PLANG: PTX
Frontend for LLVM

Vinod Grover
joint work with Andrew Kerr
and Sean Lee
PTX - Parallel Thread eXecution
- Virtual ISA describing GPU compute kernels
- JIT compilation to GPU's native ISA
- Targeted by CUDA compiler, OptiX, and NVIDIA OpenCL

LLVM - Low Level Virtual Machine
- Platform-independent compiler framework
- Mature optimization and code generation components
Goals and Motivations

- leverage LLVM compiler framework for PTX
- PTX part of LLVM ecosystem
  - build analysis/transform/translation tools for PTX using LLVM
- PTX to PTX kernel optimization
- PTX to multicore CPU
  - execution model translation
  - target ISAs with existing code generators (e.g. x86)
Overview

```
ptx -> ptxparse -> ptx2llvm

std

llvm
```

plang

```
.bc
```
PTX Instruction Set

- kernels express coarse- and fine-grain parallelism
- explicit specification of memory spaces for data
- matches GPU architectures well - SIMD execution, divergent control flow
LLVM Internal Representation

- strict type rules and explicit conversion
- expresses function in static single-assignment form (SSA)
- instructions apply to scalar and vector types
- extended by adding *intrinsics* functions
- pointers specify address space
Non-SSA to SSA

PTX is not in SSA form whilst LLVM IR is.

- allocate virtual registers in PTX as memory objects in LLVM

```
PTX
.reg .u32 %r<10>;

LLVM IR
%r9 = alloca i32
%r8 = alloca i32
%r7 = alloca i32
...```
Non-SSA to SSA (Cont’d)

PTX is not in SSA form whilst LLVM IR is.

▶ use store and load to reuse the memory objects

### PTX

```ptx
add.s32 %r7, %r5, %r3;
```

### LLVM IR

```llvm
%tmp6 = load i32 *%r5
%tmp7 = load i32 *%r3
%tmpAdd = add i32 %tmp6, %tmp7
store i32 %tmpAdd, i32* %r7
```

mem2reg optimization pass in LLVM transforms `alloca / load / store` into values and data flow.
LLVM Intrinsic Functions

Built-in Variable Accessors
- gridDim, blockDim, blockIdx, threadIdx

Transcendental functions
- cos, sin, sqrt, rsqrt, log2, ex2

Synchronization
- bar.sync, red

Texture Sampling
- tex.{1d,2d,3d}

Atomic Global Memory Access
- atomic.{ }
LLVM Intrinsic Functions

**PTX**

```plaintext
mov.u16 %rh1, %ctaid.x;
...

tex.1d.v4.s32.s32 {,%r114,%r115,%r116,%r117},[...]
```

**LLVM IR**

```plaintext
%tmp5 = call fastcc i16 @llvm.ptx.ctaid.x()
store i16 %tmp5, i16* %rh1
...

call void @llvm.ptx.tex.1d.v4.s32.s32
(i32* %r114, i32* %r115, i32* %r116, i32* %r117, ...)
```
PTX Types in LLVM

LLVM integers do not carry signedness information whilst PTX integers do. Instead, \texttt{zext}, \texttt{sext} and \texttt{trunc} are used as needed.

\begin{Verbatim}[commandchars=\[\]]
PTX
\texttt{cvt.s32.u16 \%r3, \%r2;}
\texttt{cvt.u16.u32 \%rh1, \%r3;}
\end{Verbatim}

\begin{Verbatim}[commandchars=\[\]]
LLVM IR
\texttt{\%tmp2 = load i16* \%r2}
\texttt{\%tmp3 = zext i16 \%tmp2 to i32}
\texttt{store i32 \%tmp3, i32* \%r3}
\texttt{\%tmp4 = load i32* \%r3}
\texttt{\%tmp5 = trunc i32 \%tmp4 to i16}
\texttt{store i16 \%tmp5, i16* \%rh1}
\end{Verbatim}
Kernel Translation

Translated kernel assumes CTA execution model
  ▶ PTX intrinsics convey structure
LLVM translation *should* be functionally invertible
  ▶ integer signedness ignored by LLVM
  ▶ floating-point rounding modifiers ignored by PLANG

Procedure:
  1. parse PTX source file into PTX IR
  2. for each PTX kernel, construct LLVM function
  3. local variable allocation
  4. translate each PTX instruction to one or more LLVM instructions
Goal: efficient execution of PTX kernel on multicore CPU (x86)

- transform execution model of PTX kernel in LLVM IR
- modify *nvcc* (CUDA compiler) to invoke translated function
- link to CUDA on Multicore runtime library
  - based on Windows threads and Linux pthreads
Execution Model Translation

- Thread loop placement
- Scalar expansion
- Allocation of thread-local storage
- Replacement of intrinsics
Thread loops require live variables to be spilled

```c
// transformed C code with thread loops
void kernel(
    float *Out,
    float *In)
{
    _shared_ float Shr[16][17];
    int index;

    foreach (tid in threads) {
        int i = threadIdx.y;
        int j = threadIdx.x;

        index = i * 16 + j; // 'index' value from previous
        // loop iteration killed

        Shr[j][i] = In[index];
    }

    // __syncthreads(); // 'index' is live across __syncthreads()
    foreach (tid in threads) {
        int i = threadIdx.y;
        int j = threadIdx.x;

        Out[index] = Shr[i][j]; // ERROR: only one definition of
        // 'index' reaches this expression
    }
}
```
Scalar Expansion & Thread Loop Placement

```c
// transformed C code with thread loops
void kernel(
    float *Out,
    float *In)
{
    __shared__ float Shr[16][17];
    int *scalar_index = (int *)malloc(sizeof(int) * blockDim.x * blockDim.y);
    int index;

    foreach (tid in threads) {
        int i = threadIdx.y;
        int j = threadIdx.x;

        index = i * 16 + j;
        Shr[j][i] = In[index];

        scalar_index[tid] = index; // SPILL 'index' into array
    }

    __syncthreads();

    foreach (tid in threads) {
        int i = threadIdx.y;
        int j = threadIdx.x;

        index = scalar_index[tid]; // LOAD 'index' from array
        Out[index] = Shr[i][j];
    }

    free(scalar_index);
}
```
Performance Scaling

Multicore performance normalized to single-core
Demo