# A. Artifact Appendix

## A.1 Abstract

Our artifact includes a copy of our Assembler Generator, which takes CUDA benchmarks (executables) as input, and creates CUDA SASS assemblers (written in C++) as output. Our software and inputs expect a Linux environment, and for compilation they require GNU C/C++, Flex and Bison, and NVIDIA's CUDA Toolkit. There are no explicit hardware requirements.

For more detailed information on our artifact and its usage - including links for obtaining additional inputs beyond the selection of benchmarks included in our artifact - see the Artifact Appendix in which this abstract appears. Our artifact itself is available at https://github.com/decodecudabinary/Decoding-CUDA-Binary/tree/master/artifact

## A.2 Artifact check-list (meta-information)

- **Algorithm:** CUDA SASS Assembler Generation
- **Compilation:** Requirements: GNU C/C++; Flex and Bison; NVIDIA CUDA Toolkit
- **Data set:** Rodinia Benchmark Suite and CUDA SDK Code Samples
- **Run-time environment:** Linux; we used openSUSE and Ubuntu in our experimentation.
- **Output:** C++ code for assembling CUDA SASS assembly into binary
- **How much disk space required (approximately)?:** 25 Megabytes, or more with additional benchmarks/data.
- **How much time is needed to complete experiments (approximately)?:** Seconds or minutes, depending on the benchmarks included in the data set.
- **Publicly available?:** Yes; complete code will be made available at a later date.

## A.3 Description

### A.3.1 How delivered

We plan to share a more complete and well-documented version of our tools on GitHub at a later date. In the meantime, a functional copy of our Assembler Generator for this artifact is available at https://github.com/decodecudabinary/Decoding-CUDA-Binary/blob/master/artifact/Artifact_AssemblerGenerator.tar.gz

### A.3.2 Hardware dependencies

There are no hardware dependencies.

### A.3.3 Software dependencies

For full functionality, our tools require a Linux operating system; we have tested them on openSUSE and Ubuntu.

Software requirements include C/C++, Flex and Bison, and NVIDIA's CUDA Toolkit (https://developer.nvidia.com/cuda-downloads). We used version 6.5 of the CUDA Toolkit when preparing this artifact, however, we believe any version between v5.0 and the latest release (v10.0) should work.

### A.3.4 Data sets

Our data sets consist primarily of the Rodinia Benchmark Suite - http://lava.cs.virginia.edu/Rodinia/download_links.htm - and the CUDA SDK Code Samples - which can be installed as part of the CUDA Toolkit (https://developer.nvidia.com/cuda-downloads). Our artifact includes a small subset of these benchmarks, and a Bash script to compile them.

In our own experiments, we used all of the following benchmarks from Rodinia and the CUDA SDK: backprop, bfs, bicubicTexture, b+tree, cfd, dct8x8, dxtc, FDTD3d, gaussian, heartwall, hotspot, imageDenoising, interval, kmeans, lavaMD, leukocyte, lud, matrixMul, MC_SingleAsianOptionP, mummergpu, myocyte, nbody, nn, nw, particlefilter, particles, pathfinder, RAY, recursiveGaussian, srad, streamcluster.

The current version of our Assembler Generator expects benchmarks to target Compute Capability 3.x, 5.x, and/or 6.x devices only.

## A.4 Installation

Our Assembler Generator and each of the benchmarks is accompanied by a Makefile. To build each of them, run make inside each of their directories. For the provided subset of benchmarks, we include a Bash script that will compile all of them and place their executables into a single directory.

## A.5 Experiment workflow

GPU kernel functions are extracted from compiled benchmarks and fed into the Assembler Generator one-at-a-time for analysis; persistent data from analysis is passed through standard in and standard out. After processing each of the kernel functions, the Assembler Generator can be invoked again to perform one or more rounds of bit flipping in order to improve the results of its analysis: bit-flipped binary code is written into an executable, and then re-extracted with assembly and analyzed. Finally, the Assembler Generator is invoked once more to prepare CUDA SASS assemblers (written in C++) based on its analysis. Finally, the new assembler can be tested on one or more of the benchmarks to confirm it produces the same code as the original.

In other words, the basic experiment workflow steps are as follows: prepare benchmarks, extract kernel functions, analyze kernel functions, generate bit-flipped code, inject bit-flipped code into executable, extract bit-flipped kernel function, analyze bit-flipped kernel function, generate assembler code, assemble code into benchmarks, verify that benchmarks have not changed.

## A.6 Evaluation and expected result

We provide a Bash script, with filename procExes.sh, which performs all of the above Experimental Workflow steps except preparing the benchmarks (the CUDA executables used as input). Before running the script, all of the executables being processed for analysis should be placed in the **exes** subdirectory. The script will first place generated assemblers' code into files named generatedAssemblerXX.txt, where XX is the version number of the target architecture.

Using our provided subset of benchmarks with the given Makefiles and scripts, assembler code will be generated for Compute Capability 3.5. The script will proceed to insert the new Compute Capability 3.5 assembler into the actual source code for our **asm2bin** tool in the file binary35Gen.cpp, and test it on one of the benchmarks.

In order to delete the generated files and revert to our pre-existing assemblers, the cleanAll.sh script can be used. The assemblers we have already generated for Compute Capabilities 3.x and 5.x/6.x are in the files binary35.cpp and binary50.cpp, respectively.

## A.7 Experiment customization

Additional benchmarks can be used; simply place the desired CUDA executables inside the **exes** subdirectory before running our procExes.sh script.

The target architecture for the provided subset of benchmarks can be changed by modifying their Makefiles. For example, to use Compute Capability 5.0 code instead of 3.5, you can change the -arch=sm_35 flag to -arch=sm_50.

### A.8 Notes

When injecting assembled or bit-flipped binary code into an executable, our program modifies the GPU ELF according to the specifications we provided in our **File Format** appendix, available here: https://github.com/decodecudabinary/Decoding-CUDA-Binary/blob/master/artifact/Artifact_FileFormat.pdf

Though the currently provided copy of the Assembler Generator is only designed to provide assemblers, its analysis can also be used to create more human-readable descriptions of the ISA. In our appendices we provide a list of opcode encodings and a list of operands used by each instruction, and a list of special register encodings, available here: https://github.com/decodecudabinary/Decoding-CUDA-Binary/blob/master/artifact/Artifact_Encodings.pdf

When generating bit-flipped code, our Assembler Generator outputs it as a list of BINCODE instructions. BINCODE is not a real opcode used by NVIDIA, but rather a phony opcode we use in our own tools to indicate that the instruction contains only binary code. As such, the tool we use to inject the binary code must be capable of 'assembling' the BINCODE instruction, by simply returning the parsed binary. In fact, the version of our **asm2bin** tool provided with this artifact contains an up-to-date copy of our assemblers, capable of handling the phony BINCODE instruction in addition to the majority of actual CUDA SASS assembly instructions for compatible CUDA architectures.

We plan to upload a more complete, optimized, and well-documented copy of our Assembler Generator and related tools to GitHub at a later date, most likely prior to the conference at which this work will be presented.