

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

## **Contributors**

<b>Name</b>	<b>Roll Number</b>
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. CUDAAAdvisor is a profiling framework to guide code optimization in NVIDIA GPUs. CUDAAAdvisor 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

CUDAAAdvisor, was published 8 years before.  
It was made for LLVM 4 and CUDA 7 which is unavailable.  
LLVM 4 doesn'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LLVM 4's stack trace is outdated in LLVM 14 which can't be properly replaced.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:

```
1 native: $(SRC)
2   nvcc $(DEBUG) $(OPTAPP) $(SRC) -o native -L. -lprint --gpu-
3     architecture=$(SM) -rdc=true
4
5 clang: $(SRC)
6   $(clang) $(DEBUG) -G $(OPTAPP) $(SRC) -o clang --cuda-gpu-
7     arch=$(SM) -L/usr/local/cuda/lib64 -lcudart -ldl -lrt -
8     pthread #-save-temps
9
10 device.bc instru: device.link.bc $(PASS)
11   $(opt) -load $(PASS) $(INSTRU) < device.link.bc > device.bc
12
13 $(HOST_SO) : $(UPATH)/print.cpp $(UPATH)/../common.h $(UPATH)
14   )/types.h $(UPATH)/calc.cpp
15   $(clang) -c $(DEBUG) $(OPT) -Wall -D $(ANA_TASK) -fPIC -lm
16   -fopenmp $(UPATH)/print.cpp -o $(UPATH)/print.o
17   $(clang) $(UPATH)/print.o -shared -o $(HOST_SO)
18
19 sbisi: hosti.bc $(HOST_SO)
20   $(clang) -DMD_MODE hosti.bc -S -emit-llvm -o sad.ll
```

---

```

17  $(clang) sad.ll -o $(EXE) --cuda-path=$(cuda) -lcudart -
    lstdc++ -ldl -lrt -lm -pthread -L$(UPATH) -lprint -Wl,-
    rpath='$(UPATH)'
18
19  hosti.bc: hosttmp.bc $(PASS)
20  $(opt) -load $(PASS) $(INSTRU) < hosttmp.bc > hosti.bc
21
22  host.o : host.bc
23  $(clang) host.bc -c
24
25  host.bc: device.fatbin $(SRC)
26  $(LLVM)/build/bin/clang++ $(OPT) -std=c++14 -c -emit-llvm
    $(SRC) \
27  --cuda-gpu-arch=$(SM) --cuda-path=$(cuda) \
28  -I/usr/include/c++/10 -I/usr/include/x86_64-linux-gnu/c
    ++/10 \
29  -Xclang -fcuda-include-gpubinary -Xclang device.fatbin
30  cp axpy.bc host.bc
31
32  device.ptx: device.bc
33  llc device.bc -march=nvptx64 -mcpu=sm_86 -mattr=+ptx70 -
    filetype=asm -o device.ptx
34
35  device.o : device.ptx
36  ptxas --gpu-name sm_86 device.ptx -o device.o -g -v -
    maxrregcount=31
37
38  device.fatbin: device.o host.cui
39  fatbinary --cuda -64 --create device.fatbin --image=profile
   =$(SM),file=device.o --image=profile=$(CP),file=device.
    ptx -link
40
41  device.clean.bc: $(SRC)
42  $(clang) $(DEBUG) $(OPTAPP) $(CFLAGS) -c --cuda-device-only
    --cuda-gpu-arch=$(SM) --cuda-path=$(cuda) -emit-llvm $(
    SRC) -o device.clean.bc
43  $(clang) $(DEBUG) $(OPTAPP) $(CFLAGS) -c --cuda-device-only
    --cuda-gpu-arch=$(SM) --cuda-path=$(cuda) $(SRC) -S -
    emit-llvm -o device.clean.ll

```

## 1.1 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 above we get a host code which is unlinked from the device.

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

## 2. Profiling

### 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.

## Added Profiling

Limiting to lines. Added a condition calling StoreLines only on specified line range if specified.

ReportMemoryBandwidth. A pass which uses pre-existing passes to get bandwidth.

ReportFlops. A pass which reports the Floating point Operations.