RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Arthur Heimbrecht

Internship report

HD-KIP-TODOTODOTODO

KIRCHHOFF-INSTITUT FÜR PHYSIK

Internship report

As part of the human brain project BrainScaleS is a unique project on many levels. This includes a processor solely used by the HICANN DLS, which manages synaptic weights for every neuron built into one of the many wafers. To accelerate the speed at which this so called plasticity processor unit (PPU) computes all synaptic weights of every neuron used, the processor has an extended instruction set architecture (ISA) that supports vector registers and single input multiple data (SIMD). This report deals with the task of adding built-in functions to an existing backend of GCC, specifically the one used by the PPU, in order to extend the already implemented set of functions according to the users needs

Contents

1 Compiler Structure

Most Compilers that are used nowadays are built of three basic components which handle different steps in the process of converting human-readable programming language to machine-readable machine code. As does the GNU Compiler collection (GCC) which also can be seen as made of three main parts. The so called front-end, middle-end and back-end. All three parts work more or less independently from each other and communicate over a compiler-specific „language“, which is described a the Intermediate Representation (IR). It is typically never seen by the user and exists for a fact in many different forms [reference] one of which is Register Transfer Language (RTL), which is the lowest-level IR used by GCC. It is the most interesting IR when working on a back-end and will get more attention later in this report. But in before that we need to understand the structure an functioning of a compiler in general.

The first mentioned Front-end resembles the main interaction point between the human programmer and the machine. Front-ends are usually divided by their respective programming languages such as C, C++, Java… and have the main task of converting any programming language into unified IR, that can be passed to the middle-end in GIMPLE or GENERIC language. Therefore no matter which language you prefer, in an ideal case code, that is syntactically identical, should not differ after it is processed by the front-end. This is due to the goal of compilers such as GCC and LLVM to support as many languages and machines as reasonably possible while offering the equally good optimization and saving themselves overall work. It obvious that a single compiler for every combination of language and machine would simply not be practical, especially as the optimization taking place in the middle-end follows the same rules for pretty much any architecture. The middle-end main task is basically this sort of optimization, and makes for the main difference between compilers as most compilers offer the same range of front-ends and back-ends. (Part about the optimizations taking place in the middle-end). After all optimizations are through, the middle-end passes IR in form of RTL to the back-end. As you can see the middle-end rarely needs to be modified except for fundamental changes in the compilers architecture such as new kinds of optimizations and „multiple memory handling“ (also Harvard-Architecture (vielleicht)) The back-end is responsible for the final steps of the compilation process as it translates the general RTL IR into specific Assembly commands. It uses some sort of table of available assembly instructions, that is provided, and finds the best fitting instructions. GCC for example uses a Lisp-like language (is this RTL?) that uses something called insns. These combine different properties with the Assembly commands like an equivalent set of actions that are executed, the operands and the constraints it must satisfy. These will be further explained later. The back-end also contains the the code which implements processor-specific built-in functions. Depending on the Compiler architecture the back-end it finally emits IR to the assembler which emits the machine code in assembly or emits assembly itself. Then finally the linker links the assembly code of all program files together and substitutes the offset addresses with absolut addresses to generate the final machine code. reload register spilling register handling endianess wordsize

For the rest of this report we will deal more specifically with the combination of the C front-end and the RS/6000 back-end which is also used for the Power ISA which itself is the basis for the PPU. As exemplary built-in functions we will use the AltiVec vector extension for the rs/6000 architecture because the PPU has its own vector extension as well and ultimately this is meant to help extend the set of vector functions for the PPU.

2 Insn-Coding and Register Transfer Language

When implementing a new builtin function, it is usually necessary to start on the lowest, most abstract level there is in a back-end which is the machine description. It contains the definiton, expansion and splitting of insns. First it is important to understnd what an insn is. An insn is a single expression of which the RTL is composed. The insns themselfs are doubly linked and build a chain of instructions which is the RTL. It is usually easier to describe an insn equivalent to an instruction but insns can also be used as jump-labels for declaration of the code or dispatch tables for switch statements. but in our case we are most interested in using an insn as an instruction. In this case next all insn have a pattern which describes the side-effects of the insn/assembly command it is associated with. Here are two simple examples of such patterns: - load pattern - add pattern first you will probably have noticed the name of the insn which is mostly quite descriptive but optional if you do not want to use it as an builtin-function. As you can see next the name ist followed by a pattern which fist of all represents the effects and side effects of an insn. e.g. how many registers are used? which register will be changed whch remain the same etc. But it contains more information than just the side effects of an insn. the pattern also handles operands and constraints: Operands are not only the input or output of an insn they can also describe temporarily used registers or constants. in most cases though they either represent an input or output of an Assembly macro. Each operand is „matched“ by `match_operand` and must fulfill certain conditions. `(match_operand :m n predicate constraint)` m desribes a mode the operand must match such as SI (Single Integer), SF(Single Float) or vector modes such as V8HI (vector consisting of 8 Half Integer Elements). these modes can be bundled into groups such as all vector integer modes (VI) but must match each other for known instructions such as „add“. Next n is the operand number. It s chosen by the developer and stays the same for multiple uses of the same operand and helps identifiying the assmebly operand with the insn operand. The predicate is a string that further defines conditions an operand must fulfill such as being a constant, being in a certain range or being a certain vector type. It may also check if the operand is of a certain register type. all predicates are defined in predicates.md, where existing predicates can be altered or new ones can be added. The constraint finally does not help matching an operand but already gives certian rules for the register class the operand is going to have. Most people get to know constrains when working with Assembly code. The letters that identify with a certain register class such as r for general register, f for floating point register and m for memory operand are the same contraints that are used when defining insn. The next line of defining an insn is a more or less simple boolean statement that decides whether the insn is available or not. The easiest case is having a TARGET variable such as TARGET_ALTIVEC here. This means that if the AltiVec extension is activated (TARGET_ALTIVEC = true) the insn is available. The next line is calles the output template or output statement and states the assembly macro associated with the insn. you can see a „%“-sign followed byy the opernad index n for every argument the assembly macros accepts. The number and type of opernads for each assembly macro is stated in the opcode section of binutils. This can also be a piece of C code to decide between different assembly macros. At last the insn attribute completes the insn. it is optional but helps the compiler notice key effects a insn has, such as being a load r vector load instructon, an arithmetic or logical instruction etc. there is a wide variety of options here but only a few are needed in our case. All this information makes up just a single insn in a very long chain of equally complex insn that make up the RTL. But all of this information is needed for the compiler to first find the perfect insn to do certain tasks and second optimize the RTL code as good as possible. This why it is usually worth putting some effort into the RTL-like patterns of insns. it is importnat to notice that the assmebly macro is executing what was defined by the chip developer either way. The pattern is just meant to represent the effects of the assembly macro as good as possible. When the RTL is finally generated it mostly resembles those patterns of the insn defintion as you can see in the following example the loads two vectors from memory into vector regsiters, add the vectors and stores the result into the memory as a third vector. Because the RTL is recurring at different points when working with a back-end, here are a few basic examples of RTL and basic RTL „commands“ We now want to add our own insn which will later be used as a built-in function of the compiler. We choose to implement an imaginary built-in function that multiplies a vector v with a scalar number s. ... We now finished our very own insn which will later become a fully supported built-in function.

3 Adding a built-in type

To allow the compiler to differ which insns we acutally want to use as a built-in. The rs/6000 back-end has a special file which contians macros for every builtin function that will be available. These macros will be loadad in the rs6000.c file and also handled there. But before this can happen we have to create a built-in macro for our newly created insn. WE must first differ between the different kinds of macros. the rs/6000 back-end already provides us with templates for differnt altivec built-ins. These templates basically help identifying the right macros later on and avoid naming conflicts. Next we have to differ between the number of input opernad our built-in function is going to have and the number in th etemplate name is meant to be equal to th enumber of input operands . all Macros expect output operand except for the X-names macros. which are special cases we will emphasize later. We now choose BU_ALTIVEC_2 because our newest insn has to input operands and add the following lines:

the first argument is the name of the macro and will be used by the compiler internally. the user only gets interact with the second string which is the new built-ins name for the front end. (for altivec built-ins the tempalte adds __ builtin_ altivec_ in front of the given name). Next the attribute for the builtn must be given. typically this CONST but for builtins that load or store in the memory this is MEM for example. The last argument is the name of the insn we decided on earlier.

You can also add an „overload“ macro for your builtin as this allows for type checking and reducing builtins for different vector modes to just on final builtin. here we first need a Macro name and second the function name we want to use later (this functon name is preceeded by __ builtin_ vec_)

4 Argument handling

After all builtin macros and overloaded macros are initialized, we want to assign the argument types of our builtin functions. This is quite similar to the argument type in normal functions but looks quite differently. We actually need to mention every argument type that our builtin supports. This means we need to differ between signed and unsigned variables as well as different vector and integer sizes. Hence we must not forget any valid combination of output and input operands that could come up. To achive this the file rs6000-c.c build a connecton between our overloaded macros and the builtin macros we created earlier. Htis contains an array of altivec_builtin_types called altivec_overloaded_ builtins. the altivec_builtin_types struct consits of the macro codes from rs6000-builtins.def and 4 signed chars that are the return typ and up to three input types. since we only need to the third will always be zero. For every argument type available there is a macro defined that is named also named after the argument type. e.g. RS6000_BTI_V16QI for a signed 16 element quarter inetger vector, RS6000_BTI_unsigned_INTSI for an unsigned singel integer, ~RS6000_BTI_INTQI for a pointer adress of a signed quarter integer. Now we need to go through every combination of input and return types there is for our builtin. ...

5 Special Cases

not all cases can be handles by the aforementioned method. for example a builtin function wihout a return type. in htis case we use the BU_ ALTIVEC_ X template thaat slightly differs from other tmeplates. . . . . .

6 Nameing conventions

After we fully implemented our builtin functions we will finalize the functions by defining their final names. There are two naming conventions we can follow and typically it is advised to follow both in order to offer the user a set of alternatives. The first naming convention is the nameing convention already used by the AltiVec extension which starts every builtin function with vec_ and chooses a general descriptive name which is derived from the used Assembly macro if possible. This is the most intresting case for our exemplary Altivec builtin function. because there is already a vec_ mul builltin we will choose the descriptive abbreviation vec_ smul for scalar multiplication. The other naming convention gets intresting when extending the backend with another vector extension. If there were an S2PP vector extension, all assembly macros would begin with fxv which is a good alternative for the vec_ prefix altivec uses for its builtins. then it is a good idea so offer all assembly macros with their already known names e.g. fxvaddbm would become fxv_ addbm. for more genral builtins that combine the halfword and byte macro it is recommended to leave out the letter h or b likewise. rendering fxv_ addm the builtin function that works with both V8HI and V16QI vectors.

7 Discussion

Discussion...

8 Outlook

Outlook...

Appendix

Statement of Originality (Erklärung):

I certify that this thesis, and the research to which it refers, are the product of my own work. Any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, December 18, 2016

....
(signature)