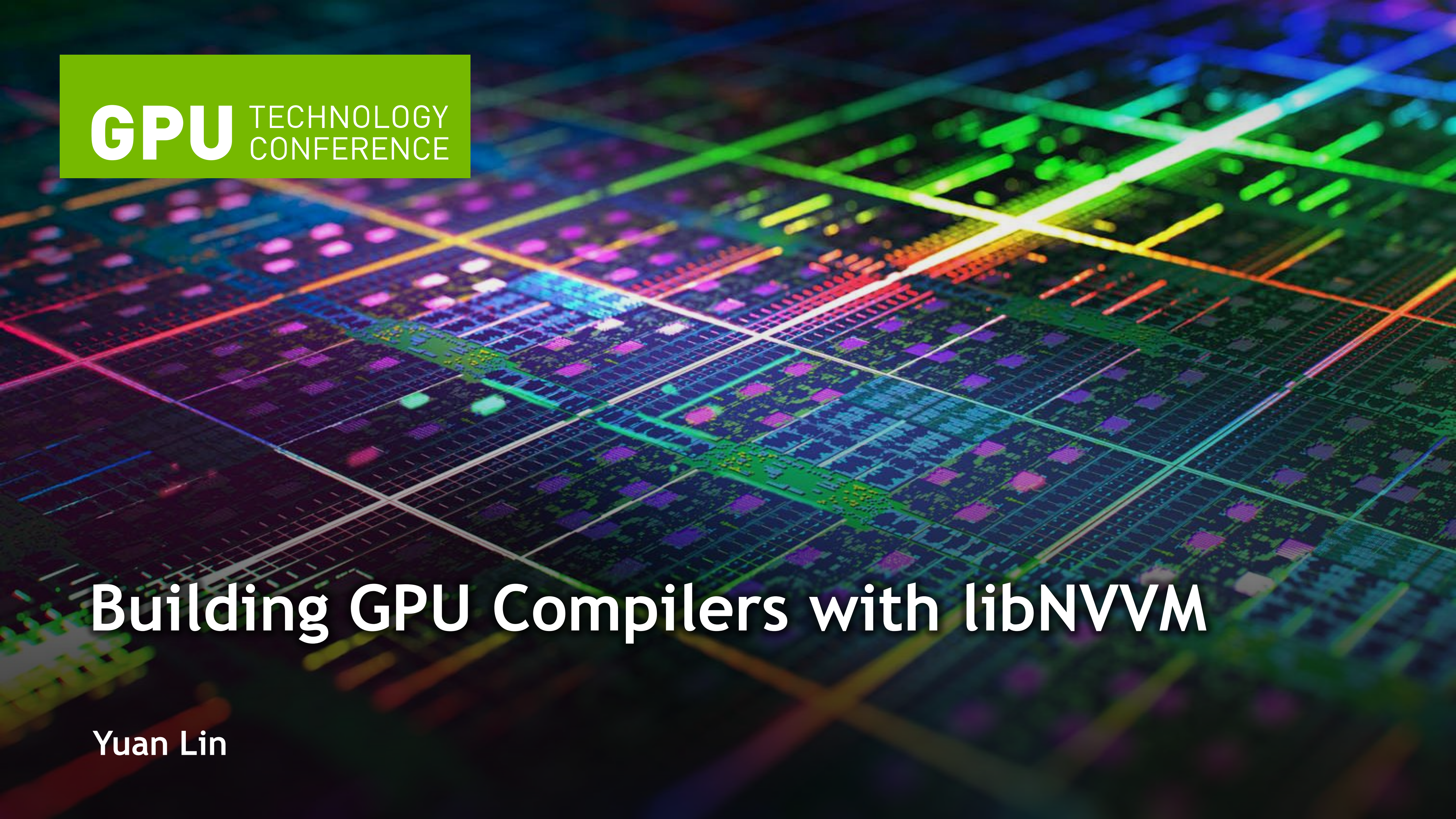
GPU TECHNOLOGY CONFERENCE

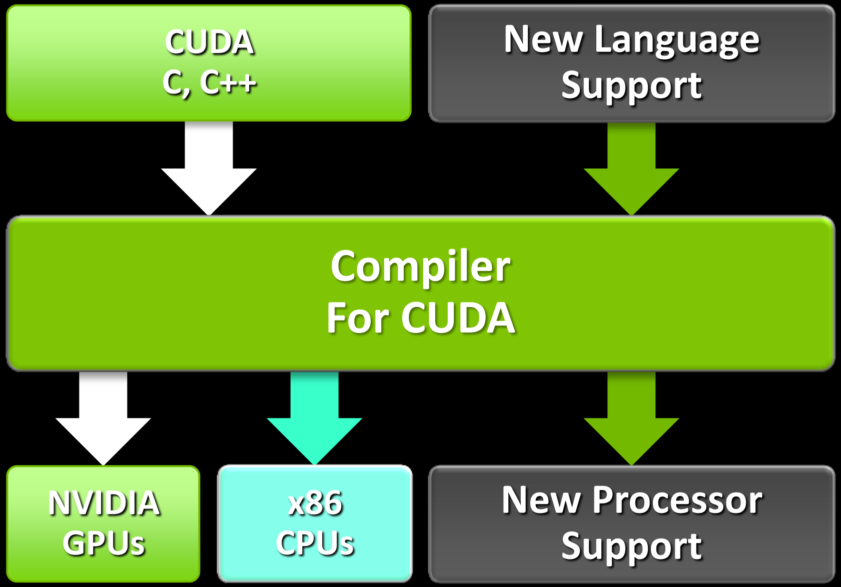


Building GPU Compilers with libNVVM

Yuan Lin

Vision

- Build a platform for GPU computing around foundations of CUDA.
  - Bring other languages to GPUs
  - Enable CUDA for other platforms
- Make that platform available for ISVs, researchers, and hobbyists
  - Create a flourishing eco-system

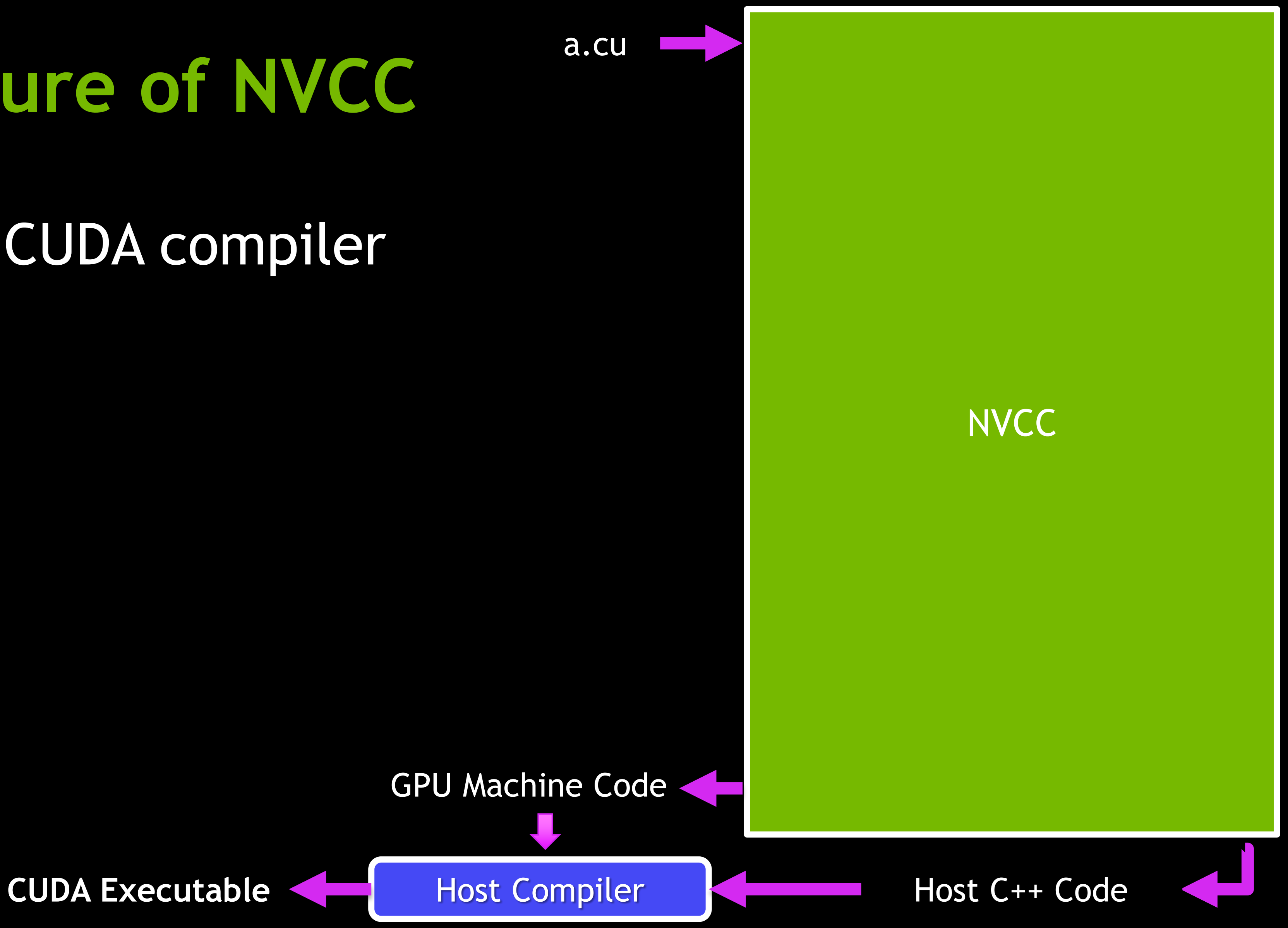


Structure of NVCC

- NVCC: CUDA compiler

Structure of NVCC

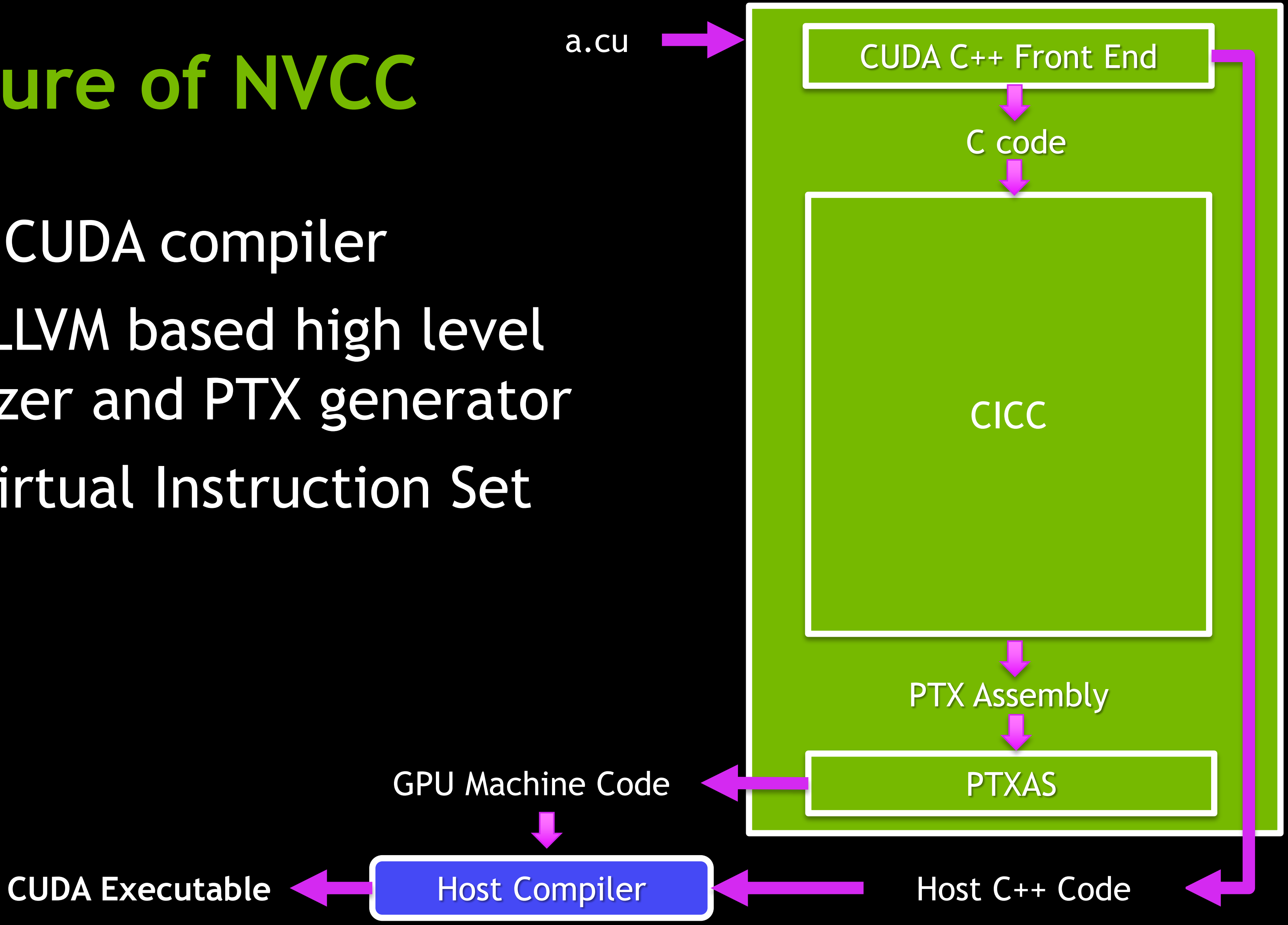
- NVCC: CUDA compiler



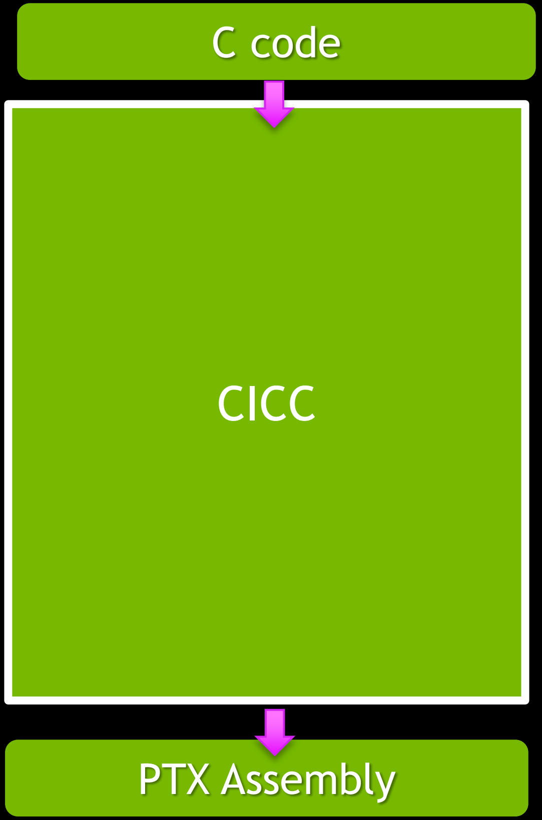
GPU TECHNOLOGY CONFERENCE

Structure of NVCC

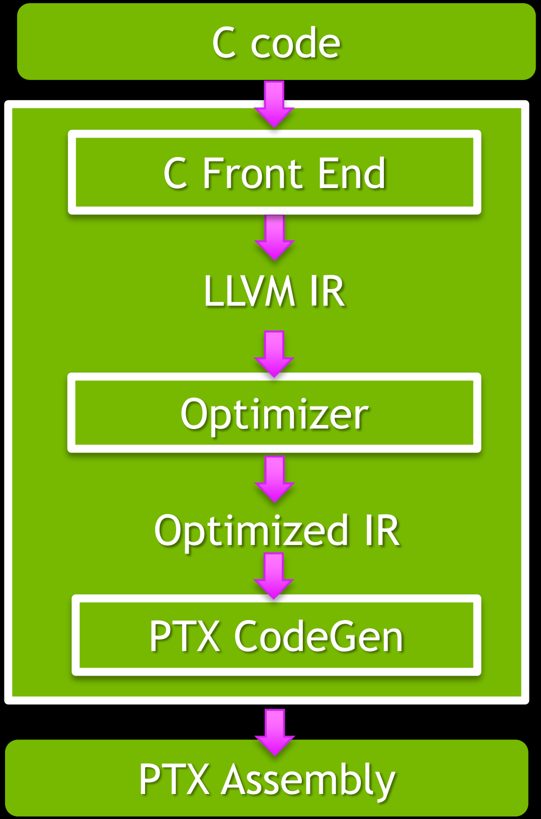
- NVCC: CUDA compiler
- CICC: LLVM based high level optimizer and PTX generator
- PTX: Virtual Instruction Set



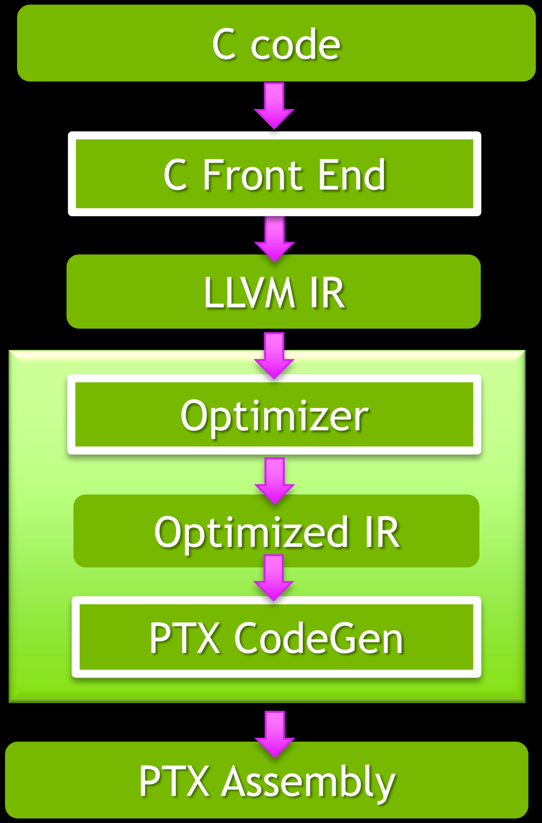
Structure of CICC



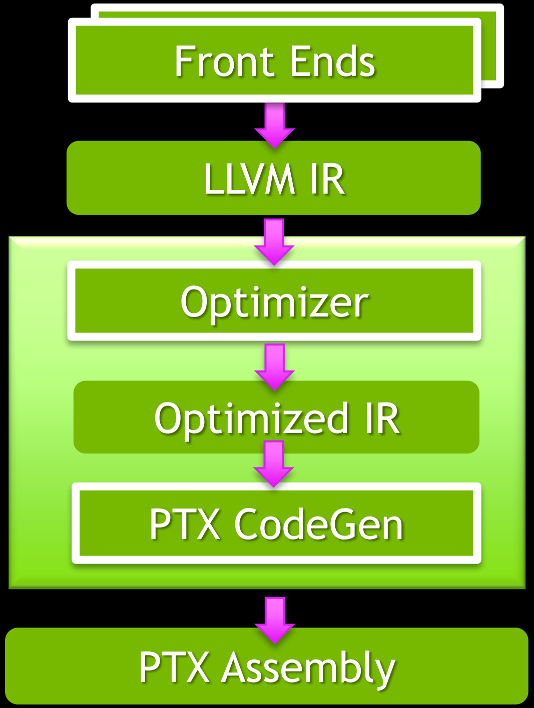
Structure of CICC



Structure of CICC

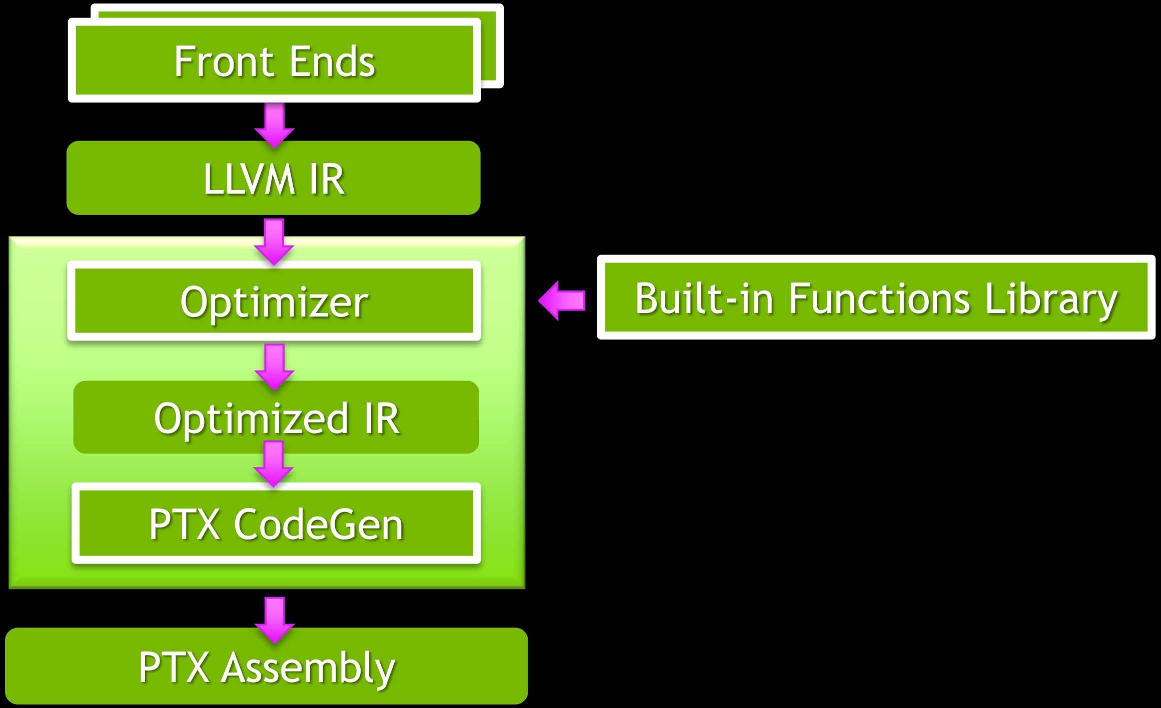


Common Compiler

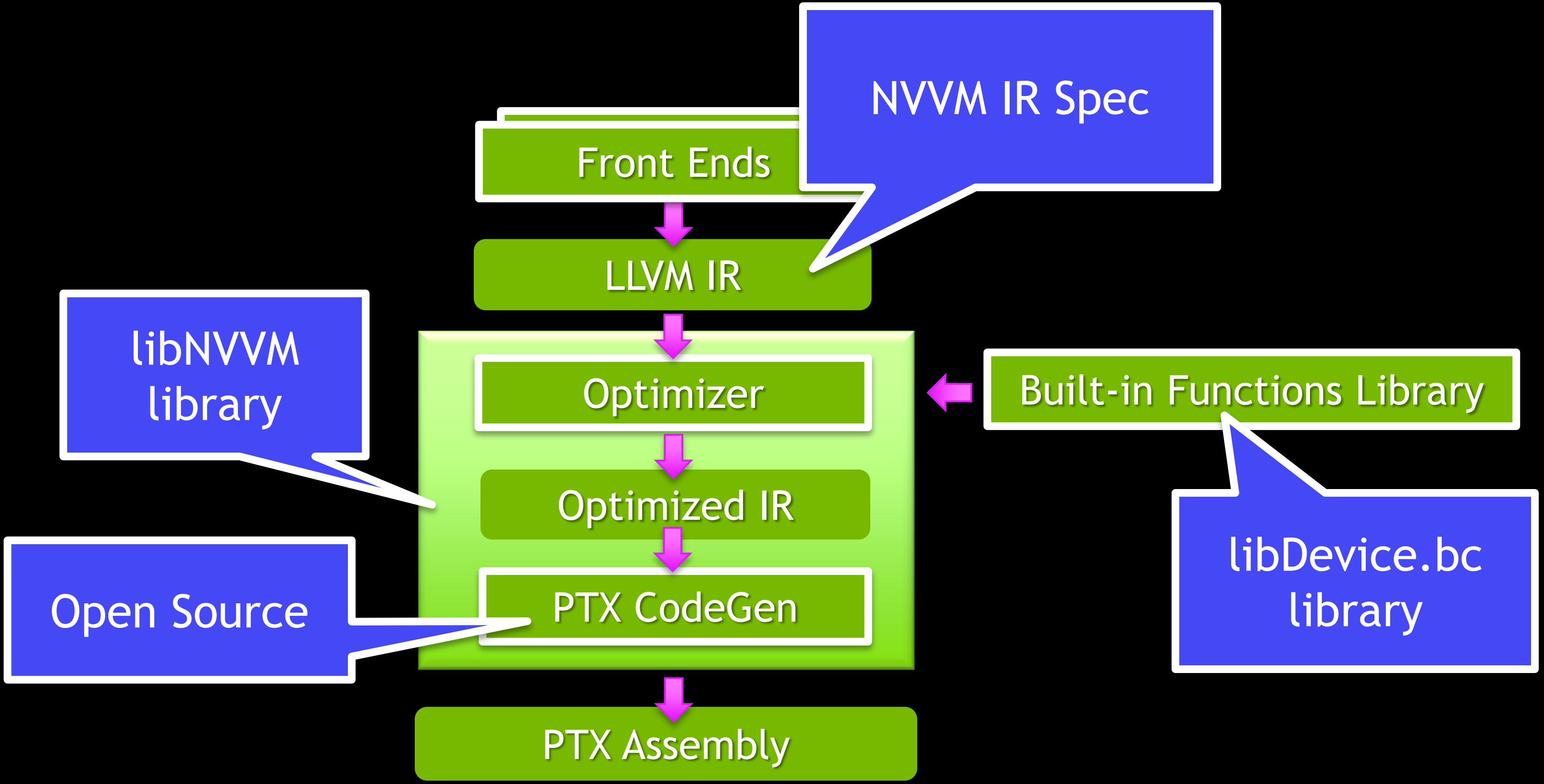


GPU TECHNOLOGY CONFERENCE

Common Compiler



Common Compiler



Two-pronged Approach

- Open-source LLVM NVPTX backend
  - Community supported
- NVIDIA Compiler SDK
  - Binary library, header files, documents
  - Supported product

Enabling Open Source GPU Compilers

- Contributed NVPTX code generator sources back to LLVM in summer 2012
- Part of LLVM 3.2 release
- Actively maintained in LLVM trunk
  - by NVIDIA and other LLVM developers
- Standard LLVM License
- Best for
  - Prototyping
  - Developers who work only with LLVM trunk

NVIDIA Compiler SDK

- Preview was released at GTC 2012.
- 1st official release will be included in CUDA 5.5 toolkit.

LLVM NVPTX / libNVVM Users

- Numba Pro: array-oriented compiler for Numpy/Python
- Halide image processing language : MIT (halide-lang.org)
- Jet fluid dynamics DSL : Double Negative
- Alea.CUDA, F# on the GPU : QuantAlea
- Delite parallel EDSL framework : Stanford PPL
- KernelGen: open source compiler project at HPC forge

CONTINUUM ANALYTICS

double negative visual effects

QuantAlea

PERVASIVE PARALLELISM LABORATORY PPL

NVIDIA Compiler SDK

- NVVM IR specification
- libNVVM library and header file
- libDevice
- Code samples
- Developer’s guide and API document

libNVVM Library

libNVVM Library

- An optimizing compiler library that generates PTX from NVVM IR
  - Supports LLVM 3.0, 3.1 and 3.2 IR format
  - Supports Fermi, Kepler and later architectures
- Available on 32-bit and 64-bit Windows, Linux and Mac
- Actually used by CUDA 5.5 compiler
  - Dynamically linked by cicc

libNVVM Library

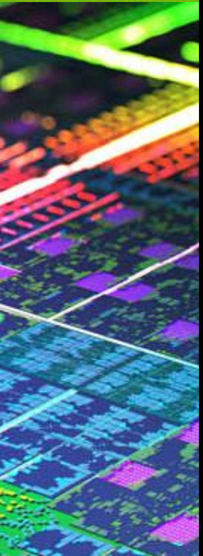
- Analysis and optimizations
  - Address space access optimization
  - Thread variance/convergence analyses
  - Re-materialization
  - Load/store coalescing
  - Sign extension elimination
  - New phi elimination
  - Enhanced alias analysis
  - …
- Support DWARF generation

libNVVM C APIs

- Create a program
  - Add NVVM IR modules to the program
  - Support NVVM IR level linking
- Verify the input IR
- Compile IR to PTX
- Get result
  - Get back PTX string
  - Get back message log

libDevice

GPU TECHNOLOGY CONFERENCE



libDevice

- Common device math functions
  - Distributed in LLVM bitcode format
  - Supports Fermi and Kepler
- Supports both ftz and non-ftz mode
- Can be linked with NVVM IR program using libNVVM API
  - Treated as a normal NVVM IR module in libNVVM
- Can be used with the open source LLVM NVPTX backend
- We build our CUDA math functions on top of it in CUDA 5.5.
- Will include more common device functions in the future.

NVVM IR

NVVM IR

- Designed to represent GPU kernel functions and device functions
  - Represents the code executed by each CUDA thread
- NVVM IR Specification 1.0
  - Based on LLVM IR 3.2

NVVM IR and LLVM IR

- NVVM IR
  - Based on LLVM IR
  - With a set of rules and intrinsics
- No new types. No new operators. No new reserved words.
- An NVVM IR program can work with any standard LLVM IR tool

  ✓ llvm-as ✓ llvm-link ✓ llvm-extract

  ✓ llvm-dis ✓ llvm-ar ✓ …
- An NVVM IR program can be built with the standard LLVM distribution.

```
svn co http://llvm.org/svn/llvm-project/llvm/branches/release_32 llvm
```

NVVM IR

NVVM IR: Address Spaces

NVVM IR: Address Spaces

- CUDA C++
  - Address space is a storage qualifier.
  - A pointer is *generic* pointer, which can point to any address space.

```
__global__ int g;
__shared__ int s;
__constant__ int c;
void foo(int a) {
    int l;
    int *p ;
    switch (a) {
    case 1:  p = &g; …
    case 2:  p = &s; …
    case 3:  p = &c; …
    case 4:  p = &l; …
    }
    …
}
```

NVVM IR: Address Spaces

- CUDA C++
  - Address space is a storage qualifier.
  - A pointer is *generic* pointer, which can point to any address space.
- OpenCL C
  - Address space is part of the type system.
  - A pointer type must be qualified with an address space.

```
constant int c;

foo(global int *pg) {

    int l;
    int *p ;
    p = &l;

    constant int *pc = &c;
    …

}
```

NVVM IR: Address Spaces

- CUDA C++
  - Address space is a storage qualifier.
  - A pointer is *generic* pointer, which can point to any address space.
- OpenCL C
  - Address space is part of the type system.
  - A pointer type must be qualified with an address space.
- NVVM IR
  - Support both use cases in the same program.

NVVM IR: Address Spaces

- Define address space numbers
- Allow generic pointers and specific pointers
- Provide intrinsics to perform conversions between generic pointers and specific pointers

NVVM IR: Address Spaces

- Allow module scope variables that are in the global address space to have generic address values.
  - Make generating NVVM IR code much easier.

```
// @a is a module scope variable residing in the
// global address space (1).
// The address value of @a is a global address value.
@a = addrspace(1) float 0.000000e+00
```

NVVM IR: Address Spaces

- Allow module scope variables that are in the global address space to have generic address values.
  - Make generating NVVM IR code much easier.

```
// @a is a module scope variable residing in the
// global address space (1).
// The address value of @a is a global address value.
@a = addrspace(1) float 0.000000e+00

// @b is a module scope variable residing in the
// global address space (1).
// The address value of @b is a generic address value.
@b = float 0.000000e+00
```

NVVM IR: GPU Program Properties

- Properties:
  - Maximum expected CTA size from any launch
  - Minimum number of CTAs on an SM
  - Kernel function vs. device function
  - Texture/surface variables
  - more
- Use named metadata

NVVM IR: Intrinsics

- Atomic operations
- Barriers
- Address space conversions
- Special registers read
- Texture/surface access
- more

Programming Guide

Developer’s Guide and API Document

- NVVM IR Specification
- libNVVM: Developer’s Guide and API Spec
- libDevice: Developer’s Guide and API Spec
- Will be available
  - online @ docs.nvidia.com
  - as PDF files

Samples

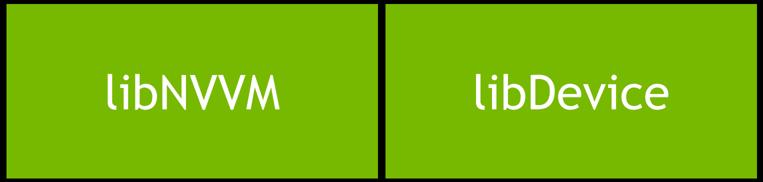
Samples

- "simple"
  - JIT compile a NVVM IR program using libNVVM
  - launch it using CUDA driver API

libNVVM
CUDA Driver

Samples

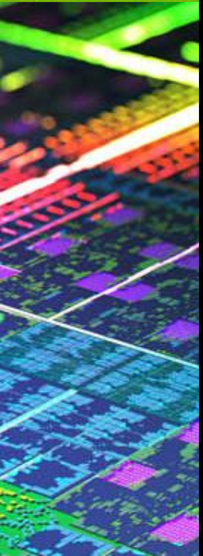
- "ptxgen"
  - A simple offline NVVM IR to PTX compiler
  - Links in libDevice



Samples

- "cuda-c-linking"
  - Create a NVVM IR program using LLVM IR builder API
  - JIT compile the NVVM IR program using libNVVM
  - Link it with a PTX generated from CUDA C using PTX JIT linking API
  - Launch the final code using CUDA driver API

libNVVM | LLVM | PTX JIT linking



Samples

- “simple”
- “ptxgen”
- “cuda-c-linking”
- Part of CUDA 5.5 toolkit

Samples

- Samples that
  - are relatively big,
  - depend on other open-source package, or
  - need to be updated more frequently

Samples on github

- Samples that
  - are relatively big,
  - depend on other open-source package, or
  - need to be updated more frequently
- Examples
  - Other language bindings for libNVVM: Python, Haskell
  - Kaleidoscope
  - Small utilities
- Open source BSD style license

Example

SAXPY

```
let (saxpy (lambda (a : f32)
                   (x : vf32)
                   (y : vf32) : vf32
                   (map (lambda (xi : f32)
                                (yi : f32) : f32
                                (+ (* a xi) yi)  )
                         x y)))
in (saxpy A X Y)
```

SAXPY

```
let (saxpy (lambda (a : f32)
                   (x : vf32)
                   (y : vf32) : vf32
                   (map (lambda (xi : f32)
                                (yi : f32) : f32
                                (+ (* a xi) yi)  )
                         x y)))
in (saxpy A X Y)
```

SAXPY

- Execute this on the GPU.
- Use one GPU thread for each vector element.

```
saxpy(float a, float *x, float *y) {
  y[thread_id] = a * x[thread_id] + y[thread_id];
}
```

```
let (saxpy (lambda (a : f32)
                   (x : vf32)
                   (y : vf32) : vf32
                   (map (lambda (xi : f32)
                                (yi : f32) : f32
                                (+ (* a xi) yi) )
                         x y)))
in (saxpy A X Y)
```

```
@n = internal global i32 0, align 4
@a = internal global float 0.000000e+00, align 4

define void @saxpy(float* %x, float* %y) {
}
```

```
@n = internal global i32 0, align 4
@a = internal global float 0.000000e+00, align 4

define void @saxpy(float* %x, float* %y) {
}

!nvvm.annotations = !{!0}
!0 = metadata !{void (float, float*, float*)* @saxpy, metadata !"kernel", i32 1}
```

```
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32
-i64:64:64-f32:32:32-f64:64:64-v16:16:16-v32:32:32-v64:64:64-v128:128:128-n16:32:64"

@n = internal global i32 0, align 4
@a = internal global float 0.000000e+00, align 4

define void @saxpy(float* %x, float* %y) {
}

!nvvm.annotations = !{!0}
!0 = metadata !{void (float, float*, float*)* @saxpy, metadata !"kernel", i32 1}
```

```
define void @saxpy(flat %a, float* %x, float* %y) {

  ; load @n and @a
  ; int i = blockIdx.x * blockDim.x + threadIdx.x;
  ; load x[i]
  ; load y[i]
  ; a * x + y
  ; store y[i]
}
```

```
define void @saxpy(float %a, float* %x, float* %y) {

  ; load @n and @a
  %n = load i32 * @n, align 4
  %a = load float * @a, align4

  …
}
```

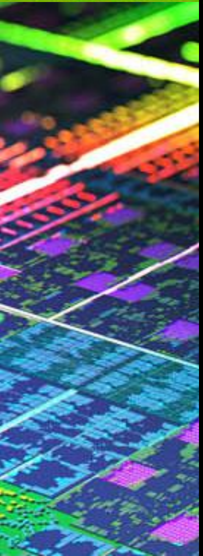
```
declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
declare i32 @llvm.nvvm.read.ptx.sreg.ntid.x()
declare i32 @llvm.nvvm.read.ptx.sreg.tid.x()

define void @saxpy(float %a, float* %x, float* %y) {

  ; load @n and @a
  %n = load i32 * @n, align 4
  %a = load float * @a, align4

  ; int i = blockIdx.x * blockDim.x + threadIdx.x;
  %0 = call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
  %1 = call i32 @llvm.nvvm.read.ptx.sreg.ntid.x()
  %mul = mul i32 %1, %0
  %2 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
  %add = add i32 %mul, %2

   …
}
```

```
declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
declare i32 @llvm.nvvm.read.ptx.sreg.ntid.x()
declare i32 @llvm.nvvm.read.ptx.sreg.tid.x()

define void @saxpy(float %a, float* %x, float* %y) {

  ; load @n and @a
  %n = load i32 * @n, align 4
  %a = load float * @a, align4

  ; int i = blockIdx.x * blockDim.x + threadIdx.x;
  %0 = call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
  %1 = call i32 @llvm.nvvm.read.ptx.sreg.ntid.x()
  %mul = mul i32 %1, %0
  %2 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
  %add = add i32 %mul, %2

  ; load x[i]
  %3 = sext i32 %add to i64
  %x_ptr = getelementptr float * %x, i64 %3
  %x_value = load float * %x_ptr, align 4

  ; load y[i]
  %y_ptr = getelementptr float * %y, i64 %3
  %y_value = load float * %y_ptr, align 4

  ; a * x + y
  %mul11 = fmul float %a, %x_value
  %add16 = fadd float %mul11, %y_value

  ; store y[i]
  store float %add16, float * %y_ptr, align 4

  ret void

}
```

Using libNVVM APIs

```
#include "nvvm.h"
char *compiler ()
{
  nvvmProgram prog;
  nvvmCreateProgram(&prog);
  char *buffer;  size_t size;
  getSAXPYir(&buffer, &size);
  nvvmProgramAddModule(prog, buffer, size);
  getLibDevice(&buffer, &size, "libdevice.compute_" #ARCH "." #MAJOR #MINOR ".bc")
  nvvmProgramAddModule(prog, buffer, size);
  nvvmCompileProgram(prog, 0, NULL);
  size_t ptxSize;
  nvvmGetCompiledResultSize(prog, &ptxSize);
  char *ptx = (char *) malloc(ptxSize);
  nvvmGetCompiledResult(prog, ptx);
  nvvmDestroyProgram(&prog);
  return ptx;
}
```

GPU TECHNOLOGY CONFERENCE



How to get them?

- Distributed with CUDA 5.5
  - NVVM IR spec
  - libNVVM library and header file
  - libDevice
  - Code samples
  - Developer's guide and API document
- LLVM.org: NVPTX backend
- Github: More samples
- devtalk.nvidia.com: Post your questions/suggestions