1.Our understanding :

CUDAAdvisor: Advanced GPU Profiling for CUDA Applications

- CUDAAdvisor is a sophisticated profiling framework designed to optimize CUDA code for modern NVIDIA GPUs. Unlike existing tools, it provides fine-grained analysis, supports multiple GPU architectures and CUDA versions, and offers actionable insights to improve performance.
- Built on the LLVM compiler infrastructure, CUDAAdvisor instruments both CPU and GPU code, enabling detailed profiling of performance bottlenecks such as memory divergence, branch divergence, and cache inefficiencies. This makes it particularly valuable for developers working on complex GPU applications in scientific computing, deep learning, and graph processing.

Why Choose CUDAAdvisor?

Fine-Grained Analysis

Provides instruction-level insights, unlike coarse-grained tools like NVProf that rely on high-level hardware counters

Extensibility

Built on open-source LLVM, allowing developers to add custom analyses for specific needs

Portability

Supports multiple NVIDIA architectures (Kepler, Pascal) and CUDA versions (7.0, 8.0), unlike SASSI which lacks CUDA 8.0 support

Optimization Guidance

Combines code- and data-centric profiling to pinpoint root causes and suggest fixes, achieving up to 2x speedup

CUDAAdvisor Architecture

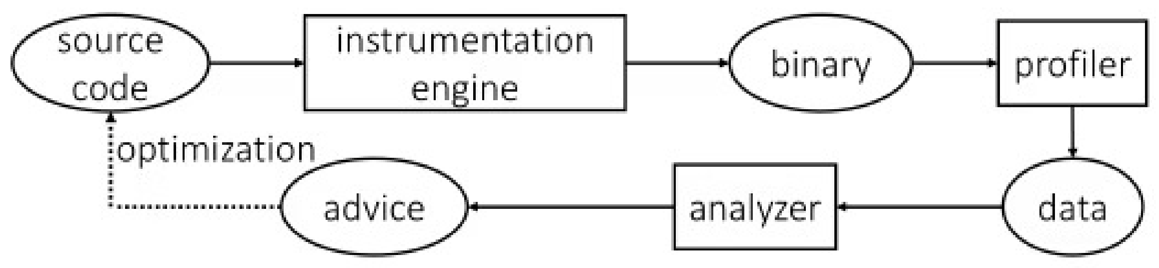


Figure 1. Workflow of CUDAAdvisor.

Instrumentation Engine

Built on the LLVM framework, it instruments CUDA code at the bitcode level for both CPU and GPU. Tracks function calls, memory allocations, and CPU-GPU data transfers, with support for custom analyses.

Profiler

Collects performance data during kernel execution using code-centric profiling (tracking call paths) and data-centric profiling (tracking data flow from CPU to GPU). Data is stored in GPU memory and copied to CPU for analysis.

Analyzer

Processes profiling data to generate optimization advice through online analysis after each kernel execution and offline aggregation across kernel instances. Outputs source code correlations to guide developers.

Current Limitations

Kernel-Level Granularity

Metrics are collected for entire kernels, diluting insights for specific regions in large kernels

Source Code Requirement

Needs source code and recompilation, which can be restrictive in some environments

High Overhead

Instrumentation causes 10-120x slowdown, disrupting timing-sensitive applications

No Custom Metrics

Lacks support for user-defined metrics like memory bandwidth, limiting domain-specific analysis

Additional limitations include the inability to analyze register usage due to NVIDIA's closed assembly layer, and dependency on LLVM's code generation rather than NVIDIA's nvcc. These constraints highlight the need for proposed extensions to make CUDAAdvisor more practical for real-world applications.

Proposed Solution: Region-Specific Profiling

Region-Specific Profiling

Define code regions, map to LLVM basic blocks

Custom Metrics Framework

Plugin system for user-defined metrics

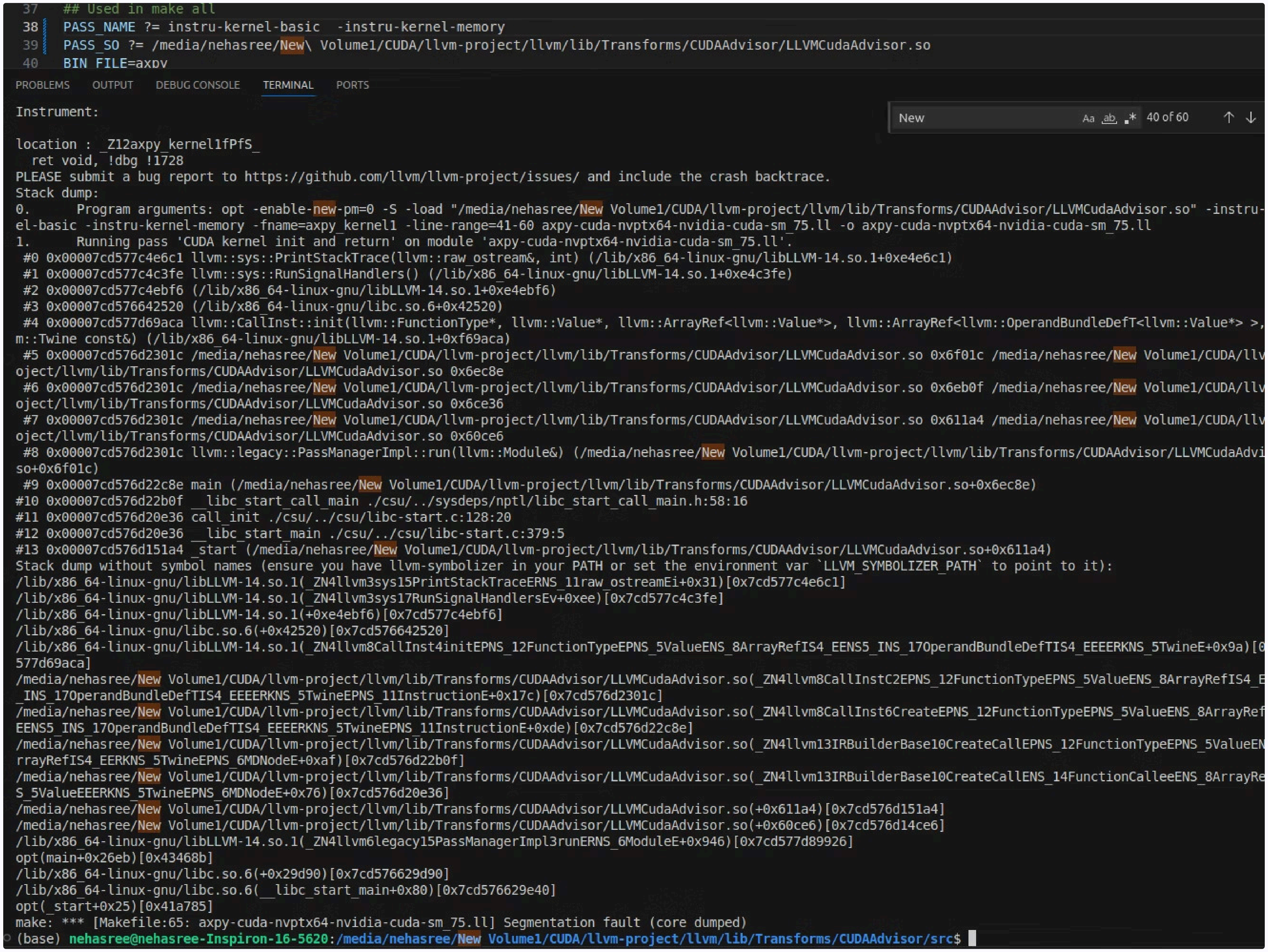
Overhead Reduction

Limit instrumentation to key regions

- The region-specific profiling allows users to define code regions (e.g., lines 22-50) , mapping these to LLVM basic blocks and only instrumenting those lines. This improves metric accuracy and reduces profiling overhead by limiting instrumentation to key regions, using lightweight instructions, batching CPU transfers, and performing heavy analysis offline.
- **Planned custom metrics** : Memory Bandwidth (bytes accessed per second), Compute Intensity (ratio of arithmetic to memory operations)
- **Additional metrics if time permits :** Cache Hit Rate, and potentially Warp Occupancy and Shared Memory Bank Conflicts if time permits.

Challenges faced with Cuda Advisor :

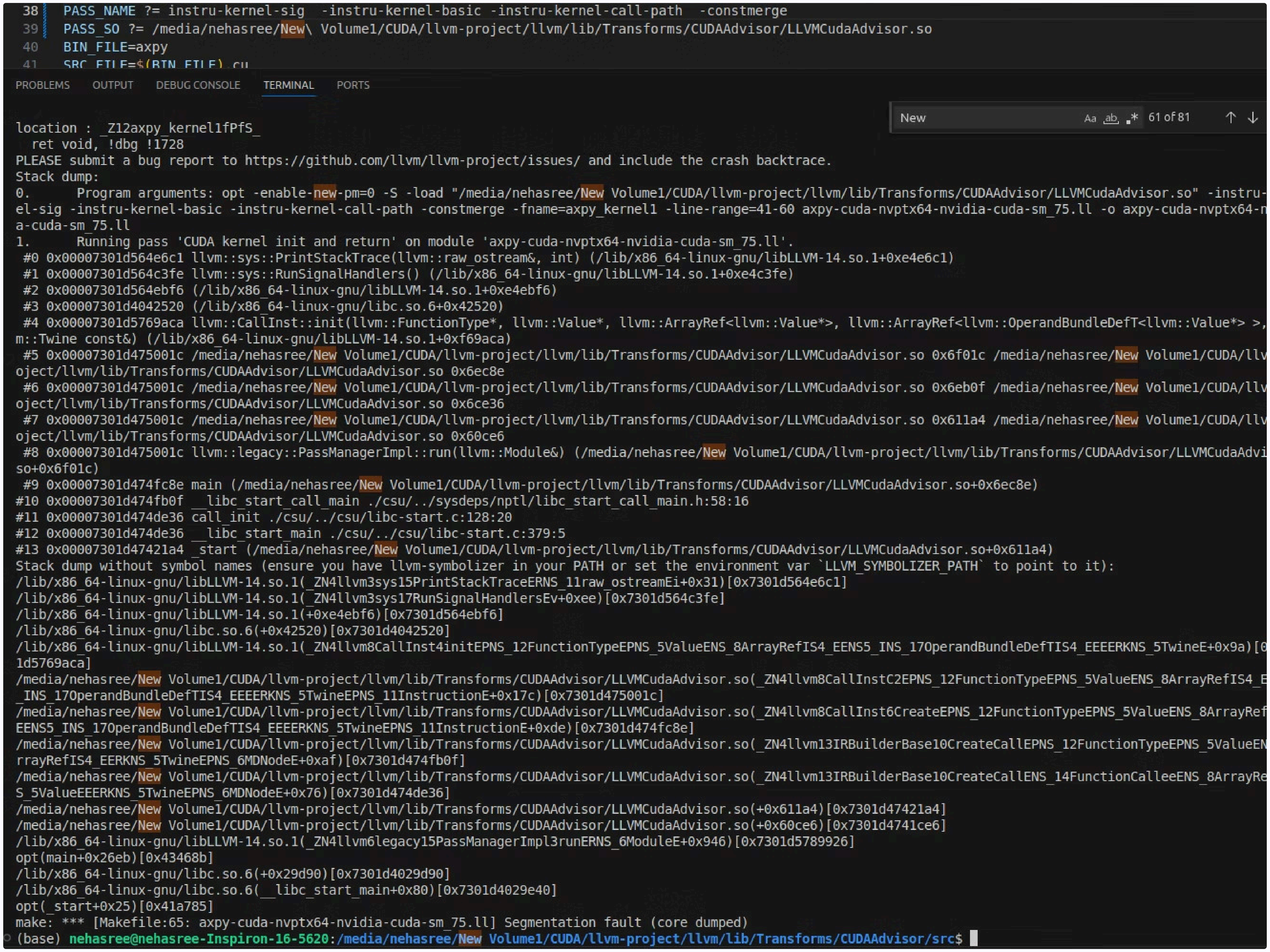
While running passes with cuda tool :



```
## Used in make all
PASS_NAME ?= instru-kernel-basic  -instru-kernel-memory
PASS_SO ?= /media/nehasree/New\ Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so
BIN_FILE=axpy
```

```
Instrument:

location : _Z12axpy_kernel1fPfS_
  ret void, !dbg !1728
PLEASE submit a bug report to https://github.com/llvm/llvm-project/issues/ and include the crash backtrace.
Stack dump:
0.      Program arguments: opt -enable-new-pm=0 -S -load "/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so" -instru-
el-basic -instru-kernel-memory -fname=axpy_kernel1 -line-range=41-60 axpy-cuda-nvptx64-nvidia-cuda-sm_75.ll -o axpy-cuda-nvptx64-nvidia-cuda-sm_75.ll
1.      Running pass 'CUDA kernel init and return' on module 'axpy-cuda-nvptx64-nvidia-cuda-sm_75.ll'.
 #0 0x00007cd577c4e6c1 llvm::sys::PrintStackTrace(llvm::raw_ostream&, int) (/lib/x86_64-linux-gnu/libLLVM-14.so.1+0xe4e6c1)
 #1 0x00007cd577c4c3fe llvm::sys::RunSignalHandlers() (/lib/x86_64-linux-gnu/libLLVM-14.so.1+0xe4c3fe)
 #2 0x00007cd577c4ebf6 (/lib/x86_64-linux-gnu/libLLVM-14.so.1+0xe4ebf6)
 #3 0x00007cd576642520 (/lib/x86_64-linux-gnu/libc.so.6+0x42520)
 #4 0x00007cd577d69aca llvm::CallInst::init(llvm::FunctionType*, llvm::Value*, llvm::ArrayRef<llvm::Value*>, llvm::ArrayRef<llvm::OperandBundleDefT<llvm::Value*> >,
m::Twine const&) (/lib/x86_64-linux-gnu/libLLVM-14.so.1+0xf69aca)
 #5 0x00007cd576d2301c /media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so 0x6f01c /media/nehasree/New Volume1/CUDA/llv
oject/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so 0x6ec8e
 #6 0x00007cd576d2301c /media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so 0x6eb0f /media/nehasree/New Volume1/CUDA/llv
oject/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so 0x6ce36
 #7 0x00007cd576d2301c /media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so 0x611a4 /media/nehasree/New Volume1/CUDA/llv
oject/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so 0x60ce6
 #8 0x00007cd576d2301c llvm::legacy::PassManagerImpl::run(llvm::Module&) (/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvi
so+0x6f01c)
 #9 0x00007cd576d22c8e main (/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so+0x6ec8e)
#10 0x00007cd576d22b0f __libc_start_call_main ./csu/../sysdeps/nptl/libc_start_call_main.h:58:16
#11 0x00007cd576d20e36 call_init ./csu/../csu/libc-start.c:128:20
#12 0x00007cd576d20e36 __libc_start_main ./csu/../csu/libc-start.c:379:5
#13 0x00007cd576d151a4 _start (/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so+0x611a4)
Stack dump without symbol names (ensure you have llvm-symbolizer in your PATH or set the environment var `LLVM_SYMBOLIZER_PATH` to point to it):
/lib/x86_64-linux-gnu/libLLVM-14.so.1(_ZN4llvm3sys15PrintStackTraceERNS_11raw_ostreamEi+0x31)[0x7cd577c4e6c1]
/lib/x86_64-linux-gnu/libLLVM-14.so.1(_ZN4llvm3sys17RunSignalHandlersEv+0xee)[0x7cd577c4c3fe]
/lib/x86_64-linux-gnu/libLLVM-14.so.1(+0xe4ebf6)[0x7cd577c4ebf6]
/lib/x86_64-linux-gnu/libc.so.6(+0x42520)[0x7cd576642520]
/lib/x86_64-linux-gnu/libLLVM-14.so.1(_ZN4llvm8CallInst4initEPNS_12FunctionTypeEPNS_5ValueENS_8ArrayRefIS4_EENS5_INS_17OperandBundleDefTIS4_EEEERKNS_5TwineE+0x9a)[0
577d69aca]
/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so(_ZN4llvm8CallInstC2EPNS_12FunctionTypeEPNS_5ValueENS_8ArrayRefIS4_E
_INS_17OperandBundleDefTIS4_EEEERKNS_5TwineEPNS_11InstructionE+0x17c)[0x7cd576d2301c]
/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so(_ZN4llvm8CallInst6CreateEPNS_12FunctionTypeEPNS_5ValueENS_8ArrayRef
EENS5_INS_17OperandBundleDefTIS4_EEEERKNS_5TwineEPNS_11InstructionE+0xde)[0x7cd576d22c8e]
/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so(_ZN4llvm13IRBuilderBase10CreateCallEPNS_12FunctionTypeEPNS_5ValueEN
rrayRefIS4_EERKNS_5TwineEPNS_6MDNodeE+0xaf)[0x7cd576d22b0f]
/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so(_ZN4llvm13IRBuilderBase10CreateCallENS_14FunctionCalleeENS_8ArrayRe
S_5ValueEEERKNS_5TwineEPNS_6MDNodeE+0x76)[0x7cd576d20e36]
/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so(+0x611a4)[0x7cd576d151a4]
/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so(+0x60ce6)[0x7cd576d14ce6]
/lib/x86_64-linux-gnu/libLLVM-14.so.1(_ZN4llvm6legacy15PassManagerImpl3runERNS_6ModuleE+0x946)[0x7cd577d89926]
opt(main+0x26eb)[0x43468b]
/lib/x86_64-linux-gnu/libc.so.6(+0x29d90)[0x7cd576629d90]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0x80)[0x7cd576629e40]
opt(_start+0x25)[0x41a785]
make: *** [Makefile:65: axpy-cuda-nvptx64-nvidia-cuda-sm_75.ll] Segmentation fault (core dumped)
(base) nehasree@nehasree-Inspiron-16-5620:/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/src$
```



```
PASS_NAME ?= instru-kernel-sig  -instru-kernel-basic -instru-kernel-call-path  -constmerge
PASS_SO ?= /media/nehasree/New\ Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so
BIN_FILE=axpy
SRC_FILE=$(BIN_FILE).cu
```

```
location : _Z12axpy_kernel1fPfS_
  ret void, !dbg !1728
PLEASE submit a bug report to https://github.com/llvm/llvm-project/issues/ and include the crash backtrace.
Stack dump:
0.      Program arguments: opt -enable-new-pm=0 -S -load "/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so" -instru-
el-sig -instru-kernel-basic -instru-kernel-call-path -constmerge -fname=axpy_kernel1 -line-range=41-60 axpy-cuda-nvptx64-nvidia-cuda-sm_75.ll -o axpy-cuda-nvptx64-n
a-cuda-sm_75.ll
1.      Running pass 'CUDA kernel init and return' on module 'axpy-cuda-nvptx64-nvidia-cuda-sm_75.ll'.
 #0 0x00007301d564e6c1 llvm::sys::PrintStackTrace(llvm::raw_ostream&, int) (/lib/x86_64-linux-gnu/libLLVM-14.so.1+0xe4e6c1)
 #1 0x00007301d564c3fe llvm::sys::RunSignalHandlers() (/lib/x86_64-linux-gnu/libLLVM-14.so.1+0xe4c3fe)
 #2 0x00007301d564ebf6 (/lib/x86_64-linux-gnu/libLLVM-14.so.1+0xe4ebf6)
 #3 0x00007301d4042520 (/lib/x86_64-linux-gnu/libc.so.6+0x42520)
 #4 0x00007301d5769aca llvm::CallInst::init(llvm::FunctionType*, llvm::Value*, llvm::ArrayRef<llvm::Value*>, llvm::ArrayRef<llvm::OperandBundleDefT<llvm::Value*> >,
m::Twine const&) (/lib/x86_64-linux-gnu/libLLVM-14.so.1+0xf69aca)
 #5 0x00007301d475001c /media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so 0x6f01c /media/nehasree/New Volume1/CUDA/llv
oject/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so 0x6ec8e
 #6 0x00007301d475001c /media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so 0x6eb0f /media/nehasree/New Volume1/CUDA/llv
oject/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so 0x6ce36
 #7 0x00007301d475001c /media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so 0x611a4 /media/nehasree/New Volume1/CUDA/llv
oject/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so 0x60ce6
 #8 0x00007301d475001c llvm::legacy::PassManagerImpl::run(llvm::Module&) (/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvi
so+0x6f01c)
 #9 0x00007301d474fc8e main (/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so+0x6ec8e)
#10 0x00007301d474fb0f __libc_start_call_main ./csu/../sysdeps/nptl/libc_start_call_main.h:58:16
#11 0x00007301d474de36 call_init ./csu/../csu/libc-start.c:128:20
#12 0x00007301d474de36 __libc_start_main ./csu/../csu/libc-start.c:379:5
#13 0x00007301d47421a4 _start (/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so+0x611a4)
Stack dump without symbol names (ensure you have llvm-symbolizer in your PATH or set the environment var `LLVM_SYMBOLIZER_PATH` to point to it):
/lib/x86_64-linux-gnu/libLLVM-14.so.1(_ZN4llvm3sys15PrintStackTraceERNS_11raw_ostreamEi+0x31)[0x7301d564e6c1]
/lib/x86_64-linux-gnu/libLLVM-14.so.1(_ZN4llvm3sys17RunSignalHandlersEv+0xee)[0x7301d564c3fe]
/lib/x86_64-linux-gnu/libLLVM-14.so.1(+0xe4ebf6)[0x7301d564ebf6]
/lib/x86_64-linux-gnu/libc.so.6(+0x42520)[0x7301d4042520]
/lib/x86_64-linux-gnu/libLLVM-14.so.1(_ZN4llvm8CallInst4initEPNS_12FunctionTypeEPNS_5ValueENS_8ArrayRefIS4_EENS5_INS_17OperandBundleDefTIS4_EEEERKNS_5TwineE+0x9a)[0
1d5769aca]
/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so(_ZN4llvm8CallInstC2EPNS_12FunctionTypeEPNS_5ValueENS_8ArrayRefIS4_E
_INS_17OperandBundleDefTIS4_EEEERKNS_5TwineEPNS_11InstructionE+0x17c)[0x7301d475001c]
/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so(_ZN4llvm8CallInst6CreateEPNS_12FunctionTypeEPNS_5ValueENS_8ArrayRef
EENS5_INS_17OperandBundleDefTIS4_EEEERKNS_5TwineEPNS_11InstructionE+0xde)[0x7301d474fc8e]
/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so(_ZN4llvm13IRBuilderBase10CreateCallEPNS_12FunctionTypeEPNS_5ValueEN
rrayRefIS4_EERKNS_5TwineEPNS_6MDNodeE+0xaf)[0x7301d474fb0f]
/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so(_ZN4llvm13IRBuilderBase10CreateCallENS_14FunctionCalleeENS_8ArrayRe
S_5ValueEEERKNS_5TwineEPNS_6MDNodeE+0x76)[0x7301d474de36]
/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so(+0x611a4)[0x7301d47421a4]
/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/LLVMCudaAdvisor.so(+0x60ce6)[0x7301d4741ce6]
/lib/x86_64-linux-gnu/libLLVM-14.so.1(_ZN4llvm6legacy15PassManagerImpl3runERNS_6ModuleE+0x946)[0x7301d5789926]
opt(main+0x26eb)[0x43468b]
/lib/x86_64-linux-gnu/libc.so.6(+0x29d90)[0x7301d4029d90]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0x80)[0x7301d4029e40]
opt(_start+0x25)[0x41a785]
make: *** [Makefile:65: axpy-cuda-nvptx64-nvidia-cuda-sm_75.ll] Segmentation fault (core dumped)
(base) nehasree@nehasree-Inspiron-16-5620:/media/nehasree/New Volume1/CUDA/llvm-project/llvm/lib/Transforms/CUDAAdvisor/src$
```

Another challenge :

CUDAAdvisor used LLVM-6, we tried upgrading the code to LLVM-14. We were still getting errors like ”ptxas fatal : Unresolved extern function 3 ’InitKernel’” for some cuda functions. We decided to write our own passes for our custom metrics as we are not able to resolve the errors in time

Compilation Process :

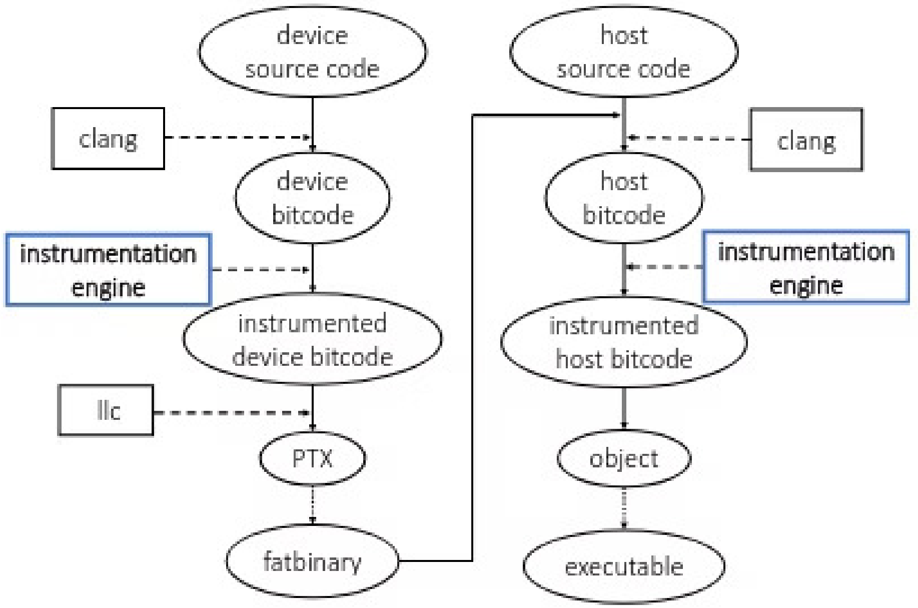


Figure 2. The workflow of the engine inserting instrumentation.

1

Host-Side Compilation

Generates **LLVM IR** for the **host code** only.

Output:

- LLVM IR for host code with embedded fatbin placeholder for device code.

2

Device-Side Compilation

Generate **device-side LLVM IR** for a specific GPU architecture (sm_75 etc.).

Output : LLVM IR for device functions.

3

Device IR Instrumentation

Run your custom LLVM pass to instrument the CUDA kernel.

Output:

Instrumented device LLVM IR with added hooks.

4

PTX Generation (Device Assembly)

Compile LLVM IR to **NVIDIA PTX** — an intermediate GPU assembly.

Output: Instrumented PTX code for NVIDIA GPUs

5

PTX Assembly to Object File

Assemble PTX into a GPU object file (.o) using NVIDIA's ptxas.

Output: axpy.ptx.o — Machine-level GPU object code.

6

Fatbinary Creation

Generate a **.fatbin** file that includes Object code and PTX code.

Output: Embedded device code bundle.

7

Device Linking

Link the device code and CUDA runtime to create a device-compatible object file.

Output: Device runtime linked object file.

8

Final Linking (Host + Device)

Link host object and device-linked object to generate final **executable**.

Output: Final binary with both host and GPU instrumented code.

What have we Implemented

1 Line Range Filtering

Only instruments instructions within specified source line ranges.

2 Metric 1 : Memory Bandwidth

3 Metric 2 : Compute Intensity

- The region-specific profiling allows users to define code regions (e.g., lines 22-50) , mapping these to LLVM basic blocks and only instrumenting those lines. This improves metric accuracy and reduces profiling overhead by limiting instrumentation to key regions, using lightweight instructions, batching CPU transfers, and performing heavy analysis offline.
- **Memory Bandwidth** shows how fast data is moved between global memory and compute units. If it's saturated, the kernel is **memory-bound**.
- **Compute Intensity** (operations per byte) indicates how much computation is done per memory access. Low intensity often means memory is the bottleneck; high intensity suggests it's compute-limited.

Line Range Filtering

- **Objective:** Instrument instructions in a specified line range, even if debug info is missing.
- **Challenge:** Not all LLVM-IR instructions have debug info.
- Need to approximate start and end line numbers for accurate instrumentation.
- **Solution:** Our approach attempts to approximate a plausible line range (start, end) for such instructions without debug info, by propagating line numbers from nearby instructions. This tells us that the current instruction lies somewhere between (start, end)

Definition of Line Ranges:

- For an instruction I without debug information: – The start line is approximated by propagating the minimum line number of its predecessors.
- The end line is approximated by propagating the maximum line number of its successors. •

Line Range - Algorithm Overview

Initialization:

We initialize two mappings: –

- **LineStart**: Maps each instruction to its start line number. If the instruction has debug information, it is initialized with that line number; otherwise, it is set to +∞ (INT MAX).
- **LineEnd**: Maps each instruction to its end line number. If the in- struction has debug information, it is initialized with that line number; otherwise, it is set to −1.

Computing LineStart: We use a worklist-based algorithm: –

- Instructions without debug information are added to a worklist.
  - For each instruction I in the worklist:
    1. If I is not the first instruction in its basic block, we take the line number of its previous instruction.
    2. If I is the first instruction, we look at the terminators of its predecessor basic blocks.
    3. The minimum line number among these predecessors is propagated to I
    4. If the line number for I changes, its successors are added to the worklist for further propagation.

Line Range - Algorithm Overview

Computing LineEnd: The computation for LineEnd follows a symmetric approach:

- Instructions without debug information are added to a worklist.
  - For each instruction I in the worklist:
    1. If I is not the last instruction in its basic block, we take the line number of its next instruction.
    2. If I is the last instruction, we look at the first instructions of its successor basic blocks.
    3. The maximum line number among these successors is propagated to I.
    4. If the line number for I changes, its predecessors are added to the worklist for further propagation.

Example :

- Basic Block B: I5 I6 I7
- Suppose: – I5 has debug info at line 20. – I6 lacks debug info. – I7 has debug info at line 30.
- After propagation: LineStart[I6] = 20, LineEnd[I6] = 30

Final Instrumentation Decision: If for an instruction the predicted line-range overlaps with input line-range, then the instruction is marked for instrumentation.

Metric 1 : Memory Bandwidth

Goal of the pass :

Instrument CUDA kernel functions to measure **runtime memory bandwidth**:

- Wraps **load** and **store** instructions with timers.
- Records:
  - **Memory access size (in bytes)**
  - **Latency (ns)** between start and end of memory access.

Key Features

- Operates **only on specified line ranges** within device functions.
- Uses **runtime hooks**:
  - getGpuTime() to get GPU clock timestamp.
  - recordMemAccess(size, duration) to record bandwidth data.

Function: runOnFunction()

1. **Extracts line range** from global string LineRange (e.g., "42-80").
2. Filters **only device functions** (F.getName() contains FunctionName).
3. Declares instrumentation hooks in the module
4. Calls findInstructionsToInstrument() to gather load/store instructions in that range.
5. For each instruction: If it's a load or store, calls instrumentMemoryOp() to instrument it.

Approach:

- **instrumentMemoryOp() Function:**
- Instruments individual memory instructions (load/store).
- Inserts:
  - getGpuTime() before the memory instruction.
  - getGpuTime() after the memory instruction.
  - recordMemAccess(bytes, duration) to log memory stats.
- **Device-Side Variables:**
- total_bytes_accessed: Tracks total bytes accessed on GPU.
- total_memory_time_ns: Tracks total time spent on memory accesses.
- Both updated using atomicAdd() for thread safety.
- **Timing and Logging Hooks:**
- getGpuTime(): Returns current GPU clock using clock64().
- recordMemAccess(bytes, cycles): Logs bytes accessed and time taken (in cycles).
- **Host-Side Summary (computeBandwidth):**
- Copies device counters to host using cudaMemcpyFromSymbol.
- Computes effective memory bandwidth in GB/s.
- Prints bandwidth, total bytes accessed, and total time in nanoseconds.

Metric 2 : Compute Intensity

Goal of the pass :

The pass and runtime system aims to **calculate compute intensity**—defined as the ratio of floating-point operations (FLOPs) to memory bytes accessed—in a CUDA kernel. This metric helps characterize whether a kernel is **compute-bound** or **memory-bound**.

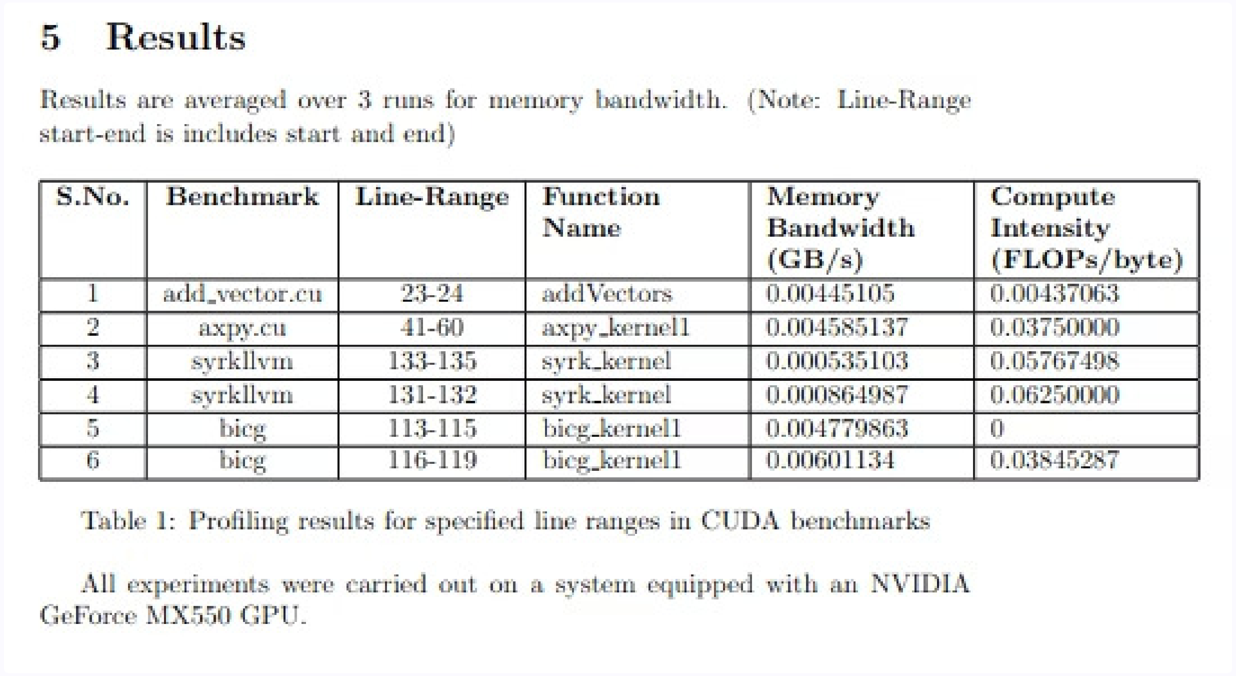
Key Concepts:

- **Compute Intensity = Total FLOPs / Total Bytes Accessed**
- **FLOPs**: Floating-point arithmetic instructions (e.g., FAdd, FMul)
- **Memory Access**: Bytes read or written from/to memory (via load, store)

Approach :

- **Line-Range Filtering:**
- Only instruments instructions within a user-specified source line range (e.g., `LineRange = "50-100"`).
- Enables focused profiling on relevant code sections.
- **FLOP Instrumentation:**
- Detects floating-point operations (e.g., `FAdd`, `FMul`).
- Inserts call to `recordFlop(1)` before each detected operation.
- **Memory Access Instrumentation:**
- Detects `load` and `store` instructions.
- Calculates accessed bytes using LLVM's `DataLayout`.
- Inserts call to `recordBytesAccess(bytes)` before each memory access.
- **CUDA Runtime Hooks (Device-Side):**
- `__device__ void recordFlop(unsigned long long count)`: Tracks FLOP count using atomic add.
- `__device__ void recordBytesAccess(unsigned long long bytes)`: Tracks memory bytes accessed using atomic add.
- **Host-Side Summary (printComputeIntensity):**
- Transfers `total_flops` and `total_bytes` from device to host.
- Computes and prints compute intensity = FLOPs / bytes.
- Resets counters for reuse in subsequent kernel runs.

Results :



5 Results

Results are averaged over 3 runs for memory bandwidth. (Note: Line-Range start-end is includes start and end)

S.No.	Benchmark	Line-Range	Function Name	Memory Bandwidth (GB/s)	Compute Intensity (FLOPs/byte)
1	add_vector.cu	23-24	addVectors	0.00445105	0.00437063
2	axpy.cu	41-60	axpy_kernel1	0.004585137	0.03750000
3	syrkllvm	133-135	syrk_kernel	0.000535103	0.05767498
4	syrkllvm	131-132	syrk_kernel	0.000864987	0.06250000
5	bicg	113-115	bicg_kernel1	0.004779863	0
6	bicg	116-119	bicg_kernel1	0.00601134	0.03845287

Table 1: Profiling results for specified line ranges in CUDA benchmarks

All experiments were carried out on a system equipped with an NVIDIA GeForce MX550 GPU.

Contributions

1. Bolla Nehasree - Implemented Memory Bandwidth custom metric
2. Nethi Keerthana - Implemented Compute Intensity custom metric
3. Nitya Bhamidipaty - Implemented the code to extract instructions for a specified line range within a CUDA kernel