SCALE: a Cross-Vendor extension of the CUDA Programming Model for GPUs

Chris Kitching

Outline

- **Motivation and overview**
- SCALE's compiler
  - Handling inline PTX `asm()`
  - The "CUDA Dialect Problem"
  - Handling different warp sizes
- Evaluation
- Conclusions

Problem: CUDA dominates the GPGPU ecosystem

- CUDA is the most widely-used GPGPU programming language
- CUDA is good:
  - Huge ecosystem of libraries, tools, etc.
  - Huge community of developers and knowledge
  - Good developer experience

CUDA can only target NVIDIA hardware.

Challenge: Make CUDA GPU programs as portable as CPU ones

- If you write CPU code in C/C++/Rust, you can compile it for ARM/Intel/AMD CPUs
  - ***Why can't CUDA be like that?!***

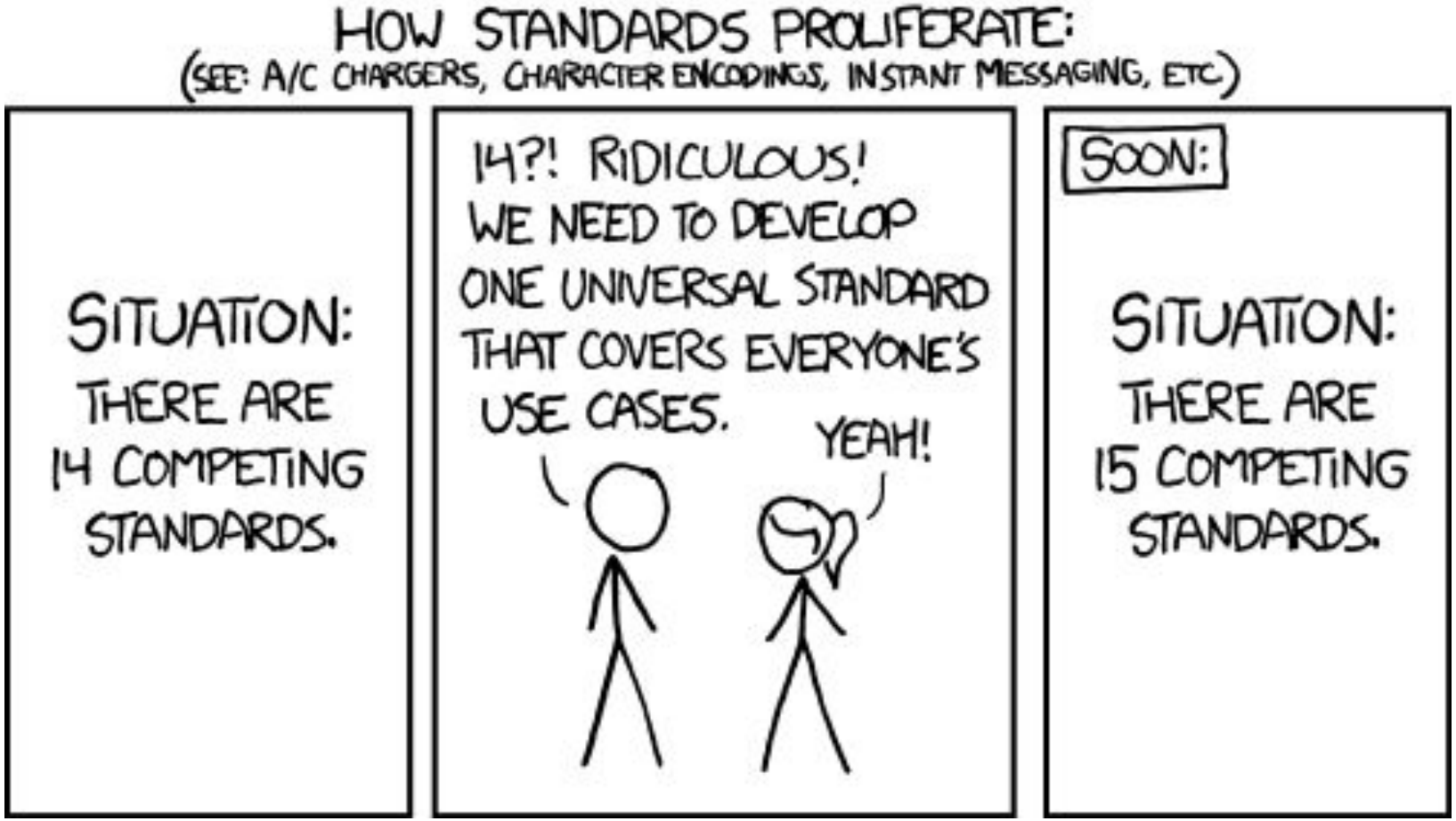
- Existing solutions emphasise porting away from CUDA
  - hipify/SYCLomatic: semi-automated porting
  - OpenCL/triton/etc. : alternative GPU languages

- Leads to a "compatibility tax".
  - **It's not a port if you're maintaining both versions.**
  - **It's not an "alternative toolkit" if you're only using it because the vendor forced you to.**

Automatic source translation does not work

- Tools like `hipify` aim to automate porting away from CUDA
- This approach has problems
  - Inline `asm()` is common in CUDA code, but has no source-level representation.
  - NVIDIA CUDA and Clang-CUDA are subtly different languages. The “dialect problem”
  - CUDA API calls are often assembled by macros, which present challenges for source translators
  - HIP’s APIs have subtly different semantics than NVIDIA’s
- Once done: now you have two codebases to maintain

Creating a new and incompatible standard does not work



OpenCL, HIP, SYCL, OneAPI, Triton, Mojo, ...

SCALE: A CUDA superset for non-NVIDIA GPUs

Solution: *Just make CUDA work*

- Build nvcc-equivalent compiler that can target other vendors
  - Build atop LLVM, so we can leverage existing vendor-maintained backends.
- Provide CUDA runtime/driver APIs for other vendors
- Write CUDA device libraries *in CUDA*
- Rewrite or shim closed-source libraries like cuBLAS/cuFFT.
- Validate by building/testing open-source CUDA projects

Outline

- Motivation and overview
- **SCALE's compiler**
  - Handling inline PTX `asm()`
  - The "CUDA Dialect Problem"
  - Handling different warp sizes
- Evaluation
- Conclusions

Handling inline `asm()` in CUDA

- NVIDIA's inline asm can simply be converted to LLVM IR during compilation.

CUDA

```
constexpr uin32_t Op = (0xF0 & 0xCC) ^ (~0xAA);
asm(
   "lop3.b32 %0, %0, %1, %2, %3;"
   : "+r"(x)
   : "r"(y), "r"(z), "n"(OP)
);
```

LLVM IR

```
%0 = and i32 %y, %x
%1 = xor i32 %0, %z
%2 = xor i32 %1, -1
```

AMDGPU Machine Code

```
v_and_b32_e32 v3, v4, v3
v_xnor_b32_e32 v2, v5, v3
```

Handling inline asm( ) in CUDA

```
__device__ __uint128_t int128_add(__uint128_t y, __uint128_t z) {
    unsigned Y[4], Z[4], X[4];

    memcpy(Y, &y, sizeof(__uint128_t));
    memcpy(Z, &z, sizeof(__uint128_t));

    asm("add.cc.u32   %0,%4,%8;   // extended-precision addition of\n"
        "addc.cc.u32  %1,%5,%9;   // two 128-bit values\n"
        "addc.cc.u32  %2,%6,%10;\n"
        "addc.u32     %3,%7,%11;\n"
        "" : "=r"(X[0]), "=r"(X[1]) , "=r"(X[2]) , "=r"(X[3]) :
            "r"(Y[0]), "r"(Y[1]), "r"(Y[2]), "r"(Y[3]),
            "r"(Z[0]), "r"(Z[1]), "r"(Z[2]), "r"(Z[3]));

    memcpy(&y, X, sizeof(__uint128_t));
    return y;
}
```

AMDGPU Machine Code

```
v_add_co_u32 v0, vcc_lo, v0, v4
v_add_co_ci_u32_e32 v1, vcc_lo, v1, v5, vcc_lo
v_add_co_ci_u32_e32 v2, vcc_lo, v2, v6, vcc_lo
v_add_co_ci_u32_e32 v3, vcc_lo, v7, v3, vcc_lo
```

The “CUDA Dialect Problem”

- The “standard” is defined by NVIDIA’s **nvcc** behavior

As the LLVM documentation puts it:

> ## Dialect Differences Between clang and nvcc
>
> There is no formal CUDA spec, and clang and nvcc speak slightly different dialects of the language.
>
> This section is painful; hopefully you can skip this section and live your life blissfully unaware.

What could possibly go wrong?

The “CUDA Dialect Problem”

- Discovered dialect issues include:
  - Accepting missing `typename` keywords
  - Differing SFINAE behaviour
  - “Exciting” parser behaviour surrounding `<< <`.
  - Accepting wildly-broken code if it can be proven not to be called.
  - “Interesting” function overloading rules
- SCALE offers an `nvcc` frontend implementing NVIDIA semantics.

The “CUDA Dialect Problem”

- Tiny example:

```
struct Foo {
    const int x = 2;
}
template<typename T>
__device__ void bar(Foo& o, T y) {
     o.x = 7;  // Invalid write to a const field
}
```

- nvcc accepts this since the template is unused.

The “CUDA Dialect Problem”

- Scarier example:

```
struct Example {
    const int x = 2;
}

__global__ void foo() {
     __shared__ Example x;
}
```

- Accepted by nvcc, silently dropping the initialiser on x.

Handling differing warp sizes

- All NVIDIA GPUs have a warp size of 32. Some AMD ones have 64.
- Many CUDA applications assume NVIDIA's 32-thread warps
  - Different warp size leads to incorrect behaviors of operations like ballot and shuffle
- SCALE offers two solutions:
  1. Emulate two 32-warps in each hardware warp
  2. Native warp-size mode, with compiler warnings to help catch non-portable code

Handling differing warp sizes

```
uint32_t x = __ballot_sync(foo, 0xFFFFFFFF);
```

- **warning**: implicit truncation of warp mask to 32-bits
- **warning**: suspicious constant warp mask 0xFFFFFFFF is half-full of zeros.

```
shfl.sync.bfly.b32 Ry, Rx, 0x10, %1, 0xFFFFFFFF
```

- **warning**: suspicious constant warp mask 0xFFFFFFFF is half-full of zeros.

  note: Did you mean -1?

Outline

- Motivation and overview
- SCALE's compiler
  - Handling inline PTX `asm()`
  - The "CUDA Dialect Problem"
  - Handling different warp sizes
- **Evaluation**
- Conclusions

Testbed

- Tested AMD microarchitectures
  - gfx942 (MI300X, CDNA3)
  - gfx94a (MI210, CDNA3)
  - gfx900 (Vega 10, GCN 5.0)
  - gfx1030 (Navi 21, RDNA 2.0)
  - gfx1100 (Navi 31, RDNA 3.0)

- Compile and run real CUDA programs
  - AMGX, FLAMEGPU2, ALIEN, GOMC, GPU JPEG2K, XGBoost, Faiss → **NO** AMD support
  - Rodinia suite, GPUJPEG, hashcat, LLaMA C++, Thrust, stdgpu → **HIP** for AMD

CUDA Coverage

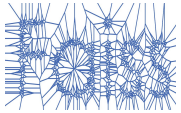
- SCALE currently tests:
  - 17 open-source CUDA projects
  - 5 AMD GPU architectures
- To run all of those SCALE covers
  - 88% of the CUDA 12.6 runtime API
  - 70%of the CUDA driver API
  - 80% of the CUDA math API
  - 100% cuSOLVER, cuBLAS, and cuFFT

Run on AMD **only** with **SCALE**

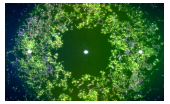
amgX

FLAMEGPU2

Faiss



alien



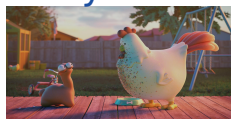
GOMC

gpujpeg2k

XGBoost

Multiple code bases:
HIP for AMD and CUDA for NVIDIA

cycles



hashcat

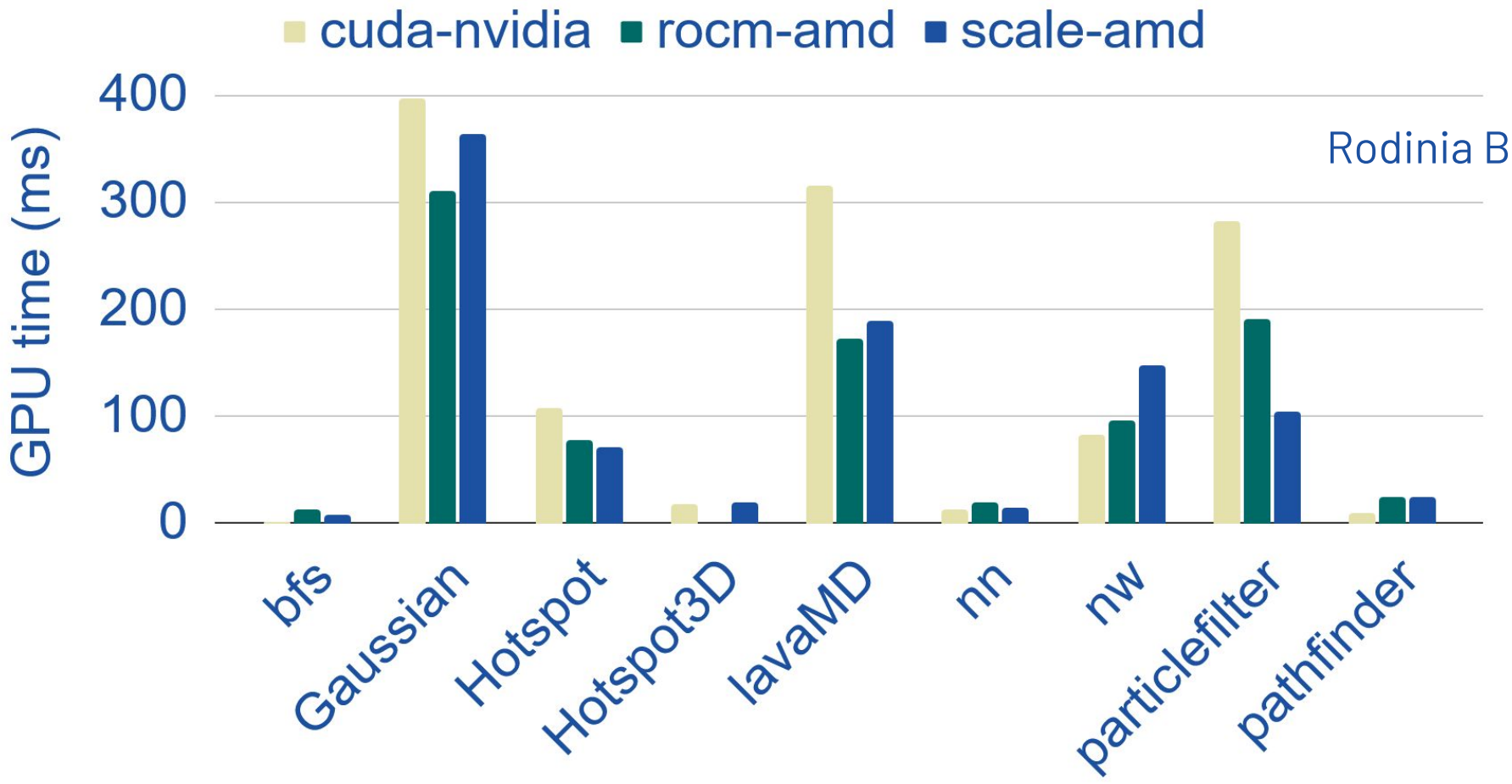
stdgpu

GPUJPEG

thrust

llama

SCALE offers native performance



Rodinia Benchmark Suite

Summary

- **SCALE** enables seamless execution of **CUDA** on **AMD** GPUs using:
  - A `clang` based compiler exposing an `nvcc`-equivalent interface
  - Support for the nvcc language dialect
  - A reimplementation of the core CUDA APIs
  - Implementations or shims of "CUDA-X" apis like cuBLAS
  - Opt-in language extensions, to innovate beyond CUDA

- We demonstrate SCALE's capabilities using
  - **17 real-world** applications
  - **5 AMD** microarchitectures

SCALE: a Cross-Vendor extension of the CUDA Programming Model for GPUs

Thank you

Chris Kitching

chris@spectralcompute.co.uk

SPECTRAL COMPUTE