User Manual to Memory-Trace-Based GPU Code Generation Tool

Generation of GPU Code from memory trace takes several steps, the following is a step-by-step guide:

Step1: Instrument the original code so that the executable could dump memory access traces during execution. The memory trace is stored in memory.txt file.

Step2: Using Nested Loop Re-constructor (NLR) to reconstruct loop structures and memory accesses

Command to use: ./nlr -k 100 < memory.txt > step2.output

Step3: Manually collect data structures (arrays) information, including array base address, array size

Detail: Look at the original code and the step2.output, for each array, put its base address and its size (in bytes) in to a file (say step3.array-info). Each array occupies one line of the file. For example, the following shows what is contained in the file:

3219962056 8

167451136 264

167424928 8712

3075112992 287496

The above means there are four arrays, the base address of them are shown in the first column, the size in bytes are shown in the second column.

Step4: Mapping memory accesses to the correct data structure (array)

Command to use:

python ds-reconst.py step2.output step3.array-info > step4.output

After running the above command, all memory accesses have been mapped to the arrays that contain the respective elements. However, memory accesses are still in the form of the following:

167451488 + 8\*i0

i.e. the base address added with offset. The next step translates the above expression to C/C++ style arrayname[index].

Step5: Expressing memory accesses in C/C++ style

Command to use:

python post-processing-v4-nlr-output.py step4.output > step5.output.c

After running this command, the above expression would be translated to:

A167451488[ ( 8\*i0 ) / 8 ]

Besides, the left and right hand side expressions are also recognized.

Step6: Modify step5.output.c according to the original code to reflect correct computations (operators)

In step4.output, operators are by default “+”, for example:

A167433664 [( 264\*i0 + 8\*i1 ) / 8] =

A3075694624 [( 8712\*i0 + 264\*i1 + 8\*i2 ) / 8] +

A3075694624 [( 264 + 8712\*i0 + 264\*i1 + 8\*i2 ) / 8] +

A3075403808 [( 8 + 8712\*i0 + 264\*i1 + 8\*i2 ) / 8] +

A3075403808 [( 8712\*i0 + 264\*i1 + 8\*i2 ) / 8] ;

Look up the original code to change the above operators, for example, the resulting

A167433664 [( 264\*i0 + 8\*i1 ) / 8] =

A3075694624 [( 8712\*i0 + 264\*i1 + 8\*i2 ) / 8] -

A3075694624 [( 264 + 8712\*i0 + 264\*i1 + 8\*i2 ) / 8] +

A3075403808 [( 8 + 8712\*i0 + 264\*i1 + 8\*i2 ) / 8] -

A3075403808 [( 8712\*i0 + 264\*i1 + 8\*i2 ) / 8] ;

Array declarations/initialization code as well as other pieces that will make step4.output.c compilable and runnable should be manually added as well. For example: #include <stdio.h> … int main () {...} as well as I/O functions. Suppose the resulting code is step6.c

Step7: so far, step6 should be equivalent to the original code but is supposed to be easier to let polyhedral compilers optimize it.

Command to use:

pocc –target=acc-kernel –target=acc-parallel step6.c

The generated step6.pocc.c would be OpenACC version of the original code.

Step8: pass step6.pocc.c to OpenACC compilers to generate CUDA/OpenCL code.