LLVM GPU code with NVPTX

Introduction

This documentation will cover the basics of how to generate a GPU code with LLVM. More specifically, we’re going to take a look at the NVPTX back-end.

CUDA (Compute Unified Device Architecture) is a programming model/language and a parallel computing platform created by Nvidia. Normally, the Nvidia CUDA Compiler (NVCC) is used to translate CUDA code into a Nvidia’s GPU intermediate language called PTX (Parallel Thread Execution), and the graphics driver contains a compiler which translates the PTX into a binary code which can then be run on the processing cores. LLVM supports the GPU programming concepts through its NVPTX back-end.

NVPTX

PTX, the intermediate language produced by compiling CUDA code with NVCC, defines two different types of functions: the device functions, that can be called only from the device code, and the kernel functions, that are callable also by the host code. Here, the device and host refer to the GPU and the machine controlling the GPU (CPU) respectively. The back-end will emit device functions by default, and, to declare a function to as a kernel, metadata should be attached to the metadata object.

Example of a kernel- and device-function, and the metadata

```c
define float @device_fun(float %x) {
  ...
}
define void @my_kernel(float* %ptr) {
  ...
  %ret = call float @device_fun(float %y)
  ...
}
!nvvm.annotations = !{!0}
!0 = metadata !{void (float*)* @my_kernel, metadata !"kernel", i32 1}
```

The last two lines declare the my_kernel function to actually be a kernel. The metadata is of the form !<metadata-index> = metadata !<function-ref>, metadata !"kernel", i32 1).

Address Spaces

Nvidia GPU devices usually have four types of memory:

- Global: Large, off-chip memory
- Shared: Small, on-chip memory shared among all threads in a CTA (co-operative thread array)
- Local: Per-thread, private memory
- Constant: Read-only memory shared across all threads

The different memory types are presented in LLVM IR by mapping them to explicit address space specifiers. The mapping is shown in the following table:

<table>
<thead>
<tr>
<th>Address Space</th>
<th>Memory Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Generic</td>
</tr>
<tr>
<td>1</td>
<td>Global</td>
</tr>
<tr>
<td>2</td>
<td>Internal Use</td>
</tr>
<tr>
<td>3</td>
<td>Shared</td>
</tr>
<tr>
<td>4</td>
<td>Constant</td>
</tr>
<tr>
<td>5</td>
<td>Local</td>
</tr>
</tbody>
</table>

Note the additional address space, "generic". This address space can represent addresses in any other address space (with a few exceptions), and allows users to write IR functions that can load/store memory using the same instructions. LLVM provides intrinsics to convert pointers between the generic and non-generic address spaces. For example, the following code defines an array @g, that is mapped to a global device memory:
Target Triple

The selection between 32/64-bit code generation and the driver-compiler interface to use is specified by using the LLVM triple class. The triple architecture can be one of nvptx (32-bit PTX) or nvptx64 (64-bit PTX). The operating system should be one of cuda or nvcl, which determines the interface used by the generated code to communicate with the driver. Most users will want to use cuda as the operating system, which makes the generated PTX compatible with the CUDA Driver API. The vendor should always be nvidia.

Example: 32-bit PTX for CUDA Driver API: nvptx-nvidia-cuda

Example: 64-bit PTX for CUDA Driver API: nvptx64-nvidia-cuda

Example LLVM IR (NVVM IR) to PTX

LLVM 3.7 documentation provides an useful tutorial for understanding the functionality of the LLVM PTX backend. The tutorial features a ptx kernel in LLVM IR format which is compiled to ptx format and a host program that is used to execute the ptx program. The kernel is a simple program that loads two floats from addresses, sums them up and stores the value to a third address. The code is standard LLVM IR with couple of special features:

llvm.nvvm.read.ptx.sreg.tid.x() is a NVPTX intrinsic for accessing the thread id. The function is declared in the beginning to mark it as external to this file. The @kernel function is annotated in the bottom using LLVM IR metadata syntax to allow the NVPTX backend to identify the function as ptx kernel. The metadata definition in the tutorial failed to compile with llvm 3.7 (for which the document is supposedly written) but by changing it to simpler one in the sample below we got useable results.

```llvm
target triple = "nvptx64-nvidia-cuda"

; Intrinsic to read X component of thread ID
declare i32 @llvm.nvvm.read.ptx.sreg.tid.x() readnone nounwind

define void @kernel(float addrspace(1)* %A,
                      float addrspace(1)* %B,
                      float addrspace(1)* %C) {
  entry:
    ; What is my ID?
    %id = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x() readnone nounwind

    ; Compute pointers into A, B, and C
    %ptrA = getelementptr float addrspace(1)* %A, i32 %id
    %ptrB = getelementptr float addrspace(1)* %B, i32 %id
    %ptrC = getelementptr float addrspace(1)* %C, i32 %id

    ; Read A, B
    %valA = load float addrspace(1)* %ptrA, align 4
    %valB = load float addrspace(1)* %ptrB, align 4

    ; Compute C = A + B
    %valC = fadd float %valA, %valB

    ; Store back to C
    store float %valC, float addrspace(1)* %ptrC, align 4

    ret void
}

!0 = !{void (float addrspace(1)*, float addrspace(1)*, float addrspace(1)*) @kernel, !"kernel", i32 1}
!nvvm.annotations = !{!0}
```
The kernel is compiled to the PTX format with the llc tool, invoking it for example with command `llc -mcpu=sm_20 kernel.ll -o kernel.ptx` where `-mcpu=sm_20` is the minimum required device compute capability. llc outputs

```plaintext
//
// Generated by LLVM NVPTX Back-End
//

.version 3.2
.target sm_20
.address_size 64
.globl   kernel
.entry kernel(
   .param .u64 kernel_param_0,
   .param .u64 kernel_param_1,
   .param .u64 kernel_param_2
)
{
   .reg .f32   %f<4>;
   .reg .f32   %f<4>;
   .reg .s32   %r<2>;
   .reg .s64   %rd<8>;

   ld.param.u64   %rd1, [kernel_param_0];
   ld.param.u64   %rd2, [kernel_param_1];
   mov.u32     %r1, %tid.x;
   ld.param.u64   %rd3, [kernel_param_2];
   mul.wide.s32   %rd4, %r1, 4;
   add.s64     %rd5, %rd1, %rd4;
   add.s64     %rd6, %rd2, %rd4;
   add.s64     %rd7, %rd3, %rd4;
   ld.global.f32   %f1, [%rd5];
   ld.global.f32   %f2, [%rd6];
   add.rn.f32   %f3, %f1, %f2;
   st.global.f32   [%rd7], %f3;
   ret;
}
```

in kernel.ptx file. The function named `kernel` starts at the `.entry`.

The kernel can now be executed using the CUDA driver api. The usage is similar to OpenCL which we explored in one of the exercises.

**NVCC Compilation Trajectory**

NVPTX is just one piece in a larger compiler machinery for compiling CUDA to executable binaries. The proprietary NVCC compiler takes .cu files and outputs executables which contain the host code as executable binary and the gpu code as ptx assembly. NVCC compiler uses a host compiler such as LLVM to process the C++ parts of the code and has internal tooling for handling the CUDA directives. The compilation is split in two parts where the host program is compiled in a straightforward manner using the host compiler and linked with the gpu code. The gpu compilation is more complicated.

In NVCC the gpu code is compiled using the host compiler (LLVM) to process the C++ code and proprietary cudafe (CUDA Front End) compiler to handle the cuda directives. NVPTX is used to compile the output of the frontend to .ptx. The ptx is packaged with the host program to a binary in non executable form. The ptx code is compiled to gpu executable by the gpu drivers.

http://llvm.org/docs/NVPTXUsage.html