SPECTRAL COMPUTE | SCALE

# Ahead-Of-Time Compilation of CUDA for HPC platforms

CUDA on AMD GPUs, and much more

# Presentation Outline

- **Motivation and overview**
- SCALE's clang compiler
  - Inline asm handling
  - Language dialect issues
  - Warp size differences
- CUDA shfl optimisation: a compiler deep-dive
- Q&A

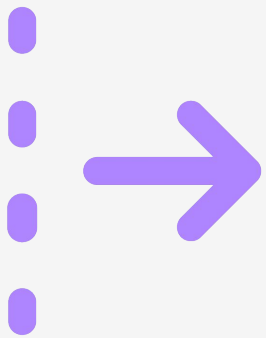
# Challenge: Make CUDA GPU programs as portable as CPU ones

→ C/C++/Rust can be compiled 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/Mojo etc.:** alternative GPU languages



Rewriting leads to a "compatibility tax".

- Maintainers frequently keep multiple versions of codebases
- Engineers forced to pick tools based on vendor compatibility

# Issues with automatic source translation

Tools like hipify partially automate porting away from CUDA.

Challenges:

- Inline asm() is common in CUDA code and is missed by source translation
- Dialect problem: NVIDIA CUDA and Clang-CUDA are subtly different languages;
- CUDA API calls are often assembled by macros;
- HIP's APIs behave subtly differently from CUDA's.

↪ Finished? Now you have two codebases to maintain 😭😭😭

# SCALE: A CUDA superset + compiler for non-NVIDIA GPUs

→ Our approach: **“just” make CUDA work!**

### `nvcc` that can target other vendors

Fork LLVM to provide a drop-in replacement for `nvcc`. Same command line, same semantics, but with *non-NVIDIA support*.

### CUDA Runtime and Driver API

Write a cross-platform implementation of the CUDA Driver/Runtime APIs, abstracting over the underlying hardware.

### CUDA Math API

Implement the CUDA Math API and other device-side APIs in CUDA, which can be built for any platform SCALE supports.

### “CUDA-X” Math libraries

Rewrite closed-source libraries such as cuBLAS in CUDA, or wrap vendor-provided options like AMD’s rocBLAS.

### Build and test FOSS CUDA projects

Validate it all by building and testing “real-world” CUDA code. Nothing beats complex GitHub projects when it comes to finding problems.

# GPU portability: A maze of dead ends

| ZLUDA | HIP | OpenCL | Mojo |
|---|---|---|---|
| + "CUDA Compatible"<br>+ Can support (most) programs without recompilation | + Open source<br>+ "Looks like CUDA" | + Open standard<br>+ Supported by most vendors | + Supports NVIDIA and AMD<br>+ Cool new language |
| - Requires use of NVIDIA toolkits<br>- Being rewritten because of legal clawback from AMD<br>- No AMD data-center grade GPU support<br>- Cannot match performance of native CUDA<br>- Uses HIP runtime | - Requires significant effort in porting and subsequent maintenance<br>- Mainly targets AMD (NVIDIA through largely untested backend)<br>- Significantly smaller ecosystem<br>- Significant stability and support issues[1] | - Restricted language features<br>- Significantly smaller ecosystem | - Significant porting and migration efforts from CUDA required<br>- In making "Python" performant, it ends up looking like CUDA/C++ without getting the ecosystem benefits |



[1]: MI300X vs H100 vs H200 Benchmark Part 1: Training – CUDA Moat Still Alive – SemiAnalysis

# CUDA Coverage

## SCALE currently tests:

- 23 open-source CUDA projects
- 11 AMD GPU architectures

## To run all of those, SCALE covers:

- ~90% of the CUDA 13 runtime API
- 70% of the CUDA driver API
- ~95% of the CUDA math API
- ~75% cuSOLVER/cuSPARSE
- 100% cuBLAS, cuFFT, cuRAND

Run on **AMD only with SCALE:**

| Project | SCALE | HIP |
|---|---|---|
| amgX | ✔ | ✘ |
| FLAMEGPU2 | ✔ | ✘ |
| Faiss | ✔ | ✘ |
| alien | ✔ | ✘ |
| XGBoost | ✔ | ✘ |
| GOMC | ✔ | ✘ |
| gpujpeg2k | ✔ | ✘ |
| CUTLASS | ✔ | ✘ |

**Multiple code bases:** HIP for AMD and CUDA for NVIDIA

| Project | SCALE | HIP |
|---|---|---|
| Hashcat | ✔ | ✔ |
| GROMACS | ✔ | ✔ |
| cycles | ✔ | ✔ |
| Llama.cpp | ✔ | ✔ |
| GPUJPEG | ✔ | ✔ |
| stdGPU | ✔ | ✔ |
| Thrust | ✔ | ✔ |

# Presentation Outline

- Motivation and overview
- **SCALE's clang compiler**
  - Inline asm handling
  - Language dialect issues
  - Warp size differences
- Live demo (what could go wrong?)
- CUDA shfl optimisation: a compiler deep-dive
- Q&A

# Handling inline asm() in CUDA

LLVM models asm as a function call: so let's just generate an IR function

### 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
```

### RDNA3 Machine Code:

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

# Handling inline asm() in CUDA

## 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
```

# Handling *cursed* inline asm() in CUDA

**It gets worse**

- Implicit dependencies between asm blocks?
- PTX block scoping with {}?
- The carry bit?

Time for a cursed example in the terminal...

# 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. Below, we describe some of the differences.
>
> This section is painful; hopefully you can skip this section and live your life blissfully unaware.

This section is painful; → *What could possibly go wrong?*

# 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”

Tiny+scary example:

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

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

↪ Accepted by **nvcc**, silently dropping the initializer on **x** (😵😵😵).

# The “CUDA Dialect Problem”

More terrifying example

```
template <typename T>
__device__ T device_callee() {
  return {};
}

template <typename Q>
__global__ void kernel() {
  device_callee<Q::Inner>(); // Missing typename keyword ignored
}
```

↪ **nvcc** has different rules for the cursed `typename` disambiguation keyword

# 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. Map two 32-warps onto 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?
```

# Presentation Outline

- Motivation and overview
- SCALE's clang compiler
  - Inline asm handling
  - Language dialect issues
  - Warp size differences
- **CUDA shfl optimisation: a compiler deep-dive**
- Q&A

# Portability doesn’t have to mean slow

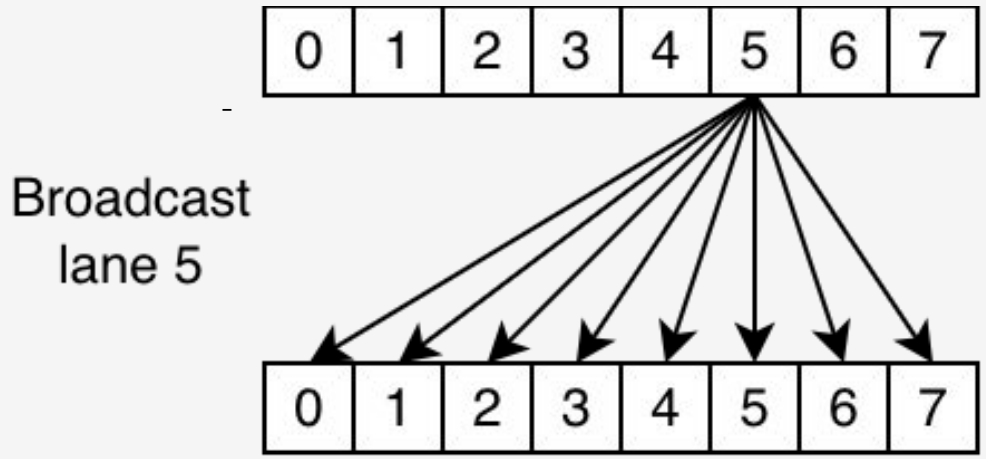
- Pervasive idea: portable code is slower than non-portable code.
- Counterexample: CPU compilers do a good job of leveraging diverse hardware features: let’s do that for gpu?!

Let’s explore how we did this for one common CUDA feature (shuffles).

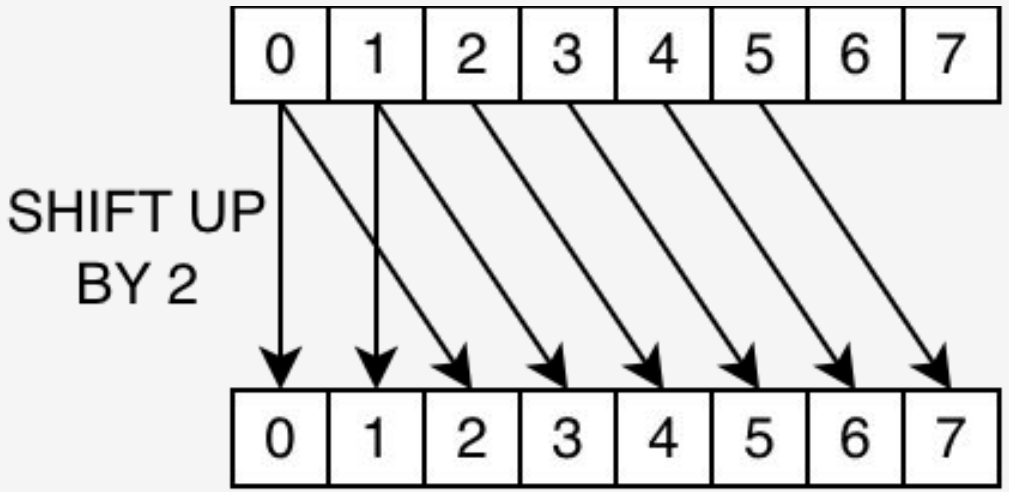
# Case study: CUDA shuffles - background

CUDA shuffles (`__shfl_xor()` and friends) move data between lanes.

__shfl(data, 5, 8)




__shfl_up(data, 2, 8)



# Case study: CUDA shuffles - AMD hardware

AMD offers multiple hardware features to perform this kind of operation

**Very general, most slow: ds_bpermute**

- Can do any permutation.
- Works by bounding data off the shared memory cache

**Least general, most fast: DPP**

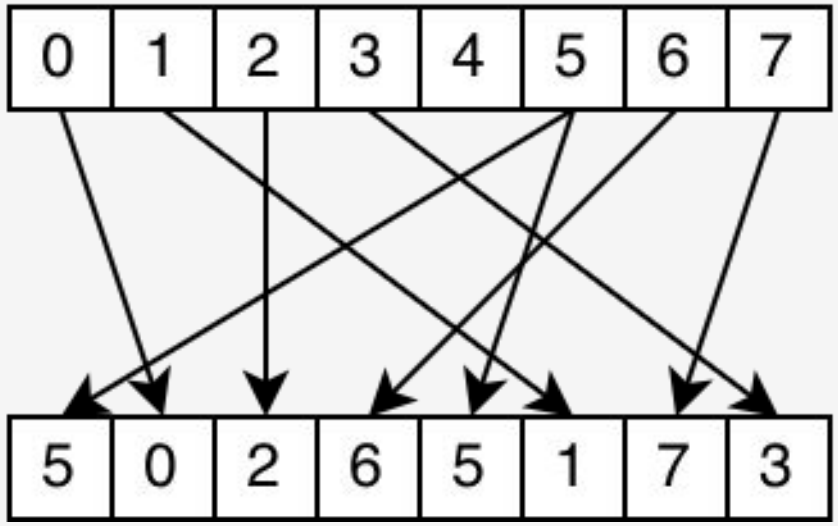
- Can only do specific hard-wired permutation patterns.
- Is literally wires
- Almost zero-cost

# Case study: CUDA shuffles - bpermute

Let's not pay for generality we don't need

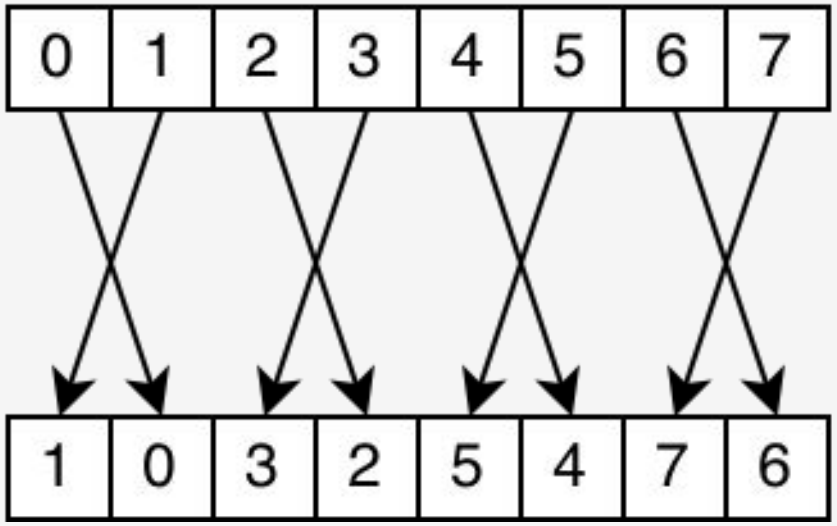
**no pattern**

bpermute is necessary




**xor 1**

should use a more specialised operation



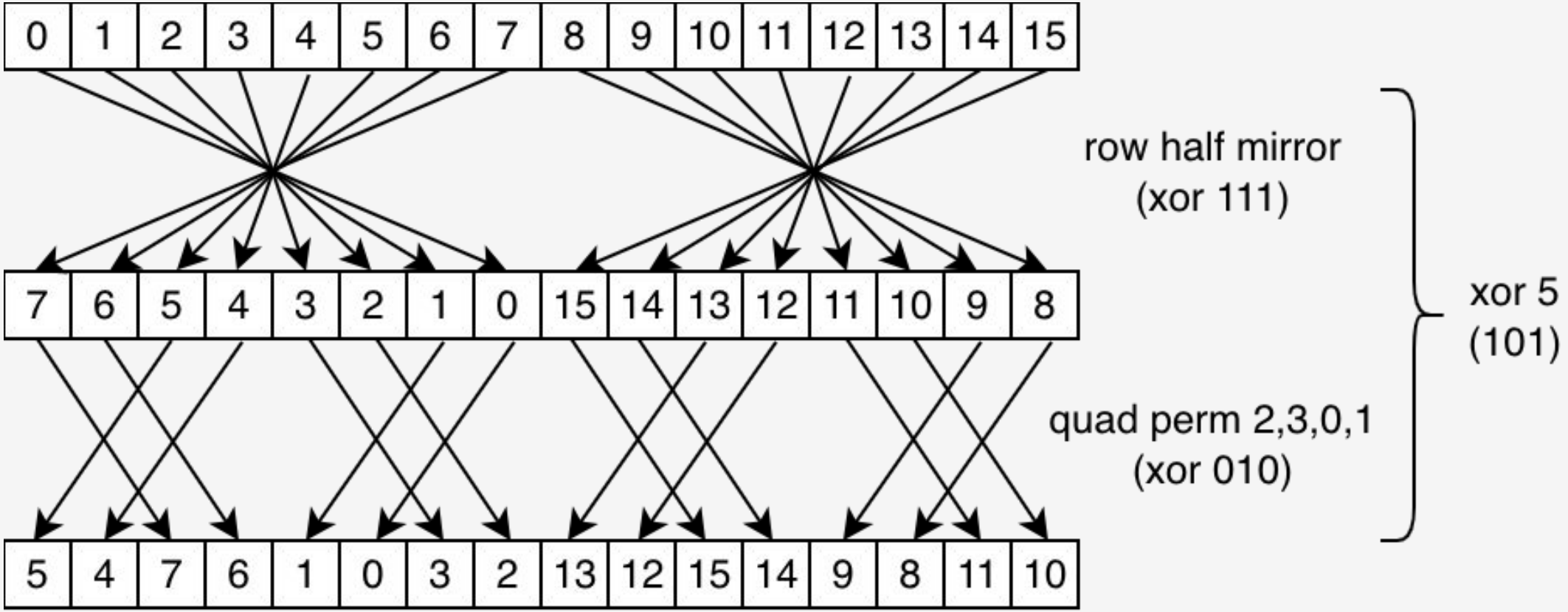
# Case study: CUDA shuffles - Let's optimise it

- Pattern match the permutations defined as `__shfl_*` APIs
- Map onto available hardware features

Optimisation applicable to both vendors (can select NVIDIA `redux` instructions!)

# Case study: CUDA shuffles - DPP chaining

DPP is so cheap, even multi-step permutations are profitable



# Where we're going

## Let's do that for tensor core instructions.

```
__global__ void k(TCD *d, const TAB *a, const TAB *b,
const TCD *c) {
    a += threadIdx.x * AFragSize;
    b += threadIdx.x * BFragSize;
    c += threadIdx.x * CFragSize;
    d += threadIdx.x * CFragSize;
    asm(".reg .f16x2 a<2>;\n"
        ".reg .f16x2 b<1>;\n"
        ".reg .f16x2 c<2>;\n"
        ".reg .f16x2 d<2>;\n"
        "ld.global.v2.b32 {a0, a1}, [%0];\n"
        "ld.global.b32 b0, [%1];\n"
        "ld.global.v2.b32 {c0, c1}, [%2];\n"

"mma.sync.aligned.m16n8k8.row.col.f16.f16.f16.f16
{d0, d1}, {a0, a1}, {b0}, {c0, c1};\n"
        "st.global.v2.b32 [%3], {d0, d1};\n" ::
        "l"(a), "l"(b), "l"(c), "l"(d));
}
```

```
v_mfma_f32_4x4x1f32 v[10:13], v8, v9, v[10:13] cbsz:2
        s_waitcnt lgkmcnt(2)
        v_cndmask_b32_e64 v9, v0, v6, s[6:7]
        v_cndmask_b32_e64 v0, v6, v0, s[6:7]
        s_nop 0
        v_mfma_f32_4x4x1f32 v[10:13], v5, v9, v[10:13] cbsz:2
        s_waitcnt lgkmcnt(1)
        v_cndmask_b32_e64 v9, v14, v3, s[6:7]
        s_nop 1
        v_mfma_f32_4x4x1f32 v[10:13], v8, v9, v[10:13] cbsz:2 abid:1
        s_waitcnt lgkmcnt(0)
        v_cndmask_b32_e64 v9, v15, v2, s[6:7]
        s_nop 1
        v_mfma_f32_4x4x1f32 v[10:13], v5, v9, v[10:13] cbsz:2 abid:1
        v_mfma_f32_4x4x1f32 v[10:13], v8, v1, v[10:13] cbsz:2 abid:2
        v_mfma_f32_4x4x1f32 v[10:13], v5, v0, v[10:13] cbsz:2 abid:2
        v_cndmask_b32_e64 v0, v3, v14, s[6:7]
        s_nop 1
        v_mfma_f32_4x4x1f32 v[6:9], v8, v0, v[10:13] cbsz:2 abid:3
        v_cndmask_b32_e64 v0, v2, v15, s[6:7]
        s_nop 1
        v_mfma_f32_4x4x1f32 v[6:9], v5, v0, v[6:9] cbsz:2 abid:3
        s_nop 4
        v_cvt_f16_f32_e32 v0, v9
        v_cvt_f16_f32_e32 v8, v8
        v_cvt_f16_f32_e32 v7, v7
         v_cvt_f16_f32_e32 v6, v6
```

# 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

- 21 real-world applications
- 14 AMD microarchitectures

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

## Thank you


**Chris Kitching, CTO**
chris@spectralcompute.com

**Ruben van Dongen, Head of Academic Solutions**
ruben@spectralcompute.com


- SCALE docs

- Discord