# LLVM-Based Runtime Profiling for extracting sub-kernel level metrics - Final Report

Nethi Keerthana - CS22BTECH11043
Bolla Nehasree - CS22BTECH11012
Nitya Bhamidipaty - CS21BTECH11041

May 2025

## 1 Problem Statement

CUDAAdvisor is an open-source CUDA profiler that operates at the kernel level. We extend CUDAAdvisor to enable fine-grained profiling for specified line ranges within a kernel, given a function name and line range. Additionally, we introduce two custom metrics: Memory Bandwidth and Compute Intensity for more targeted performance analysis.

## 2 CUDAAdvisor Understanding

## 3 Approach

We were unable to update all the passes of CUDAAdvisor. Therefore, considering the time-constraint we implemented our own pass by using the CUDAAdvisor code as needed.

### 3.1 Compilation process

#### 3.1.1 Overview

The CUDA Advisor framework performs analysis and instrumentation of CUDA kernels using a combination of LLVM IR passes, custom runtime hooks, and device-host integration. The overall workflow consists of the following stages:

1. Compilation of host and device code to LLVM IR.
2. Instrumentation of device IR using a custom LLVM pass.
3. Generation of PTX and fatbinary objects.
4. Linking host and device binaries into a final executable.

### 3.1.2 Host-Side Compilation

The host code is compiled into LLVM IR using `clang++` with the following command:

```
$(CLANGPP) -stdlib=libstdc++ -Wall -Werror $(BIN_FILE).cu -march=x86-64 \
  --cuda-host-only -relocatable-pch -Xclang -fcuda-include-gpubinary \
  -Xclang $(BIN_FILE).fatbin -S -g -c -emit-llvm -isystem /usr/include/c++/10 \
  -isystem /usr/include/x86_64-linux-gnu/c++/10 -isystem /usr/include/c++/10/backward \
  -isystem /usr/lib/gcc/x86_64-linux-gnu/10/include -fPIC
```

**Important Flags:**

- `--cuda-host-only`: Compiles only the host code.
- `-Xclang -fcuda-include-gpubinary`: Embeds device fatbinary for CUDA runtime.
- `-S -emit-llvm`: Emits LLVM IR.
- `-g`: Includes debug information.
- `-fPIC`: Generates position-independent code for linking.

### 3.1.3 Device-Side Compilation

The device-side compilation to LLVM IR is performed using:

```
$(CLANGPP) -g -x cuda --cuda-device-only -emit-llvm -S \
  --cuda-gpu-arch=$(SM) -I/usr/local/cuda/include \
  -include __clang_cuda_runtime_wrapper.h -isystem /usr/include/c++/10 \
  -isystem /usr/include/x86_64-linux-gnu/c++/10 \
  -isystem /usr/include/c++/10/backward \
  -isystem /usr/lib/gcc/x86_64-linux-gnu/10/include \
  $(BIN_FILE).cu -o $(BIN_FILE)-cuda-nvptx64-nvidia-cuda-$(SM).ll
```

**Important Flags:**

- `--cuda-device-only`: Compiles only the device code.
- `--cuda-gpu-arch=`$(SM) : $Targets specified architecture$ (*e.g.*, `sm_75`).`-emit-llvm -S` : $Emits LLVM IR$.
- `-g`: Includes debug information.

### 3.1.4 Device Instrumentation

The device LLVM IR is instrumented using a custom LLVM pass:

```
$(OPTCC) -enable-new-pm=0 -S -load $(PASS_SO) -$(PASS_NAME) \
  -fname=axpy_kernel1 -line-range=41-60 \
  $(BIN_FILE)-cuda-nvptx64-nvidia-cuda-$(SM).ll \
  -o $(BIN_FILE)-cuda-nvptx64-nvidia-cuda-$(SM).ll
```

**Important Flags:**

- `-enable-new-pm=0`: Uses the legacy LLVM pass manager.

- `-load` $(PASS\_SO)$ : $Loads the shared library with the LLVM pass.$

#### 3.1.5 PTX Generation and Linking

**Convert Instrumented IR to PTX**

- ```
  llc -march=nvptx64 -mcpu=sm_75 \
    $(BIN_FILE)-cuda-nvptx64-nvidia-cuda-$(SM).ll \
    -o $(BIN_FILE).ptx
  ```

**Assemble PTX to Device Object**

```
ptxas $(BIN_FILE).ptx -o $(BIN_FILE).ptx.o -arch=sm_75
```

**Create Fatbinary**

```
fatbinary --64 --create $(BIN_FILE).fatbin \
  --image=profile=$(SM),file=$(BIN_FILE).ptx.o \
  --image=profile=$(CP),file=$(BIN_FILE).ptx -link
```

`fatbinary` combines the PTX and object code to produce a runtime-compatible binary blob.

**Device Linking**

```
nvcc -dlink -o $(BIN_FILE)_dlink.o $(BIN_FILE).fatbin \
  -lcudart -lcudart_static -lcudadevrt
```

#### 3.1.6 Final Linking

```
nvcc -o $(BIN_FILE) $(BIN_FILE).o $(BIN_FILE)_dlink.o -lc++
```

This links the host object and device-linked object into the final executable.

#### 3.1.7 Summary

- Separation of host and device compilation allows fine-grained control over instrumentation and optimization.
- Custom LLVM passes enable dynamic analysis of performance metrics like memory bandwidth and compute intensity.
- The modular workflow supports line-specific profiling, making it ideal for pinpointing performance bottlenecks in CUDA kernels.
```

### 3.2 Extracting instructions for a given Line Range

- In LLVM-IR, each instruction may have associated debug information (represented by `DILocation`), which provides the source code line number. However, not all instructions in LLVM-IR have debug information. For instrumentation, we need to associate each instruction with its corresponding source line range. Our approach attempts to approximate a plausible line range $(\text{start}, \text{end})$ for such instructions without debug info, by propagating line numbers from nearby instructions.
- **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.
- **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 $+\infty$ (INT_MAX).
  - `LineEnd`: Maps each instruction to its end line number. If the instruction 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.
- **Example for LineStart Propagation:**

  $$\text{Basic Block A:} \quad I_1 \quad I_2 \quad I_3 \quad I_4$$

  Suppose:
  - $I_1$ has debug info at line 10.

- $I_2$ and $I_3$ lack debug info.
- $I_4$ has debug info at line 15.

During propagation:

$$\text{LineStart}[I_2] \to 10, \quad \text{LineStart}[I_3] \to 10$$

since their predecessor $I_1$ has line 10.

- **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 for LineEnd Propagation:**

$$\text{Basic Block B:} \quad I_5 \quad I_6 \quad I_7$$

Suppose:

- $I_5$ has debug info at line 20.
- $I_6$ lacks debug info.
- $I_7$ has debug info at line 30.

During propagation:

$$\text{LineEnd}[I_6] \to 30$$

since its successor $I_7$ has line 30.

- **Final Instrumentation Decision:** After the propagation:
  - For each instruction without debug information, we have an approximate range $[start, end]$.
  - If this range overlaps with the specified range $[lineStart, lineEnd]$, the instruction is considered for instrumentation.

- **Output Format:** The pass displays which instructions are marked for instrumentation (useful for debugging). Each instruction is displayed with its approximated line range. If it overlaps with the target range, it is marked for instrumentation:

  $$\text{Instruction: } I \rightarrow \text{Line Range: } [start, end]$$

  If it falls in the target range:

  $$\text{Instruction to instrument: } I$$

## 3.3 Implementing Custom Metrics

### 3.3.1 Memory Bandwidth

**Goal of the Pass:** Instrument CUDA kernel functions to measure runtime memory bandwidth by wrapping load and store instructions with timers.

**Records:**

- Memory access size (in bytes)
- Latency (in nanoseconds) between the start and end of a memory access

**Key Features:**

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

**Function:** `runOnFunction()`

- Extracts the line range from the global string `LineRange` (e.g., `"42-80"`)
- Filters only device functions (`F.getName()` contains the target function name)
- Declares instrumentation hooks in the module
- Calls `findInstructionsToInstrument()` to gather relevant load/store instructions
- For each instruction:
  - If it is a load or store, calls `instrumentMemoryOp()` to instrument it

**Function:** `instrumentMemoryOp()`

- **Purpose:** Instruments a single memory access instruction
- **Steps:**

- Calculates bytes accessed using the data type size
- Inserts the following calls:
  - `getGpuTime()` before the instruction
  - `getGpuTime()` after the instruction
  - `recordMemAccess(bytes, duration)` to log statistics

**Device-Side Variables:**

```
__device__ unsigned long long total_bytes_accessed = 0;
__device__ unsigned long long total_memory_time_ns = 0;
```

These track total memory accessed (in bytes) and time spent (in nanoseconds) during memory operations. They are updated atomically to handle concurrent thread access.

**Timing and Logging Hooks:**

```
__device__ unsigned long long getGpuTime() {
    return clock64();
}
```

Returns the current GPU clock cycle counter using `clock64()`.

```
__device__ __noinline__ void recordMemAccess
bytes(unsigned long long bytes, unsigned long long cycles) {
    atomicAdd(&total_bytes_accessed, bytes);
    atomicAdd(&total_memory_time_ns, cycles);
}
```

Logs memory usage and timing using atomic operations. Marked `__noinline__` to prevent compiler inlining and ensure hooks are preserved.

**Host-Side Summary:**

```
void computeBandwidth() {
    unsigned long long bytes, time_ns;
    cudaMemcpyFromSymbol(&bytes, total_bytes_accessed, sizeof(unsigned long long));
    cudaMemcpyFromSymbol(&time_ns, total_memory_time_ns, sizeof(unsigned long long));

    double bandwidth = ((double)bytes / 1e9) / (time_ns * 1e-9);  // GB/s
    printf("Memory Bandwidth: %.8f GB/s\n", bandwidth);
}
```

This function copies counters from the device to the host and computes the effective bandwidth in GB/s. It prints the result, total bytes transferred, and total memory access time in nanoseconds.

### 3.3.2 Compute Intensity

**Objective:** Calculate *compute intensity*—the ratio of floating-point operations (FLOPs) to memory bytes accessed—in a CUDA kernel. This metric characterizes whether a kernel is compute-bound or memory-bound.

**Key Concepts:**

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

**LLVM Instrumentation Pass:** The `ComputeIntensityPass` performs the following:

- **Line-Range Filtering:**
  - Instruments only instructions within the user-defined line range (e.g., `"50-100"`)
- **FLOP Instrumentation:**
  - Detects floating-point operations (e.g., `FAdd`, `FMul`)
  - Inserts a call to `recordFlop(1)` before each detected FLOP
- **Memory Access Instrumentation:**
  - Detects load and store instructions
  - Computes bytes accessed using LLVM's `DataLayout`
  - Inserts `recordBytesAccess(bytes)` before each memory access

**Runtime Hooks (CUDA):**

```
__device__ void recordFlop(unsigned long long count);
__device__ void recordBytesAccess(unsigned long long bytes);
```

These functions use atomic operations to increment global counters: `total_flops` and `total_bytes`.

**Host-Side Summary:**

- The function `printComputeIntensity()`:
  - Transfers totals from device to host
  - Prints total FLOPs, total bytes, and compute intensity
  - Resets counters for reuse in subsequent kernel launches

**Benefits:**

- Dynamic and precise profiling at instruction level

- Works even on optimized code with partial debug information
- Enables fine-grained instrumentation by line range
- Helps guide performance tuning:
  - Low intensity → optimize memory access patterns
  - High intensity → exploit parallelism or FP throughput

# 4 Challenges Faced

1. CUDAAdvisor used LLVM-6, we tried upgrading the code to LLVM-14. We were still getting errors like "ptxas fatal : Unresolved extern function '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.

# 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.

# 6 Deliverables

- Code: GitHub Repository
- Final Report
- Final Presentation (PPT)

# 7 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