Code Generator for Register Optimizations on GPUs

ACM Reference format:
. 2016. Code Generator for Register Optimizations on GPUs. 1, 1, Article 1 (January 2016), 3 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 GRAMMAR

The code generator presented here accepts as input a sequence of stencil statements that are expressed in a restricted subset of C consistent with the grammar described in figure 1. The input to the code generator is a full stencil code in a .cu file, with the regions that the user wants to reorder annotated with `#pragma begin stencil`... and a matching `#pragma end`. The parser is simplified and does not perform any syntactic verification, so when creating a new benchmark, it is the user's responsibility to ensure that the syntax adheres to the specifications for the code generator to work correctly.

In the grammar of figure 1, a *BaseExpr* in a statement represents an accessed storage location, and is identified by a *label*. A crucial part of the reordering framework is to identify the *same BaseExpr*. This is done by checking the equivalence of the labels that are assigned to any two *BaseExpr*. This is complicated by the fact that post unrolling, one has to guarantee that `A[i+1+1]` will be assigned the same label as `A[i+2]`. In general, the array index expressions can be rewritten in multiple, albeit equivalent ways – `A[i+j]` is the same as `A[j+i]`, `A[i+1+j]` is the same as `A[i+j+1]`, and so on. To ensure that the semantically equivalent *BaseExpr* are assigned the same label, the code generator has a mechanism that "canonicalizes" the accesses, so that semantically equivalent accesses have the same pattern. We define an array index expression to comprise of an iterator-access component (e.g., `(i+j)` in access `A[i+j+2]`), and an integer offset component (e.g., 2 in accesses `A[i+j+1+1]` and `A[i+j+2]`). Within the code generator, this logic is encoded in the function *array_access_info ()*, and it works in two steps:

(1) it separates out iterator-access component and the integer offset from an index expression
(2) it canonicalizes both the extracted components. For the iterator-access component, this involves identifying the presence of an operator in the iterator-access, and then exploiting the properties of the operator to align the iterators in lexicographical order. For example, `(j-i)` will be converted to `(j+(-i))`, and then `(j` *op* `i)` will be rearranged

2016. XXXX-XXXX/2016/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

⟨Program⟩	::	⟨StmtList⟩;
⟨StmtList⟩	::	⟨Stmt⟩⟨StmtList⟩ | ⟨Stmt⟩
⟨Stmt⟩	::	⟨BaseExpr⟩⟨AssignOp⟩⟨Expr⟩;
⟨Expr⟩	::	Side-effect-free expression of ⟨BaseExpr⟩
⟨BaseExpr⟩	::	⟨ID⟩ | ⟨ArrayExpr⟩
⟨ArrayExpr⟩	::	⟨ID⟩⟨ArrayDims⟩
⟨ArrayDims⟩	::	[⟨AccessExpr⟩]⟨ArrayDims⟩ | [⟨AccessExpr⟩]
⟨AccessExpr⟩	::	Side-effect-free expression of ⟨AccessElement⟩ with restrictions described in text
⟨AccessElement⟩	::	⟨Iterator⟩ | ⟨Const⟩
⟨AssignOp⟩	::	`=` | `+=` | `-=` | `*=` | ...
⟨DataType⟩	::	`double` | `float` | `int` | `bool` | ...

Fig. 1. Grammar for valid input statements to the code generator

as `(i op j)`, if `i` precedes `j` in lexicographical ordering, and *op* is commutative, like +,*. For the offset component, it identifies the operators involved, and performs the mathematical operation to reduce the offset component to a single integer value.

The canonicalization part is non-trivial, and currently primitive in implementation in the code generator. We currently restrict the number of operators in the iterator-access component to 1. For example, we allow A[k+1+1][2*j+2][i] since we can canonicalize each of the array index expressions, but not something as complex as A[(k+2)*(k+3)][j][i], since this access will result in an iterator-offset component (k*k+5k) where there are two arithmetic operations involved.

In summary, We impose the following restrictions on the statements:

- The statements must not modify the iterators (e.g., i, j and k in listing 1) in any way.
- The statements can only have mathematical functions like sine, sqrt, cosine, etc. on the RHS, that are side-effect free.
- The accessed locations in the stencil statements are either scalars or array elements. The array index expressions can only comprise the loop iterators, integers or simple arithmetic operations between them (+,-,*,/). We allow only *one* arithmetic operation in the iterator-access component of the array index expressions, and the offset component must be an integer.

Once the parsing is complete, the code generator lowers the input further to sub-statements consistent with the grammar of figure 2. Informally, this lowering is like GIMPLE IR of GCC (Stallman and DeveloperCommunity 2009) with accumulations.

2 CREATING AN EXAMPLE

Listing 1 shows a snippet of a cuda input code that can be passed as an input to the reordering framework. The computation in lines 5-7 is affine, and in compliance with the restrictions we

$$\begin{aligned}
\langle\text{CodeBody}\rangle &:: \langle\text{StmtList}_l\rangle\,; \\
\langle\text{StmtList}_l\rangle &:: \langle\text{Assignment}_l\rangle\,;\langle\text{StmtList}_l\rangle \mid \langle\text{Assignment}_l\rangle \\
\langle\text{Assignment}_l\rangle &:: \langle\text{BaseExpr}\rangle\langle\text{AssignOp}\rangle\langle\text{Expr}_l\rangle \\
\langle\text{Expr}_l\rangle &:: \langle\text{BaseExpr}\rangle\langle\text{BinaryOp}_l\rangle\langle\text{BaseExpr}\rangle \mid \langle\text{BaseExpr}\rangle \\
\langle\text{BinaryOp}_l\rangle &:: + \mid - \mid * \mid / \mid \,|\, \mid \& \mid \ldots
\end{aligned}$$

Fig. 2. Grammar for the lowered 3-address IR for instruction reordering

Listing 1. The input representation with pragmas

```
1    for (k=1; k<=N-2; k++) {
2         for (j=1; j<=N-2; j++) {
3              for (i=1; i<=N-2; i++) {
4  #pragma begin stencil1 unroll k=4,j=1,i=1
5                        out[k][j][i] = a*(in[k+1][j][i]) + b*(in[k][j-1][i] +
6                                    in[k][j][i-1] + in[k][j][i] + in[k][j][i+1] +
7                                        in[k][j+1][i]) + c*(in[k-1][j][i]);
8  #pragma end stencil1
9              }
10        }
11   }
```

placed on the input grammar. Note the pragmas in line 4 and 8 that surround the statement – these demarcate the region that the code generator can parse and optimize. Additionally, the pragma supplies the unrolling factor along each dimension. Given an input file with several such pragma-demarcated regions, the steps in optimizing the code will be:

(1) A script will remove the statements between all the pragma, and put them into separate `.idsl` files, each named based on the stencil name (e.g., `stencil1.idsl` for the example in listing 1). Therefore, each stencil name must be distinct.
(2) From the pool of `stencilname.idsl` files generated, the code generator will pick one at a time, parse it, optimize it, and write the results back in `stencilname.cu`.
(3) A script will then read the contents from `stencilname.cu`, and replace the original statements demarcated by `#pragma begin stencilname ... #pragma end stencilname` with the optimized code.

The changes are made in-place to the input cuda file. Please look at the benchmarks in the examples folder to learn more about how to use this code generator for your applications.

REFERENCES

Richard M. Stallman and GCC DeveloperCommunity. 2009. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Paramount, CA.