auch `cl::sycl::image`

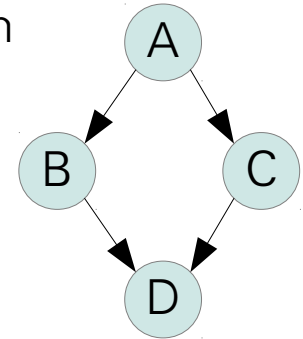
- `cl::sycl::buffer` und `cl::sycl::image` sind auf allen vorhandenen Devices sichtbar
- Objekte kapseln unter Umständen mehrere Speicherreservierungen
- Abhängigkeiten werden von Laufzeitumgebung erkannt und synchronisiert
- Programmierer kann atomare Zugriffe angeben

## ● Vier Ebenen der Speicherhierarchie

- *Global memory*: Hauptspeicher des Devices, für alle *work-items* sichtbar
- *Constant memory*: nur lesbarer Speicher, für alle *work-items* sichtbar
- *Local memory*: work-group-lokaler Speicher, sichtbar für *work-items* einer *work-group*
- *Private memory*: Speicher eines einzelnen *work-items*

- Inhärentes Prinzip des SYCL-Standards
- Alle Kernel werden grundsätzlich asynchron ausgeführt
- SYCL-Laufzeitumgebung erkennt und serialisiert Abhängigkeiten

```
queue.submit(/* A */);  
queue.submit(/* B */);  
queue.submit(/* C */);  
queue.submit(/* D */);
```



# Zusammenfassung

	CUDA	HIP	HC	SYCL
Datenbewegung	explizit, implizit	<b>Explizit</b>	explizit, implizit	explizit, implizit
1D-Anordnung	Zeiger	Zeiger	gekapselt	gekapselt
2D-Anordnung	Zeiger, optimiert	<b>fehlt</b>	gekapselt	gekapselt
3D-Anordnung	Zeiger, optimiert	Zeiger, optimiert	gekapselt	gekapselt
Tex.-Anordnung	gekapselt	gekapselt	<b>fehlt</b>	gekapselt
Datenaffinität	<b>explizit, implizit</b>	explizit	implizit	implizit
Datenlokalität	globaler Speicher, lokaler Speicher, <b>programmierbare Caches</b>	globaler Speicher, lokaler Speicher, <b>programmierbare Caches</b>	globaler Speicher, lokaler Speicher	globaler Speicher, lokaler Speicher
Aufgabengraph	Asynchronität, <b>Graph-API</b>	Asynchronität	<b>Asynchronität (eingeschränkt)</b>	<b>automatisch</b>

---

# Benchmark-Ergebnisse





- NVIDIA: Taurus-gpu2-Partition
  - SCS5-Umgebung
  - Tesla K20x (ECC aktiviert)
  - CUDA/10.0.130
  - GCC/7.3.0-2.30
  - HIP: Version 1.5.19061, ROCm-GitHub-Repository
  - SYCL: ComputeCpp 1.0.5 für Ubuntu 14.04

- Compiler-Flags

```
nvcc -std=c++14 -O3 -gencode arch=compute_35,code=sm_35
```

```
hipcc -std=c++14 -O3 -gencode arch=compute_35,code=sm_35
```

```
compute++ -std=c++17 -O3 -sycl-driver -sycl-target ptx64
```

# Verwendete Hard- und Software (AMD)

---

- AMD: separater Rechner
  - CPU: AMD Ryzen Threadripper 1950X
  - GPU: AMD Radeon RX Vega 64
    - 64 Multiprozessoren
    - 64 Kerne pro Multiprozessor (insgesamt 4.096)
    - 1.536 MHz Maximaltakt
    - 8 GiB HBM2-Speicher
    - Speicherbusbreite: 2.048 bit
    - Speicherbandbreite: 483,3 GiB/s
    - Speichertakt: 945 MHz
    - kein ECC
- Compiler-Flags und GPU-Einstellungen (ROCm 2.1.96)

```
hcc `hcc-config --cxxflags --ldflags` -O3 std=c++17 -amdgpu-target=gfx900
```

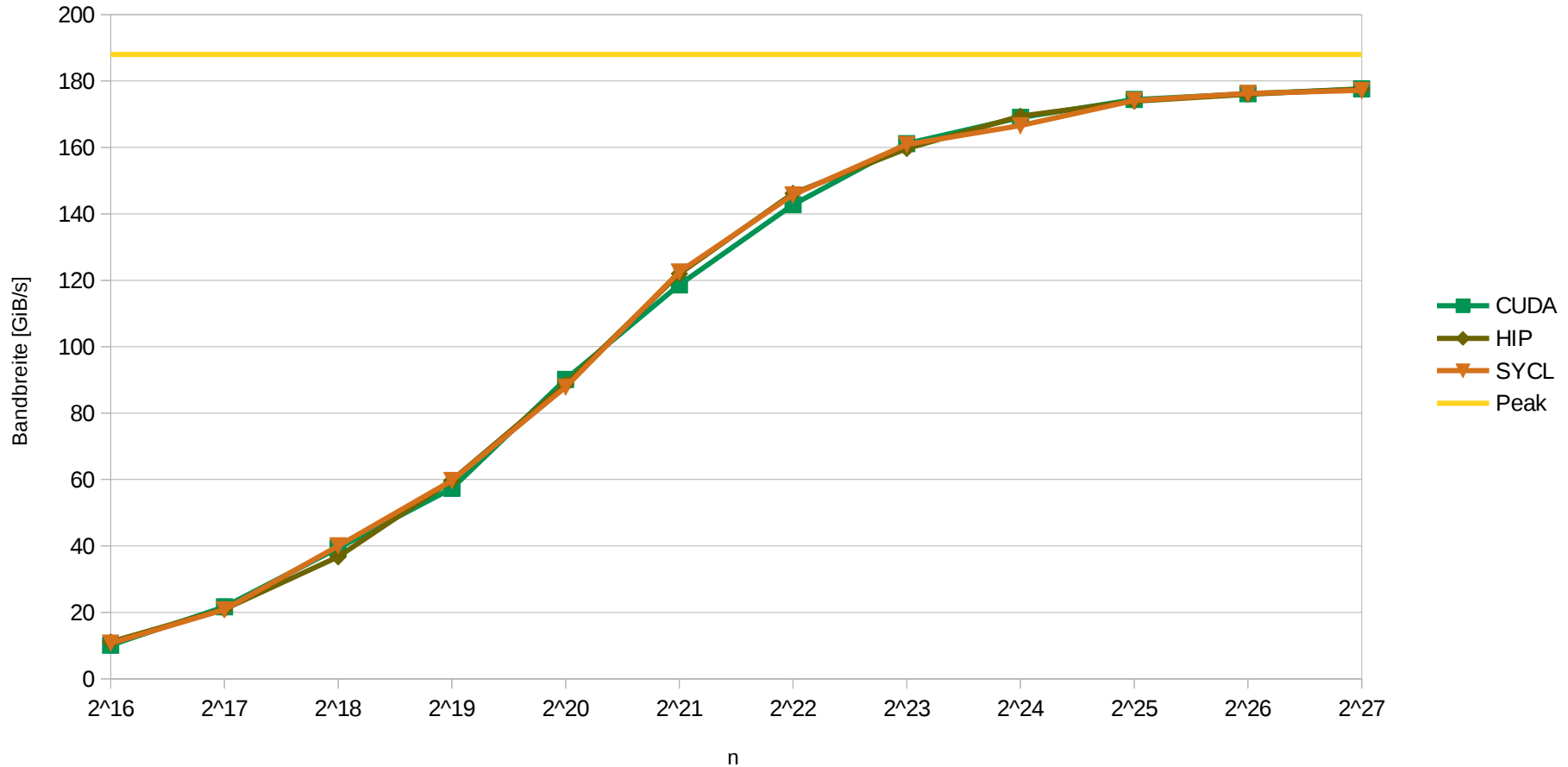
```
hipcc -O3 -std=c++17
```

```
rocm-smi --setsclk 7
```

- Reduction: memory-bound
  - Additionsreduktion (ganze Zahlen)
  - 1. Stufe: *grid-stride loop*
    - $n$  Elemente werden auf  $x$  Blöcke mit jeweils  $p$  Threads verteilt
    - Jeder Thread führt Reduktion  $y = n/p$ -Mal aus
    - Bedingungen:  $n$  ist Vielfaches von  $p$  und  $y$  Vielfaches von  $x$
  - 2. Stufe: blockweise Reduktion
    - Threads laden Ergebnis der ersten Stufe in lokalen Speicher
    - Hälfte der Threads führt Reduktion auf jeweils einem Elementepaar aus usw.
    - Ziel: ein Ergebnis pro Block
  - Erneuter Aufruf des Kernels mit  $x/2$  Threads berechnet Gesamtergebnis

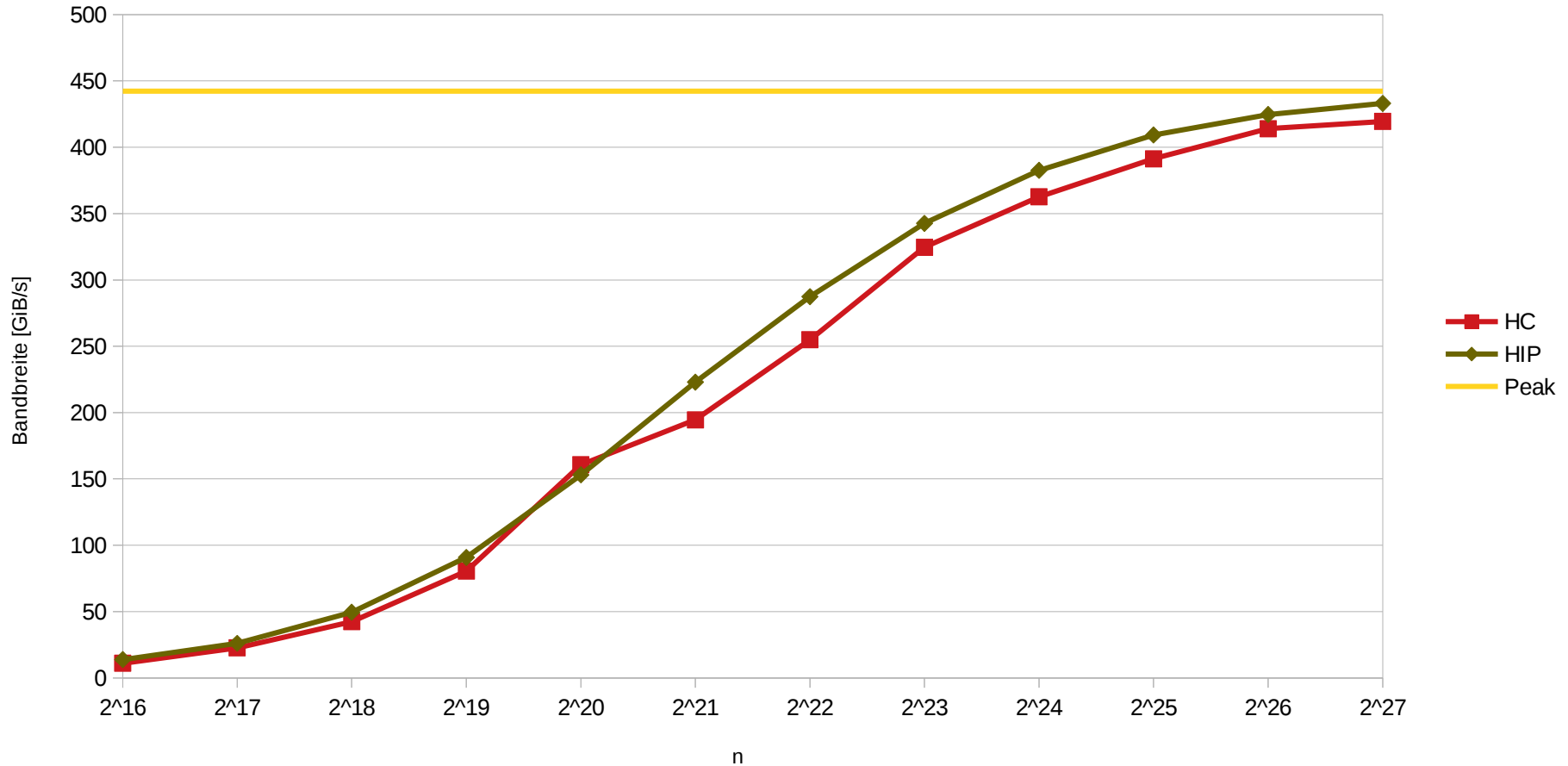
# Reduction-Benchmark: NVIDIA

K20x - Reduction - 32 bit Integer - acht 256er-Blöcke pro Multiprozessor



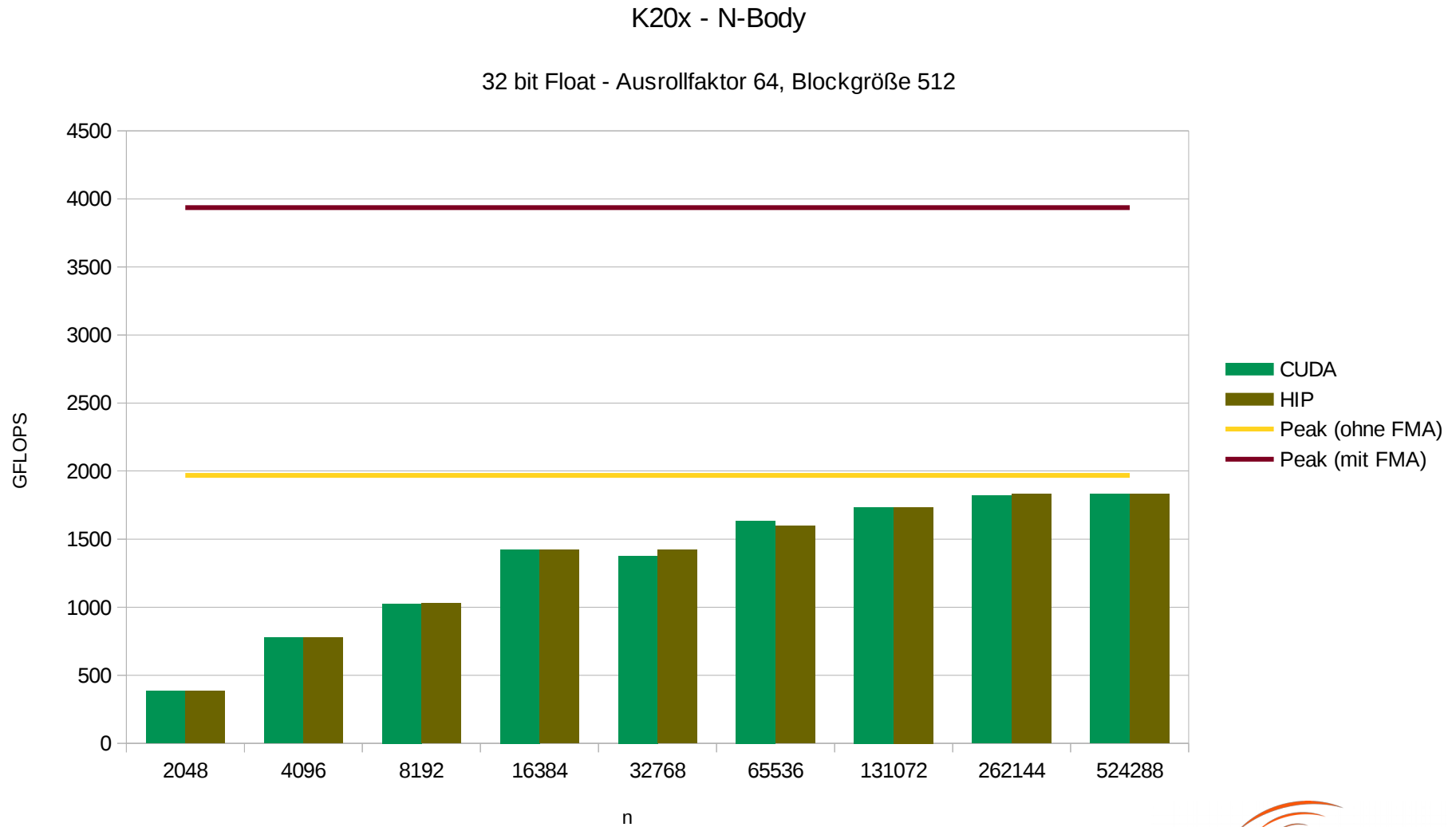
# Reduction-Benchmark: AMD

AMD - Reduction - zwei 256er-Tiles pro Multiprozessor

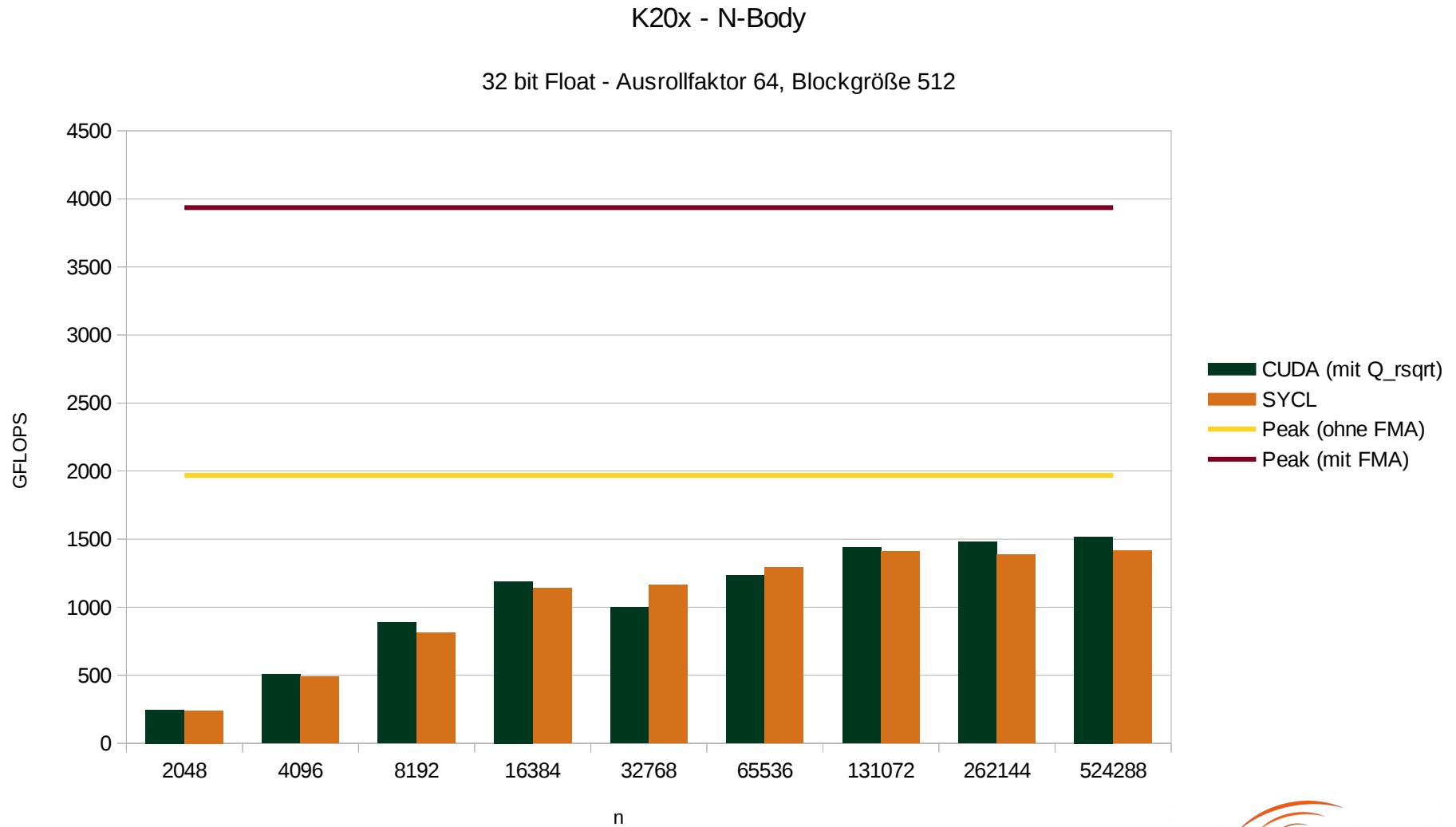


- N-Body: compute-bound
  - $O(n^2)$ -Komplexität
  - 20 FLOPs pro Interaktion
  - Zehn Zeitschritte
  - Implementierung nach Nyland et al. [2]
- Separater Vergleich zwischen CUDA und SYCL erforderlich
  - ComputeCpps Implementierung unterstützt zur Zeit die `rsqrt`-Funktion nicht
  - `Q_rsqrt` aus *Quake 3 Arena* für diesen Benchmark übernommen

# Reduction-Benchmark: NVIDIA

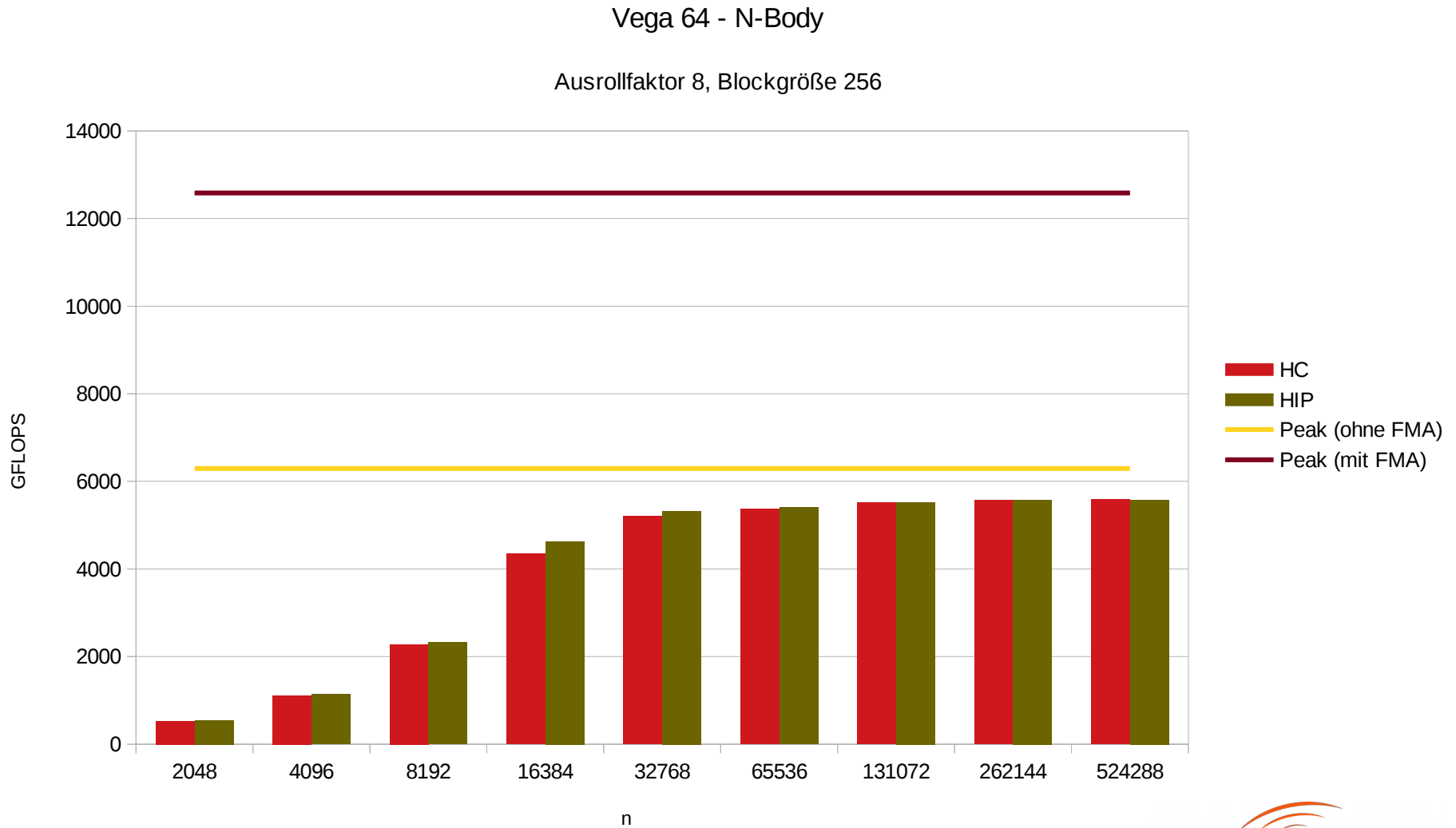


# Reduction-Benchmark: NVIDIA

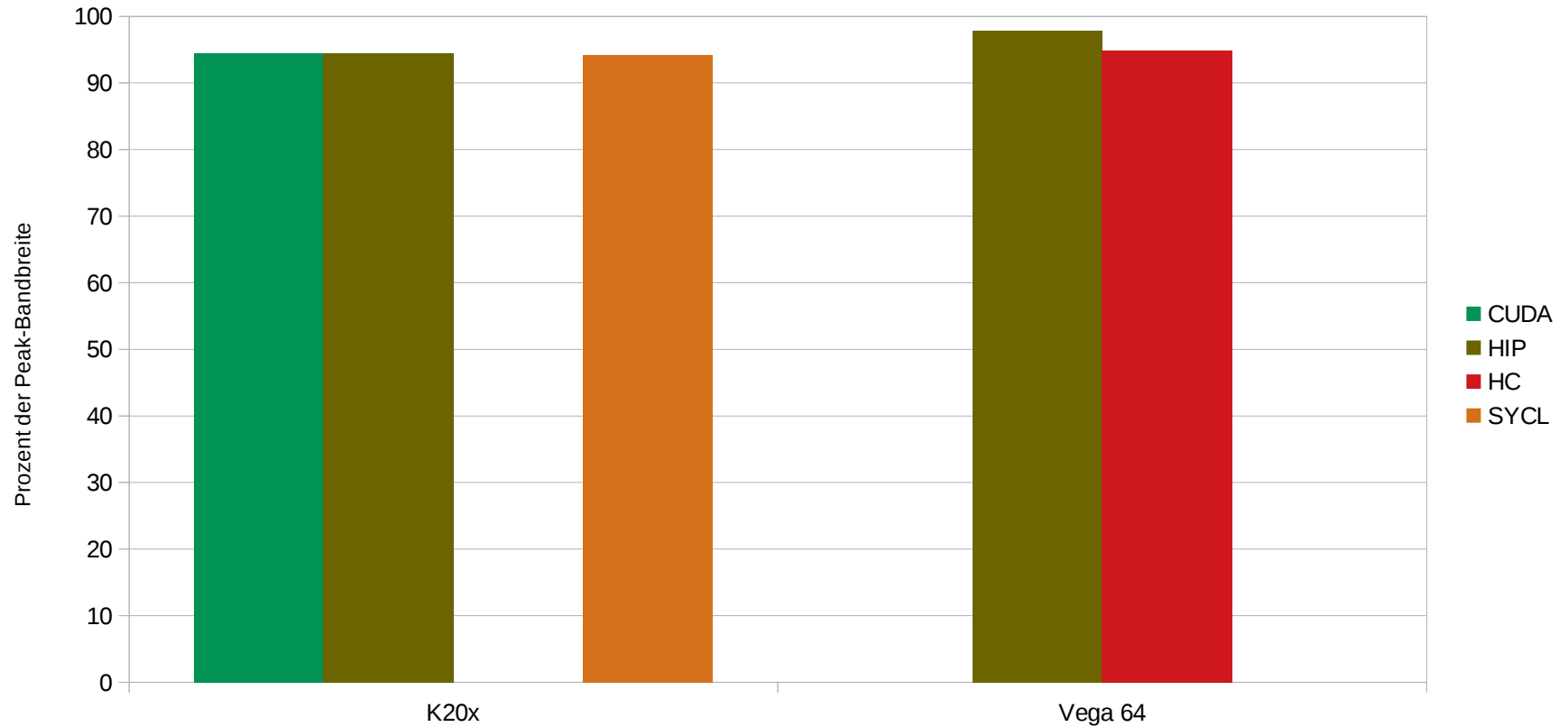




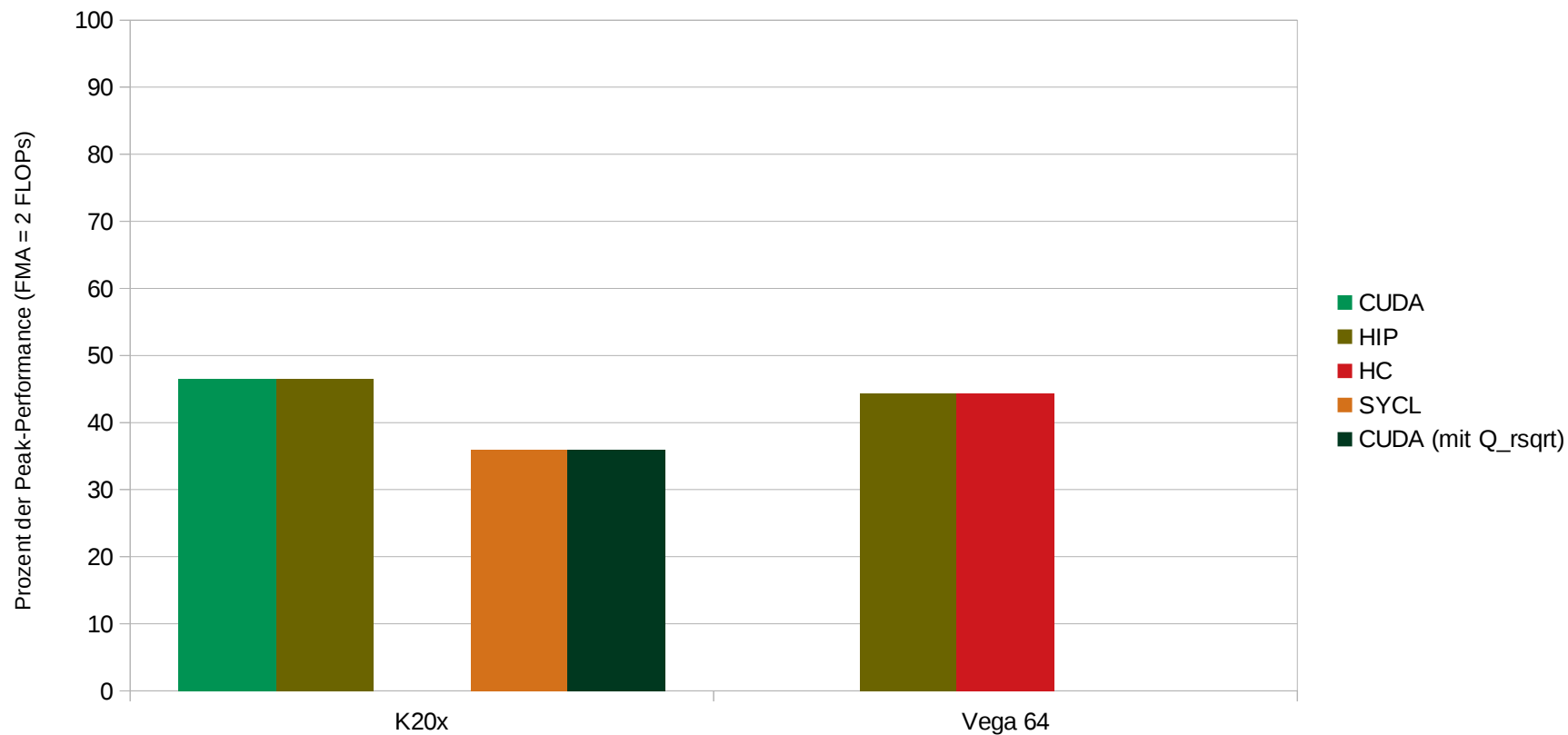
# Reduction-Benchmark: NVIDIA



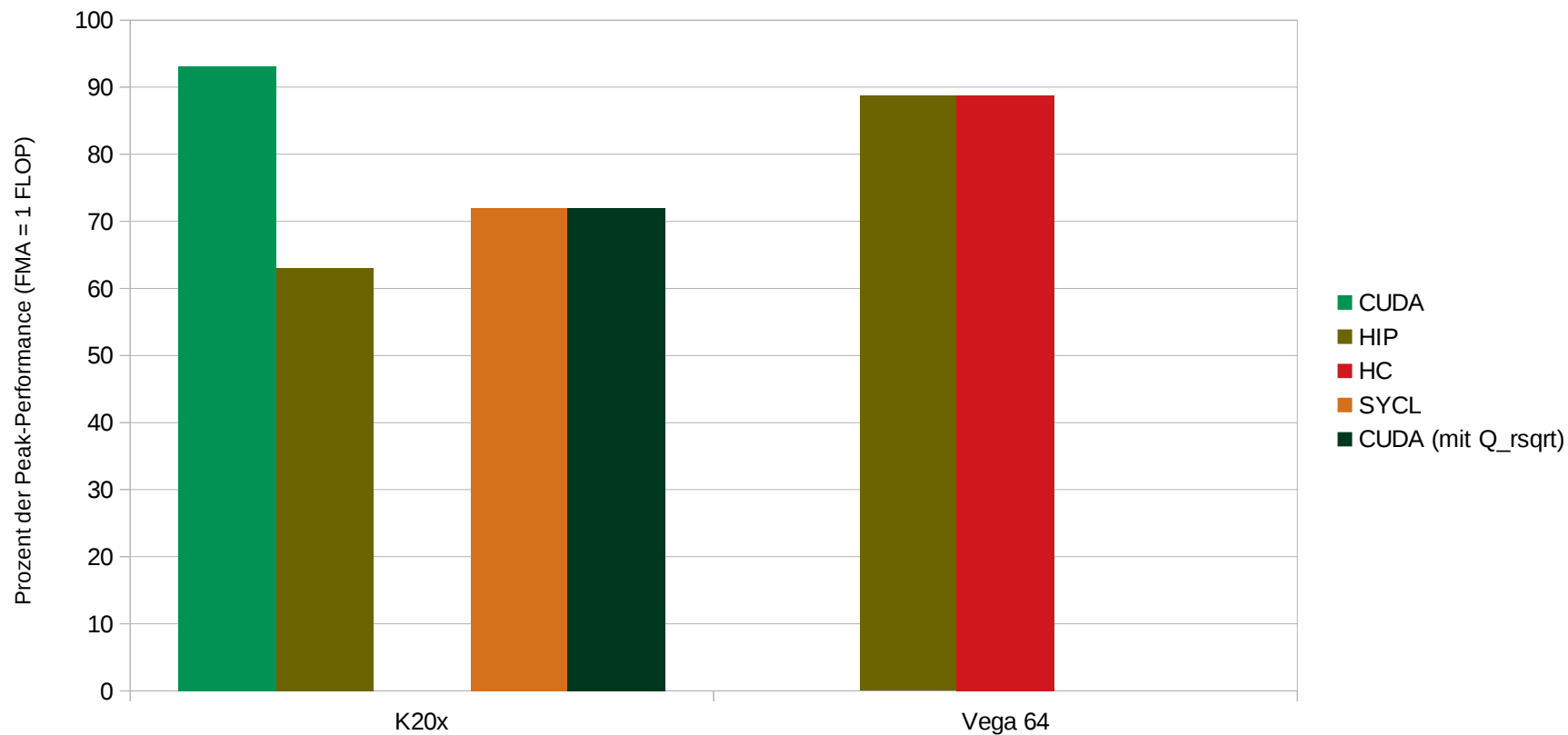
# Leistungsvergleich – Bandbreite



# Leistungsvergleich – FLOPS



# Leistungsvergleich – FLOPS



---

# Fazit und Ausblick

- Spracherweiterungen verfügen über ähnlichen Feature-Umfang
  - Nachteil HC: beschränkte Graph-Fähigkeiten
- Auf derselben Hardware gibt es nahezu keine Performance-Unterschiede
  - AMD: HC generell etwas schlechter als HIP
- Plattformübergreifend ebenfalls geringe Performance-Unterschiede

- CUDA nahezu alternativlos
  - Ausgereiftes und umfangreiches Ökosystem
  - Zur Zeit die beste Wahl
- SYCL ist vielversprechender Wettbewerber
  - Voraussetzung: bessere Hard- und Software-Unterstützung
  - Sollte weiter beobachtet werden
- HC bietet gegenüber HIP lediglich ein modernes C++-Interface
  - HIP besser portierbar
- HIP als Ersatz für CUDA möglich
  - Besser portierbar
  - Es fehlen weiterhin wichtige Features

- Untersuchung des Multi-GPU-Verhaltens
  - Peer-to-Peer
  - Remote direct memory access
- Vergleich der vorhandenen Ökosysteme
- Analyse von SYCLs Performance-Portabilität
  - Lauffähig auf CPUs, GPUs, FPGAs
- Ausgang des Rechtsstreits Oracle vs. Google
  - Darf man APIs klonen?



---

# Vielen Dank!

- [1] Wong, Michael: *The Future Direction of C++ and the Four Horsemen of Heterogeneous Computing*. Vortrag, November 2018. [https://www.youtube.com/watch?v=7Y3-pV\\_b-1U](https://www.youtube.com/watch?v=7Y3-pV_b-1U), zuletzt abgerufen am 25. Januar 2019
- [2] Nyland, Lars; Harris, Mark; Prins, Jan: *Fast N-Body Simulation with CUDA*. In: Nguyen, Hubert (Hrsg.): *GPU Gems 3*. Erste Auflage. Addison-Wesley, August 2007, Kapitel 31, S. 677-696
- ComputeCpp CE – Overview*. 2019. – <https://developer.codeplay.com/computecppce/latest/overview>, zuletzt abgerufen am 10. Februar 2019
- C++AMP: Language and Programming Model* – Version 1.0. August 2012.
- CUDA C Programming Guide. Oktober 2018. – <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>, zuletzt abgerufen am 26. Januar 2019
- HC API: Moving Beyond C++AMP for Accelerated GPU Computing. Dezember 2017 – <https://scchan.github.io/hcc/index.html>, zuletzt abgerufen am 03. Februar 2019

# Quellen

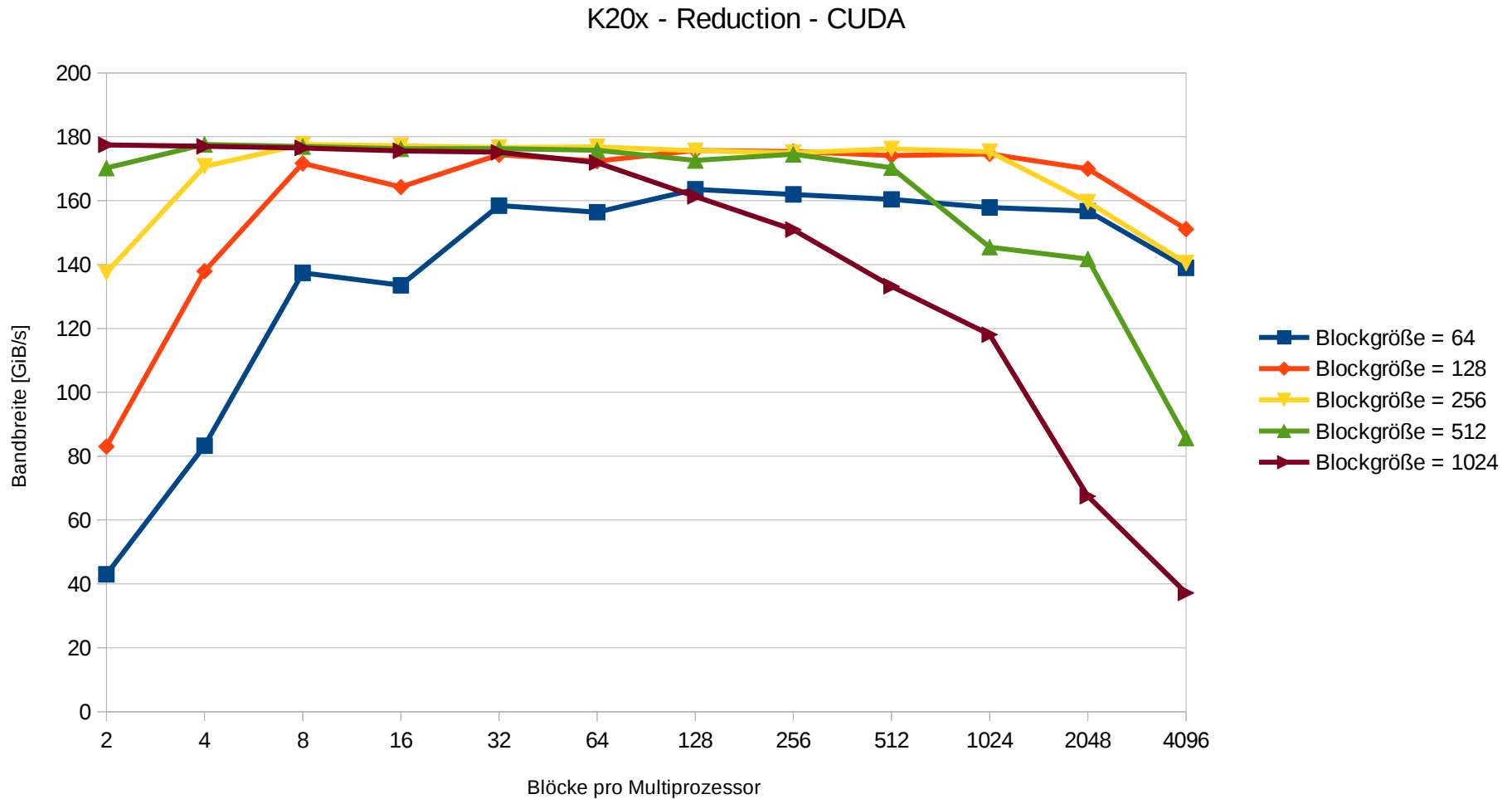
---

*Intel Project for LLVM technology.* Januar 2019. – <https://github.com/intel/llvm/tree/sycl>, zuletzt abgerufen am 10. Februar 2019

TOP500.org: *November 2018 / TOP500 Supercomputer Sites.* November 2018 – <https://www.top500.org/lists/2018/11>, zuletzt abgerufen am 18. Januar 2019

*triSYCL.* Dezember 2018. – <https://github.com/triSYCL/triSYCL>, zuletzt abgerufen am 10. Februar 2019

# Reduction-Benchmark: NVIDIA



# Reduction-Benchmark: AMD

