Project 3: LLVM-Based Runtime Profiling for extracting sub-kernel level metrics **Fixing CUDAAdvisor**

Contributors

Name	Roll Number
Yash Patel	CS22BTECH11047
Siddhant S. Godbole	CS22BTECH11054

Introduction

Description

General-purpose GPUs have been widely utilized to accelerate parallel applications. Given a relatively complex programming model and fast architecture evolution, producing efficient GPU code is nontrivial. CUDAAdvisor is a profiling framework to guide code optimization in NVIDIA GPUs. CUDAAdvisor performs various fine-grained analyses based on the profiling results from GPU kernels, such as memory-level analysis (e.g., reuse distance and memory divergence), control flow analysis (e.g., branch divergence) and code-/data-centric debugging.

Compatibility

CUDAAdvisor, was published 8 years before.
It was made for LLVM 4 and CUDA 7 which is unavailable.
LLVM 4 dosen't support available CUDA; which means upgrading to higher version.
It requires the old pass manager, i.e. LLVM 14 or before;
more upgrade is difficult because of the way it uses its library with dependent passes.
LLVM 14 supports upto CUDA 11.5.
LLVM 4's stack trace is outdated in LLVM 14 which can't be properly replaced.
Other such Function were appropriately replaced.

Compilation

The Cmake command for llvm should have Dumps ON ,
NVPTX as target ,
Cuda ON ,
Set Cuda Compiler,
RTTI ON ,
CUDA Toolkit Directory,

Required files to make:

- proper pass in opt for Advisor.
- axpy.cu (input program)
- ansf.cu (Advisor's cuda library)
- calc.cpp (Advisor's calculation library)
- print.cpp (Advisor's printing library)
- common.h (constraints)
- types.h (types)

1. Makefile

Build Process

The build process involves several stages, including generating LLVM IR files, compiling CUDA programs, and linking the necessary components for both device and host code. Below is a part of the Makefile used for the build process:

```

native: $(SRC)
  nvcc $(DEBUG) $(OPTAPP) $(SRC) -o native -L. -lprint --gpu-
      architecture=$(SM) -rdc=true

clang: $(SRC)
  $(clang) $(DEBUG) -G $(OPTAPP) $(SRC) -o clang --cuda-gpu-
      arch=$(SM) -L/usr/local/cuda/lib64 -lcudart -ldl -lrt -
      pthread #-save-temps

device.bc instru: device.link.bc $(PASS)
  $(opt) -load $(PASS) $(INSTRU) < device.link.bc > device.bc

$(HOST_SO) : $(UPATH)/print.cpp  $(UPATH)/../common.h $(UPATH
    )/types.h $(UPATH)/calc.cpp
  $(clang) -c $(DEBUG) $(OPT) -Wall -D $(ANA_TASK) -fPIC -lm
      -fopenmp $(UPATH)/print.cpp -o $(UPATH)/print.o
  $(clang) $(UPATH)/print.o -shared -o $(HOST_SO)

sbsi: hosti.bc $(HOST_SO)
  $(clang) -DMD_MODE hosti.bc -S -emit-llvm -o sad.ll
  $(clang) sad.ll -o $(EXE) --cuda-path=$(cuda) -lcudart -lstdc++ -ldl -lrt -lm -pthread -L$(UPATH) -lprint -Wl,-rpath='$(UPATH)'

hosti.bc: hosttmp.bc  $(PASS)
  $(opt) -load $(PASS)  $(INSTRU) < hosttmp.bc > hosti.bc

host.o : host.bc
  $(clang) host.bc -c

host.bc: device.fatbin $(SRC)
  $(LLVM)/build/bin/clang++ $(OPT) -std=c++14 -c  -emit-llvm $(SRC) \
    --cuda-gpu-arch=$(SM) --cuda-path=$(cuda) \
    -I/usr/include/c++/10 -I/usr/include/x86_64-linux-gnu/c++/10 \
    -Xclang -fcuda-include-gpubinary -Xclang device.fatbin
  cp axpy.bc host.bc

device.ptx: device.bc
  llc device.bc -march=nvptx64 -mcpu=sm_86 -mattr=+ptx70 -filetype=asm -o device.ptx

device.o :  device.ptx
  ptxas --gpu-name sm_86  device.ptx -o device.o -g -v -maxrregcount=31

device.fatbin: device.o host.cui
  fatbinary --cuda -64 --create device.fatbin --image=profile=$(SM),file=device.o --image=profile=$(CP),file=device.ptx -link

device.clean.bc:  $(SRC)
  $(clang) $(DEBUG) $(OPTAPP) $(CFLAGS) -c --cuda-device-only --cuda-gpu-arch=$(SM) --cuda-path=$(cuda) -emit-llvm $(SRC) -o device.clean.bc
  $(clang) $(DEBUG) $(OPTAPP) $(CFLAGS) -c --cuda-device-only --cuda-gpu-arch=$(SM) --cuda-path=$(cuda) $(SRC) -S -emit-llvm -o device.clean.ll
```

2. Profiling

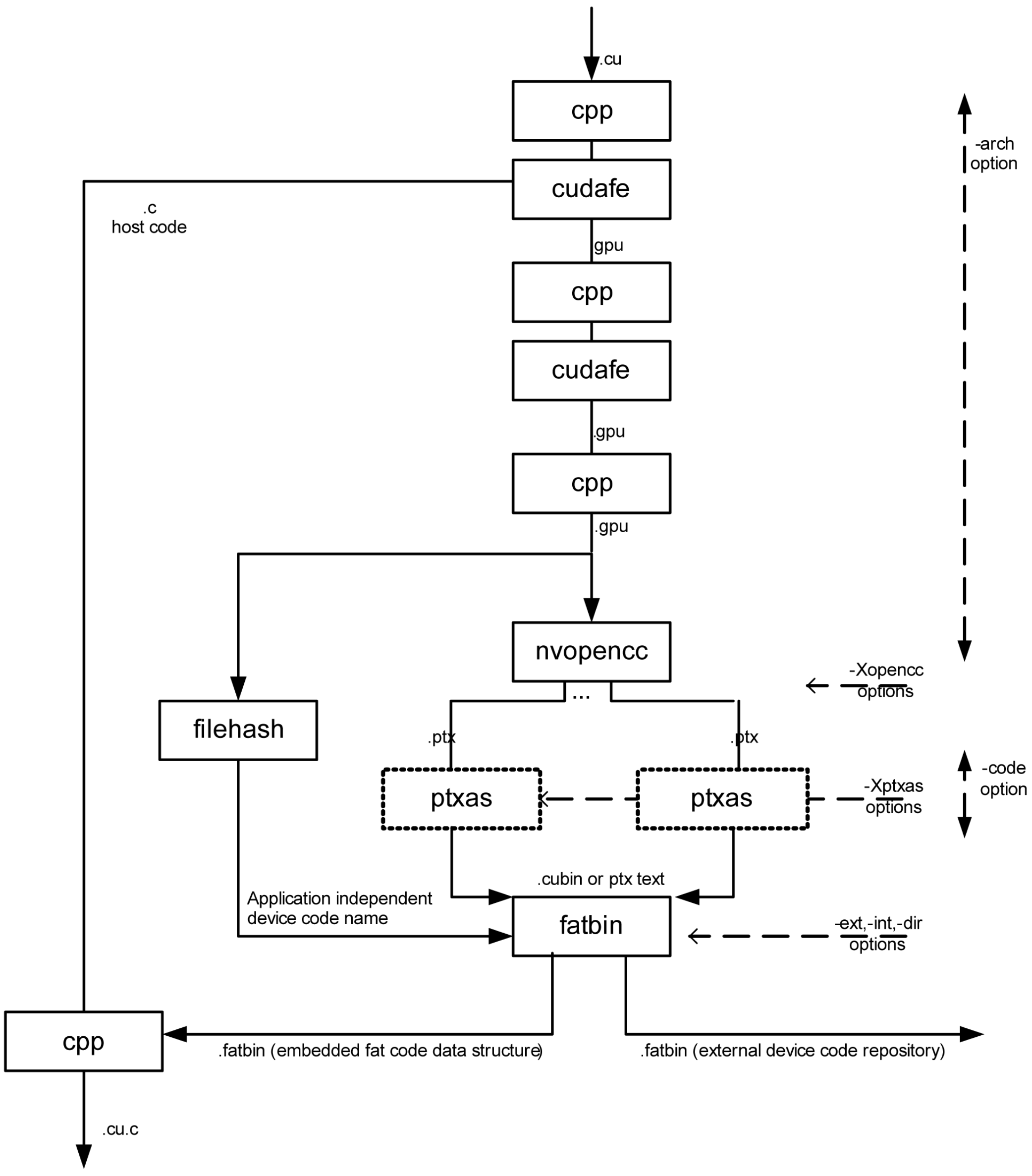


Figure 1: Example of Cuda compilation

2.1 Explanation

First we use clang –cuda-device-only to get the device side unlinked code.
Then we link it with the compiled ansf library.
Then we apply the opt pass.
Then we generate the ptx.
Then we get the Fatbinary.

Second we use clang –cuda-host-only to get the host side nulinked code.
Like abovve we get a host code which is unlinked from the device.

Third we link all required libraries and both files to generate an executable.

CUDAAdvisor Profiling

Profiling in Advisor works by adding calls to ”hooks” to get the runtime information. CUDAAdvisor’s profiler divides its task into two stages: (1) the data collection during the CUDA kernel execution, and (2) the data attribution at the end of each CUDA kernel instance.

CUDAAdvisor maintains a shadow stack to mirror the execution stack of each thread when the kernel runs on GPU. The profiler pushes the call site onto the shadow stack in the instrumented function at every call instruction, and pops the call site from the shadow stack in the instrumented function at every return instruction.
To scale the analysis, each GPU thread maintains its own shadow stack; the shadow stacks are also in GPU’s global memory. Upon kernel return, CUDAAdvisor copies all the data from GPU to CPU for further analysis.

On the CPU side, CUDAAdvisor maintains similar shadow stacks for CPU threads, which are used to determine the call stack for the invocation of each CUDA kernel.

Typically, a data object is allocated on the CPU side dynamically or statically. The profiler interprets the malloc family functions for dynamic allocation and reads the symbol table for static allocation. The profiler maintains a map that records the allocation call path for dynamic data objects and names for static data objects, and their allocated memory ranges. Similarly, CUDAAdvisor's profiler captures the data object allocation on the GPU side and keeps these data objects in another map. To correlate the two maps, CUDAAdvisor overloads the memcpy family functions and captures the two memory ranges involved in the memory copy.

With the two maps ready, CUDAAdvisor's profiler associates the memory accesses in GPU kernels with the data allocation.

LLVM PStacks (Profiling Stacks) is a profiling mechanism used to collect stack traces in LLVM-based tools during program execution. It's not a core part of LLVM IR or compilation, but rather a runtime or instrumentation feature that's typically used to help analyze performance or behavior of code.

Most of the Advisor's passes are dependent on each other so it is important to run them in order.

Added Profiling

- Limiting to lines. Added a condition calling StoreLines only on specified line range if specified. This line range can be given through the command line.
- ReportMemoryBandwidth. A pass which uses pre-existing passes to get bandwidth. Uses the Advisor instru-kernel-memory pass to get memory accesses; Uses this to get the bandwidth.
- ReportFlops. A pass which reports the Floating point Operations. Adds a function call after such operation.

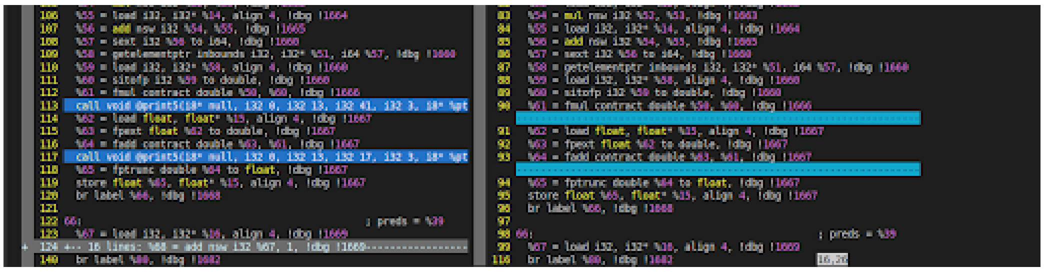
3. Individual Contributions

Siddhant S. Godbole (CS22BTECH11054)

- Attempted initial setup of CUDAAdvisor on both LLVM-6 and LLVM-18 to evaluate compatibility issues.
- Successfully debugged CUDAAdvisor to run on LLVM-14, resolving various version conflicts and deprecated function calls.
- Implemented logic to limit pass execution to a specified range of source lines, enhancing precision of instrumentation.
- Debugged multiple profiling passes, including the memory bandwidth instrumentation, and ensured correct insertion of hooks.
- Worked on proper linking and configuration of kernel binaries to enable execution with the CUDAAdvisor runtime.

Yash Patel (CS22BTECH11047)

- Successfully set up and compiled CUDAAdvisor within the LLVM-14 environment with appropriate CMake configurations.
- Implemented a new LLVM pass for profiling GPU memory bandwidth, which involved identifying memory instructions and collecting size information.
- Attempted to extend profiling capabilities by adding a new metric to count floating point instructions (FLOPs) within CUDA kernels. [Did not work because of old printintg.]



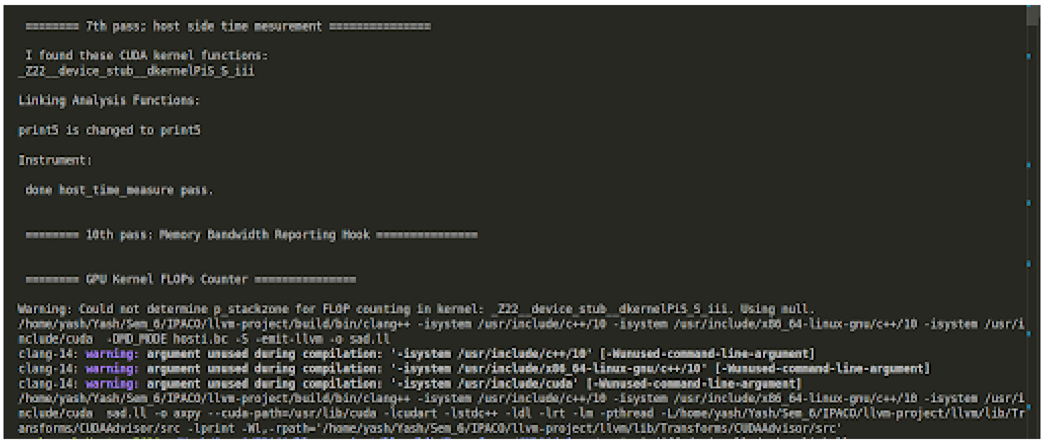
```
  %55 = load i32, i32* %14, align 4, !dbg !1664
  %56 = add nsw i32 %54, %55, !dbg !1665
  %57 = sext i32 %56 to i64, !dbg !1660
  %58 = getelementptr inbounds i32, i32* %51, i64 %57, !dbg !1660
  %59 = load i32, i32* %58, align 4, !dbg !1660
  %60 = sitofp i32 %59 to double, !dbg !1660
  %61 = fmul contract double %50, %60, !dbg !1666
  call void @print5(i8* null, i32 0, i32 13, i32 41, i32 3, i8* %pt
  %62 = load float, float* %15, align 4, !dbg !1667
  %63 = fpext float %62 to double, !dbg !1667
  %64 = fadd contract double %63, %61, !dbg !1667
  call void @print5(i8* null, i32 0, i32 13, i32 17, i32 3, i8* %pt
  %65 = fptrunc double %64 to float, !dbg !1667
  store float %65, float* %15, align 4, !dbg !1667
  br label %66, !dbg !1668

66:                                               ; preds = %39
  %67 = load i32, i32* %16, align 4, !dbg !1669
+--- 16 lines: %68 = add nsw i32 %67, 1, !dbg !1669-----------------
  br label %80, !dbg !1682
```

```
  %54 = mul nsw i32 %52, %53, !dbg !1663
  %55 = load i32, i32* %14, align 4, !dbg !1664
  %56 = add nsw i32 %54, %55, !dbg !1665
  %57 = sext i32 %56 to i64, !dbg !1660
  %58 = getelementptr inbounds i32, i32* %51, i64 %57, !dbg !1660
  %59 = load i32, i32* %58, align 4, !dbg !1660
  %60 = sitofp i32 %59 to double, !dbg !1660
  %61 = fmul contract double %50, %60, !dbg !1666
--------------------------------------------------------------------
  %62 = load float, float* %15, align 4, !dbg !1667
  %63 = fpext float %62 to double, !dbg !1667
  %64 = fadd contract double %63, %61, !dbg !1667
--------------------------------------------------------------------
  %65 = fptrunc double %64 to float, !dbg !1667
  store float %65, float* %15, align 4, !dbg !1667
  br label %66, !dbg !1668

66:                                               ; preds = %39
  %67 = load i32, i32* %16, align 4, !dbg !1669
  br label %80, !dbg !1682                        16,20
```

Figure 2: Required Instruction are inserted



```
 ======== 7th pass: host side time mesurement ================

 I found these CUDA kernel functions:
_Z22__device_stub__dkernelPiS_S_iii

Linking Analysis Functions:

print5 is changed to print5

Instrument:

 done host_time_measure pass.


 ======== 10th pass: Memory Bandwidth Reporting Hook ================


 ======== GPU Kernel FLOPs Counter ================

Warning: Could not determine p_stackzone for FLOP counting in kernel: _Z22__device_stub__dkernelPiS_S_iii. Using null.
/home/yash/Yash/Sem_6/IPACO/llvm-project/build/bin/clang++ -isystem /usr/include/c++/10 -isystem /usr/include/x86_64-linux-gnu/c++/10 -isystem /usr/i
nclude/cuda  -DMD_MODE hosti.bc -S -emit-llvm -o sad.ll
clang-14: warning: argument unused during compilation: '-isystem /usr/include/c++/10' [-Wunused-command-line-argument]
clang-14: warning: argument unused during compilation: '-isystem /usr/include/x86_64-linux-gnu/c++/10' [-Wunused-command-line-argument]
clang-14: warning: argument unused during compilation: '-isystem /usr/include/cuda' [-Wunused-command-line-argument]
/home/yash/Yash/Sem_6/IPACO/llvm-project/build/bin/clang++ -isystem /usr/include/c++/10 -isystem /usr/include/x86_64-linux-gnu/c++/10 -isystem /usr/i
nclude/cuda  sad.ll -o axpy --cuda-path=/usr/lib/cuda -lcudart -lstdc++ -ldl -lrt -lm -pthread -L/home/yash/Yash/Sem_6/IPACO/llvm-project/llvm/lib/Tr
ansforms/CUDAAdvisor/src -lprint -Wl,-rpath='/home/yash/Yash/Sem_6/IPACO/llvm-project/llvm/lib/Transforms/CUDAAdvisor/src'
```

Figure 3: But pstacks give warning