A Guide to writing applications and compiling them for REDEFINE

May 4, 2017

This document describes how to write a C program targeting REDEFINE and compile it to REDEFINE target. The parallel IR document describes what is generated from an optimization pass. A developers' guide to porting known frontends onto IR maybe found in the same IR document.

1 C Program specifications

1.1 Musts

- Entry point to a REDEFINE kernel is defined as a function with signature *void redefine_start*. It doesn't take command line arguments. In general, functions can't take variable length arguments.
- Arguments to the kernel maybe specified as globals with prefix *redefine_in*. Outputs from the kernel must be marked as *redefine_out*. Globals which may need to be marked as both input and output need to be specified as *redefine_inout*.
- Data types are supported as long as they fit in data width of 32 bits. Floating point immediate values need to be labeled with suffix *f*. For example, 3.0 needs to be used as *3.0f*. Float data variable needn't be prefixed.
- Specifier *volatile* doesn't work. One needs to verify if keywords *register, auto* work. *static* variable is kept alive throughout the program.

- Special floating point operations such as sin, cos supported in REDEFINE ISA (refer to ISA document) need to be used with a prefix *_builtin*. For example, when invoking sin, use the call *_builtin_sinf* instead to replace with sin instruction of REDEFINE.
- Dynamic memory management calls (malloc, calloc, realloc, free) are not supported even with statically enumerated graphs and known allocation sizes.
- Inclusion of headers does not work. Every function call which is not a part of a cycle in the call graph (i.e., not a recursive call) gets inlined. One needs to check if this works across multiple files too (unlikely).
- I/O operations, signals and interrupts are not supported.
- Recursive nesting is not supported currently.

1.2 DOs

Most dos here are to ensure that the program does get parallelized. Try following the application skeletons committed in the code to ensure that the generated HIG gets parallelized appropriately.

- Recursive calls/affine loop nests are the sites for task parallelism. Try to write programs such that they mostly fit in either category. Non recursive calls get inlined and hence, modularity lasts only till IR.
- Try to pass arguments to functions by value. This ensures that the arguments are mapped onto the context frame of the HyperOp.
- Try to write functions such that the scalar arguments (i.e., integers, floating point values, char, unsigned etc) fit in a context frame (¡=16 arguments). Passing structs and other composite data types always results in data being passed through memory. (This is with the assumption that passing arguments through context frame is cheaper than through memory. However, in the current state of hardware, a remote write to context frame always results in reconcile, wasting clock cycles. Memory operations tend to be cheaper since only a last sync results in reconcile. Once the choice of reconcile is revisited, context frame writes become cheaper. This document assumes that it indeed is.)

- Try to write recursive calls such that the recursion tree is not very unbalanced. Try not to write degenerate (small) code in the non-recursive part of a function with recursion.
- Try to write loops such that there aren't any loop carried dependences or the loop carried dependences' distance is large.
- Try to write functions with fewer local variables.
- Ensure that the memory footprint of the application does not exceed the available memory. The compiler generates a warning at the last step in case of dynamically unfolding HIGs.
- Loops with large bounds show significant improvement in performance even if ILP window and degree of ILP offered is small. Try to ensure that loops are not partitioned into seperate HyperOps, take a quick look at the IR to ensure this.

1.3 DON'Ts

- Avoid using pointers and references as much as possible, be it global or local. Avoid passing pointers as arguments to functions.
- Avoid modifying globals across function calls since this serializes function calls. Try to follow dynamic single assignment as far as possible but within the allowed memory footprint. Globals without data distribution support lead to memory read/write bottlenecks and network bandwidth contention. Try to keep the kernel stateless.
- Avoid using scalars to maintain function states, they stay alive throughout the program.
- Avoid unbalanced control paths from same source node.
- Avoid using large sized globals that are not inputs or outputs. The reason is, these globals get initialized in the kernel code since REDEFINE does not support data segment and loading initial values onto it. The initialization thus adds to the overhead of execution.

2 Compiler options

2.1 Frontend

Clang frontend is used to generate LLVM IR from C Code. Use the target=redefine and -emit-llvm switches when invoking clang to generate IR.

2.2 HyperOp creation pass

HyperOp creation pass maybe invoked as an optimization pass using opt command by loading HyperOpCreationPass.so library to generate HIG in parallel IR format. By default, the pass merges returns of each function, inlines functions that are not in recursion and analyses dependences too. In order to run traditional optimizations on the generated HIG, an optimization pass is being written. Another pass that reverts to the older HIG is also being implemented currently. opt can take other optimization flags before HyperOpCreationPass library as per user's choice.

2.3 Low Level Compiler

llc pass maybe invoked for redefine target by using the option -mtriple=redefine. The other options available for the target can be seen by using the option -mattr=help along with llc. Various choices of CE count, number of rows and columns of fabric are available as listed in -mattr help and maybe used simultaneously by specifying them in a comma seperated manner. Additionally, WCET analysis maybe enabled or disabled with the option –mattr=+wcet. WCET analysis lists the estimated execution time of each HyperOp and the overall execution time of the kernel when the kernel is executed in an isolated manner.