

GPGPUプログラミングの デバッグを楽しみたい

Zin (Sho Ikeda)

アジェンダ

- GPGPUは楽しい
- CPUとGPUで共通で使えるコードを書く
- 今回試した環境

GPGPUは楽しい

- 大量の並列演算器による高速な処理
 - CPUの数倍のコアによる大量並列処理
- グラフィックスや画像処理に向いている
 - 計算結果を見るのが楽しい

GPGPUは楽しい、ただし…

- GPGPUプログラミングは難しい
 - GPUアーキテクチャの理解 (Data-Level Parallelism, Memory address space, etc...)
 - GPU APIの理解 (DX, OpenCL, Vulkan, etc...)
 - プログラミング言語の理解
 - デバッグが難しい(ツールの不足, GUIのクラッシュ, NaN NaN Nan!, etc...)

GPGPUのデバッグを簡単にできないか？

CPUとGPUで共通で使えるコードを書く

- CPUのデバッグ環境をGPUコードでも使えるようにする
 - CPU側で実装ロジックが正しい事を検証してからGPU実行する
 - なるべくGPU環境でデバッグしない
- CPUによる同時実行、フォールバック実行
 - ただし、コードは(おそらく)GPU向けに最適化されている

本セッションはC++環境を想定

- C++17やC++20が使える
- Visual Studioやgdbなどのデバッガーが使える
- GPU側の言語はCuda, HIP, OpenCL C++など
- たぶんRustにも適用できる

CPUのデバッグ環境

- ブレイクポイント, ステップ実行, etc...
- Sanitizer (Address, UndefinedBehaviour, etc...)
 - AddressSanitizer: visual studio, clang, gccでサポート
 - Out of bounds access
 - Use after free
 - Division by zero
 - Mis-aligned access

The screenshot displays a Visual Studio Code interface with a C++ source file and its debug console. The source file shows a function `getTriangle` that takes a `face_buffer` and a `leaf_index` as input. The debug console shows an AddressSanitizer error: `AddressSanitizer: heap-use-after-free`. The error message indicates that the program is using memory that has been freed. The call stack shows the following frames:

- #13: `std::unique_ptr<zivc::ThreadManager, zivc::pmr::UniquePtrDeleter<zivc::ThreadManager>, unsigned int&, std::::1::pmr::memory_resource*> /home/sho/Projects/NanairoRtc9/source/dependencies/Zivc/...`
- #14: `zivc::CpuBackend::initData(zivc::ContextOptions&) /home/sho/Projects/NanairoRtc9/source/...`
- #15: `zivc::Backend::initialize(std::::1::weak_ptr<zivc::Backend&&, zivc::ContextOptions&) /...`
- #16: `void zivc::Context::initBackend(zivc::CpuBackend&) (zivc::ContextOptions&) /home/sho/Proj...`
- #17: `zivc::Context::initialize(zivc::ContextOptions&) /home/sho/Projects/NanairoRtc9/source/...`
- #18: `zivc::createContext(zivc::ContextOptions&) /home/sho/Projects/NanairoRtc9/source/depend...`
- #19: `nanairo::Renderer::Data::initializeContext(nanairo::CliOptions const&, std::::1::pmr::m...`
- #20: `nanairo::Renderer::initialize(nanairo::CliOptions const&, nanairo::GltfScene const&) /h...`
- #21: `main /home/sho/Projects/NanairoRtc9/source/nanairo/cmd/cmd.cpp:134:15`
- #22: `__libc_start_call_main csu/./sysdeps/nptl/libc_start_call_main.h:58:16`

The bottom of the screenshot shows the `CALL STACK` and `USE OF DEALLOCATED MEMORY` sections. The `CALL STACK` shows the following frames:

- 1: `tid=20083 "Nanairo"` (PAUSED)
- 2: `tid=20104 "Nanairo"` (PAUSED)
- 3: `tid=20105 "Nanairo"` (PAUSED)
- 4: `tid=20106 "Nanairo"` (PAUSED)
- 5: `tid=20107 "Nanairo"` (PAUSED)

The `USE OF DEALLOCATED MEMORY` section shows the following frames:

- `__asan_report_load4` (asan_rtl.cpp:121:1)
- `ReportGenericError` (asan_report.cpp:497:1)
- `ScopedInErrorReport` (asan_report.cpp:192:7)
- `sanitizer_termination.cpp` (55:7)
- `asan_rtl.cpp` (44)

CPUとGPUコードを共通化する1

- 特定の処理(関数)を共通化する
 - 乱数生成、レイの生成、BRDF、ライトサンプリングなど
- 比較的共通化しやすい
- 共通化できていない部分のGPUコードはデバッグしづらい

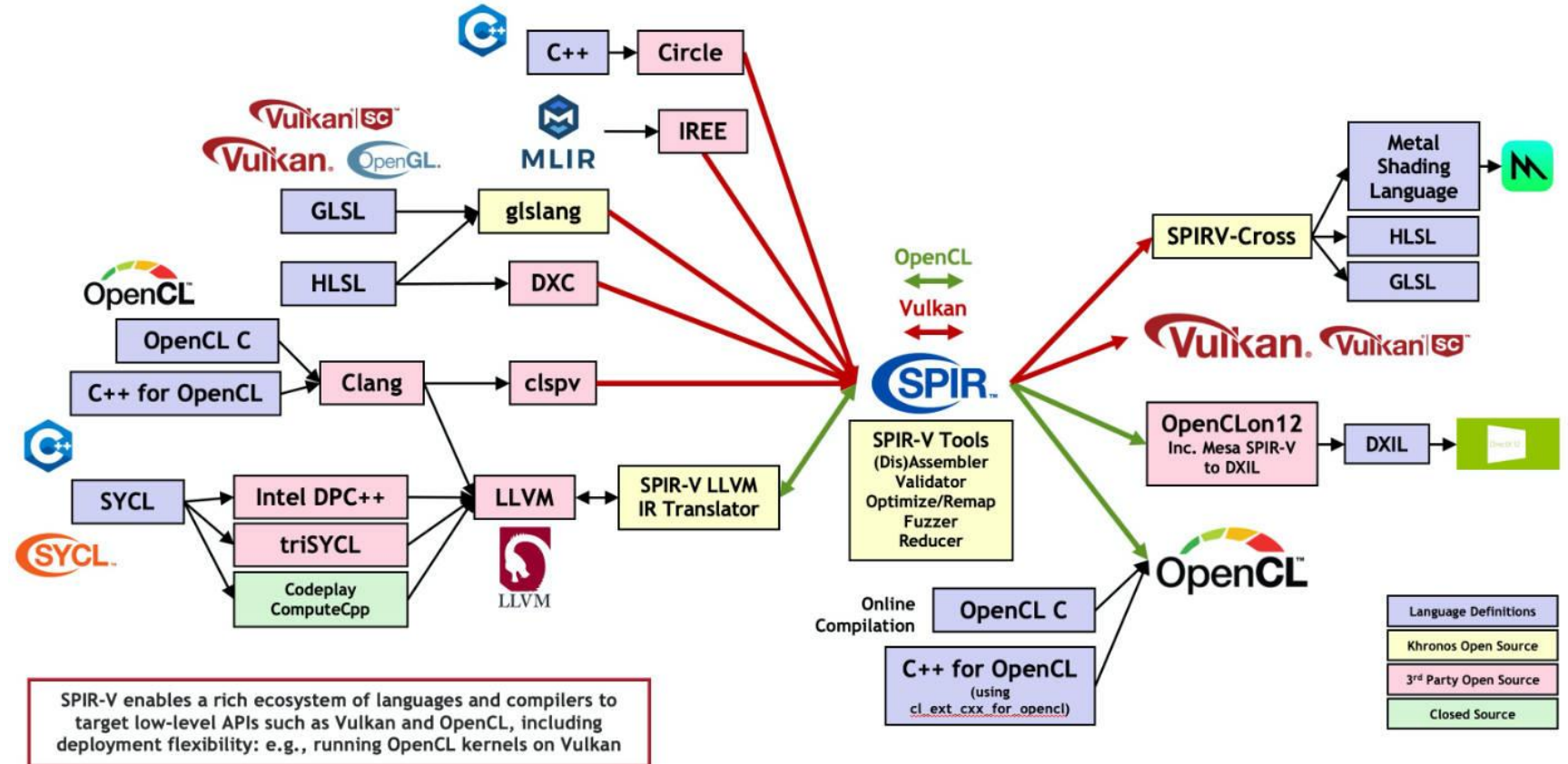
CPUとGPUコードを共通化する2

- カーネル全体をCPUとGPUで実行できるようにする
- カーネル全体のロジックをCPUで検証できる
- CPU側の実装が難しい
 - GPUアーキテクチャをCPU側でサポートする
 - thread_localを用いたwork-item情報の保持
 - coroutineを用いたwork-groupの実装
 - Device, Buffer, Kernelなどの抽象化

	Device	Buffer	Kernel
CPU	C++	配列	C++ function
GPU	Vulkan device	Vulkan buffer	OpenCL C++

今回試した環境

- C++20, Vulkan, OpenCL C++ (clspv)
- Linux+AMD, Windows+NV, cpu



ありがとうございました