Contents

1 Introduction

The Memtracer is a LLVM/Clang extension used to generate traces from CUDA applications. It is integrated into the compilation pipeline where it instruments CUDA code with tracing logic. Since the size of traces is unpredictable and they tend to be prohibitively large it is not feasible to store them in GPU memory. Therefore, during execution the trace data is copied to the host. Running the application will then produce one trace file per CUDA stream which can be used for analysis.

The goal of memory traces is to allow for deeper analysis of runtime effects and behaviour of (multi-)GPU applications, e.g. NUMA effects or Unified Memory behaviour. The gained insights might then lead to a better understanding of the given program, uncover performance bottlenecks, or enable better optimization etc.

The following tutorial provides a step-by-step guide on how to build LLVM/Clang with the Memtracer from source, how to build CUDA applications using the Memtracer, and how to analyze a trace. All examples use the "matrixMul" CUDA sample which is provided with the CUDA 8 toolkit.

2 Prerequisites

- Cuda 8 toolkit (`https://developer.nvidia.com/cuda-80-ga2-download-archive`)
- LLVM 7 requirements (`http://releases.llvm.org/7.0.1/docs/GettingStarted.html#requirements`)
- The Memtracer has only been tested on Linux 64bit.

2.1 Where to download the CUDA 8 toolkit?

You can find the CUDA 8 toolkit here:
`https://developer.nvidia.com/cuda-80-ga2-download-archive`.

2.2 How to extract the CUDA samples from the CUDA toolkit?

The samples can be separately extracted from the CUDA installer using the "samples", "samplespath", and "silent" flags, e.g.

```
sh cuda_8.0.61_375.26_linux.run --samples --samplespath=<myHome/
    cuda_samples> -silent
```

3 Building CUDA-Memtrace Step-by-Step

1. Download the LLVM 7.0 source:

```
cd <the folder you want llvm to reside in>
git clone http://github.com/llvm-mirror/llvm
git branch v7 origin/release_70
git checkout v7
```

2. Download Clang 7.0 source into LLVM's *tools* directory:

```
cd <folder containing llvm>/llvm/tools
git clone http://github.com/llvm-mirror/clang
cd clang
git branch v7 origin/release_70
git checkout v7
```

3. Download the CUDA-Memtrace source into the LLVM *tools* folder:

```
cd <folder containing llvm>/llvm/tools
git clone https://github.com/UniHD-CEG/cuda-memtrace
```

4. Add an entry for CUDA-Memtrace to LLVM's external project list
   - locate `CMakeLists.txt` inside LLVM's tools folder
   - include `add_llvm_external_project(cuda-memtrace)` in the list of external projects

```
add_llvm_external_project(lld)
add_llvm_external_project(lldb)
add_llvm_external_project(cuda-memtrace)

# Automatically add remaining sub-directories containing a '
    CMakeLists.txt'
```

5. Build LLVM

   5.1. Create a folder for your build:

```
cd <folder containing llvm>
mkdir build
cd build
```

   5.2. Run cmake (inside the newly created build folder)
   - the memtracer has to be built using shared libraries and with assertions enabled
   - to reduce build size/time we only build x86 and nvptx (to compile CUDA)
   - in case your CUDA installation is not located at `/usr/local/cuda` include `-DMEMTRACE_CUDA_FLAGS=<path to your CUDA installation>` in your `cmake` call

```
cmake -DLLVM_TARGETS_TO_BUILD="X86;NVPTX" -DBUILD_SHARED_LIBS=ON
     -DLLVM_ENABLE_ASSERTIONS=On -DCMAKE_BUILD_TYPE=Release -G
    Ninja ../llvm
```

5.3. Run Ninja

```
ninja
```

3.1 Possible errors when building:

Error: "add_llvm_loadable_module"
Solution: This error occurs when trying to build LLVM 8 or above with the Memtracer plug-in. Checkout LLVM 7 and Clang 7, as described above, before building.

Error: duplicate 'debug'
Solution: Build LLVM/Clang with dynamic libraries.

Error: "clang-7: error: cannot find CUDA installation. Provide its path via –cuda-path, or pass -nocudainc to build without CUDA includes."
Solution: Install the CUDA toolkit. If it is already installed ensure clang can find it.

Error: "clang-7: error: cannot find libdevice for sm_30. Provide path to different CUDA installation via –cuda-path, or pass -nocudalib to build without linking with libdevice."
Solution: Install a CUDA toolkit supporting compute capability 3.0.

4 Building Memory-Traced CUDA Applications

How to build CUDA applications using the Clang compiler in general is described in detail at https://prereleases.llvm.org/7.0.0/rc2/docs/CompileCudaWithLLVM.html.

There are some additional steps to compile a CUDA program with memory tracing. Firstly the plugin has to be enabled during compilation with Clang. Furthermore code has to be compiled using first level optimization ("-O1"). Assuming <build> is the path to your clang build your command line should look somewhat like this:

```
<build>/bin/clang++ -fplugin=<build>/lib/LLVMMemtrace.so --cuda-gpu-arch=
    sm_30  -O1 -c <file(s) to compile>
```

When linking your program you also need to link the Memtracer's host object file found at `<build>/lib/libmemtrace-host.o`. The thusly generated program will produce traces during runtime.

Instrumented code runs a lot slower and trace files tend to be very large. Therefore, depending on the type of application, the generation of traces might only be feasible for smaller problem sizes. Aside from this, there are a few technical restrictions: The Memtracer plug-in currently supports CUDA up to and including version 8. Furthermore Clang7 does not support texture memory. Due to the way it is instrumented by the Memtracer device code has to be free of recursion and dynamic parallelism. Since tracing logic is inserted at compile time, the use of pre-compiled libraries may lead to untraceable applications.

4.1 Example: Building the matrixMul CUDA sample using Clang with memory tracing

First we compile the source code in *matrixMul.cu* with memtracing enabled. As described above we use the **-fplugin** flag to enable memory tracing, specify a GPU architecture (also known as compute capability), use level one optimization, and include the CUDA sample's utility files:

```
<build>/bin/clang++ -O1 -fplugin=<build>/lib/LLVMMemtrace.so --cuda-gpu-arch=sm_30 -I<path_to_cuda_samples>/common/inc -o matrixMul.o -c matrixMul.cu
```

To link the newly created object file and all it's dependencies Clang needs several libraries (for more details see the Clang documentation linked above) including CUDA, as well as the memtracer's host code object file:

```
<build>/bin/clang++ -o matrixMul.out matrixMul.o <build>/lib/libmemtrace-host.o -lcudart_static -ldl -lrt -pthreads -L/usr/local/cuda-8.0/lib64
```

4.2 Example: Runtime increase and trace size of memory traced *matrixMul* CUDA sample

The matrixMul CUDA sample performs 301 iterations of a basic tiled matrix multiplication on a single GPU using shared memory. If not specified, the input matrices are of fixed size, 320x320 and 320x640 respectively, using 32-bit floating point numbers.

The matrixMul CUDA sample's measured average runtime per iteration on a NVIDIA K20 GPU increased from 5ms when using a non-traced version to over 500ms when using a traced one. The total runtime measured using Linux's time command increased from 200ms for the untraced version to over 3 minutes for the traced version.

The resulting trace is around 1.1GB in size.

5 The CUDA-Memtrace Trace File

A trace file generated by the CUDA-Memtrace tool contains information about the launched kernels of a single CUDA stream. If the program uses multiple CUDA streams multiple enumerated files are generated.

The default naming scheme of trace files is `<name of your program>-<CUDA stream ID>.trc`. To change this you can set the environmental variable MEMTRACE_PATTERN to a string of your choosing containing a '?', which is later exchanged for the stream id.

Trace files consist of kernel headers and records. A header contains the name of the kernel and a record is a tuple detailing the type of memory operation, the start of the operations first byte, a count of individual accesses, the size per access in bytes, a triple of the executing CTA's grid coordinates, and the ID of the physical SM the CTA was executed on.

Trace files are compressed by combining memory accesses of the same type and size by the same CTA on consecutive addresses into a single record. A record's count details how many individual records it contains.

An in-depth breakdown of headers and records is provided below.

6 Trace File Analysis

Since the traces contain detailed information about all memory accesses performed on the GPU(s) you can analyze the runtime behaviour in regards to memory of entire programs, their kernels, and the CTAs, etc. By analyzing accesses of different CTAs to the same memory addresses information about implicit communication can be extracted. As an example, data ownership can be assigned to the CTA that has last written to a location.

The cta-sets.py script, found inside the cuda-memtrace/tools folder, provides an example of how to read traces. It catalogs all kernel launches tagged with the application name and the launch order, as well as every memory access of all CTAs as a tuple of associated kernel id, coordinates in the grid, kind of memory access, and the range of memory accessed. The script stores the extracted information into a sqlite3 database (a lightweight SQL database).

6.1 Example: Analyzing a memory trace from the matrixMul CUDA sample.

First we use the *cta-sets.py* script located in the CUDA-memtracer's tools folder to extract initial data into a sqlite3 database. The script takes three inputs: a trace file, a tag, and the name of either the new database or an existing one. The data is then extracted from the trace file, is tagged, and stored into the specified database.

```
<cuda-memtrace folder>/tools/cta-sets.py matrixMul-0.trc aTag matrixMul-trace.db
```

Now we extract some information about the *matrixMul* sample by querying the database:

Number of kernel launches:

Query: `select count(*) from kernels (where tag = aTag);`
Result: `301`

As mentioned above the sample runs 301 iterations of the matrix multiply hence the 301 kernel launches.

Dimensions of the CTA grid

Query: `select count (distinct x), count (distinct y), count (distinct z) from ctas;`
Result: `20 | 10 | 1`

A grid of 20x10x1 contains 200 CTAs. The resulting matrix is of size 320x640, which is 204800 elements, divided by 200 yields 1024 elements per CTA. That is exactly the maximum number of threads per single CTA. We also can deduce the tile dimensions as 32x32.

Number of loads/stores per CTA of a specific kernel launch

Query: `select sum((stop - start) / 4) from ctas where kernel = 1 and kind = 0;`
Result: `4096000`

By subtracting the starting address from the ending address the amount of bytes read is calculated. By further dividing by sizeof(float) (=4) the number of floats loaded is calculated. The return value is the total number of floats loaded during the kernel's execution. Dividing this value by the number of CTAs per kernel yields the amount of loads per CTA: $4096000/200 = 20480$. Since each CTA loads 32 lines and 32 columns (see tile size above) into shared memory this is exactly the expected value: $32*320+32*320 = 20480$.

Query: `select sum((stop - start) / 4) from ctas where kernel = 1 and kind = 1;`
Result: `204800`

The calculation is the same as before but this time we queried for write accesses. Dividing by the number of CTAs we find every CTA to write 1024 floats which is the number of elements calculated per CTA.

7 The Byte Structure of a Trace File

Every trace file starts with a 10 byte magic number (ASCII for `.CUDATRACE`) which identifies it as a CUDA trace file. The rest of the trace file is either a kernel header or a record. Every kernel launch produces a header in the trace file, which is followed by all records of that kernel's memory accesses. Headers are signalled by "0x00", followed by a 2 Byte word indicating the length of the kernel's name, followed by the specified number of bytes containing the name in ASCII.
A record is either compressed or uncompressed with multiple accesses of the same kind by the same CTA to consecutive addresses, compressed into a single record by extending an uncompressed record of the initial access by the count of consecutive accesses. An uncompressed record is signalled by `0xFF` followed by 24 bytes of data (totalling 25 bytes per uncompressed record) and a compressed record is signalled by `0xFE` followed by 26 bytes of data (totalling 27 bytes per compressed record).
A breakdown of those data structures follows:

Kernel Header	
1 Byte	Identifier "0x00"
2 Bytes	length L of the kernel name
L Bytes	kernel name

Uncompressed Record	
1 Byte	Identifier `0xFF`
4 Bytes	ID of the Streaming Multiprocessor
4 Bytes	type of the access (initial 4 bits) and size of the access (remaining 28 bits)
8 Bytes	the address accessed
4 Bytes	CTA Id x
2 Bytes	CTA Id y
2 Bytes	CTA Id z

Compressed Record	
1 Byte	Identifier `0xFE`
4 Bytes	ID of the Streaming Multiprocessor
4 Bytes	type of the access (initial 4 bits) and size of the access (remaining 28 bits)
8 Bytes	the address accessed
4 Bytes	CTA Id x
2 Bytes	CTA Id y
2 Bytes	CTA Id z
2 Bytes	Count of consecutive accesses

7.1 Example: *matrixMul*'s Trace File

Reminder: x86 uses Little Endian (least significant byte first).

First 75 bytes of the trace file:

```
1A 43 55 44  41 54 52 41  43 45 00 23  00 5F 5A 31  33 6D 61 74
72 69 78 4D  75 6C 43 55  44 41 49 4C  69 33 32 45  45 76 50 66
53 30 5F 53  30 5F 69 69  FE 04 00 00  00 02 00 00  00 00 8C E0
96 39 7F 00  00 00 00 00  00 02 00 00  00 20 00
```

Trace file identifier:

`1A 43 55 44  41 54 52 41  43 45`	magic number (`.CUDATRACE` in ASCII)

The first kernel header:

```
      00 23  00 5F 5A 31  33 6D 61 74  72 69 78 4D  75 6C 43 55
44 41 49 4C  69 33 32 45  45 76 50 66  53 30 5F 53  30 5F 69 69
```

`00`	kernel header identifier
`23 00`	kernel name's length - 1: 36 = 35 + 1
`00 5F 5A 31  33 6D 61 74  72 69 78 4D`	kernel name
`75 6C 43 55  44 41 49 4C  69 33 32 45`	`_Z13matrixMulCUDAILi32EEvPfS0_S0_ii`
`45 76 50 66  53 30 5F 53  30 5F 69 69`	

The first record:

```
         FE  04 00 00 00  02 00 00 00  00 8C E0 96  39 7F 00 00
00 00 00 00  02 00 00 00  20 00
```

`FE`	compressed record signal
`00 00 00 02`	ID of the Streaming Multiprocessor: 2
`00 00 00 04`	type of access: 0 (= load), size of the access: 4 byte
`00 00 7F 39  96 E0 8C 00`	address accessed: `0x7fe996e08c00`
`00 00 00 02`	CTA Id x: 2
`00 00`	CTA Id y: 0
`00 00`	CTA Id z: 0
`00 20`	32 consecutive 4 byte loads : `0x7fe996e08c00 - 0x7fe996e08e00`