Research Journal of Applied Sciences, Engineering and Technology 8(1): 64-68, 2014
DOI:10.19026/rjaset.8.941
ISSN: 2040-7459; e-ISSN: 2040-7467
© 2014 Maxwell Scientific Publication Corp.
Submitted: February 22, 2014 Accepted: April 09, 2014 Published: July 05, 2014

Research Article

The Optimization of IF-conversion in Whole Function Vectorization

Jianmin Pang, Feng Yue, Zheng Shan, Chao Dai and Jiuzhen Jin
State Key Laboratory of Mathematical Engineering and Advanced Computing,
Zhengzhou, 450001, China

Abstract: In order to get better performance, lots of optimization methods are used in code transformation. When migrating SPMD to multi-core platform, vectorization is one key optimization to improve performance. Control flow is the main challenge for vectorization and IF-conversion is usually used to transform control flow into data flow. In most researches, after IF-conversion both the two branch vector codes have to be executed even the predications in scalar lane for one branch are all false. This study proposes code bypass technology to improve this situation in whole function vectorization of SSA from. The region of consecutive instructions guarded by the same predicate is first identified. Then detecting operation is added to identify if predications in scalar lane are all false and a jump operation followed to bypass the consecutive instructions region. For loop structure, we add loop mask to indicate which lane is not alive in loop which could help to treat iteration in loop. The experiment shows our method could improve performance by 6.8%.

Keywords: Branch, IF-conversion, loop, optimization, vectorization

INTRODUCTION

There are several vectorization methods for using SIMD units in modern architecture. Most vectorizing compilers pay attention to vectorize inner loops and get different degrees of success. Under Single Program Multiple Data (SPMD) model, programs appear to be regular serial program but in parallel execution model. Lots of thread instances could execute across SIMD lanes on multiple processors like GPU. Some programming languages, such as CUDA (NVIDIA, 2008a), Open CL (Lee *et al*., 2010), attempt to assist the vectorization of loops by stating in outer loop which have unspecified order and unsynchronized interaction time. When migrating SPMD programs to multi-core platform, IF-conversion (Tyson and Farrens, 1994) is used to solve the problem that there is no un-synchronized interaction between loop iterations.

SPMD programs use kernel function to express coarse parallel. When executing, it looks like synchronized by outer explicit loop. The outer loop is often distributed across multiple vector lanes. When implementing SPMD in no-GPU platform, some technologies are used to undertake vectorization. The whole function vectorization contains two steps: Masking and flattening of SPMD code (Karrenberg and Hack, 2011). First, mask is created for each basic block in the CFG. Next, each instruction in the program which has a side-effect is conditioned with a mask. After masks are assigned to different basic blocks, the blocks are ordered in linear order.

IF-conversion is a traditional compiler technology which converts code with control flow into a single control stream code. It places predicates which control the execution of instructions. IF-conversion in the context of scalar CPU code is studied in the context of their interaction with the processor branch predictor mechanism. Usually predicted branches are cheaper to execute than predicated code. A technique to generate Branches-on-Superword-Condition-Codes (BOSCCs) automatically is introduced by Jaewook Shin to overcome this overhead of executing all control paths (Shin *et al*., 2009). Because a sequence of consecutive vector instructions guarded by the same vector predicate can be bypassed by a BOSCC if all fields of the guarding vector predicate are false.

In this study, we extend BOSCCs to whole function vectorization which we call it Branch-on-Superword-Condition-Code-in-SSA (BOSCCS). We apply BOSCCS after the stages of masking and flattening of SPMD code in SSA form.

Through analyzing the relation of mask in consecutive blocks by conjunction transformation, we construct the mask region which could be bypass when all fields of the mask are false. The contributions of this study are mainly summarized as follow:

- We use conjunction transform to analyze the mask relation on which consecutive instructions of a mask could be identified.

Corresponding Author: Jianmin Pang, State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, 450001, China

- For loop structure, we use loop mask and entry mask to indicate the execution of loop.
- For consecutive region, we add mask checking and jump operation after entry mask created to improve the performance.

MATERIALS AND METHODS

IF-conversion: IF-conversion is one of the common compiler technologies to improve program performance (Tyson and Farrens, 1994). With the help of predication that modern micro-architectures supply, it transforms the branch statement to predication execution instruction. In the architecture supporting prediction execution, instructions are appended a prediction. When the prediction is true, instruction gets executed. Otherwise, it is treated as a null instruction. Prediction execution could remove the branch instructions effectively, which could convert control flow to data flow, bring more instruction level parallel.
For example, the following code:

```
if (a>b) c = c+1
else c = d*e+1
```

The branch a>b could be removed by adding prediction code:

```
pT, pF = compare (a>b)
(pT) c = c+1
(pF) c = d*e+1
```

When a>b is true, pT is set to 1 and pF is 0. Else pT is 0 and pF is 1. The control relation of c = c+1 and c = d*e+1 becomes data relation.

The main work of IF-conversion includes three steps. First, the basic blocks with same branch are selected to assign same prediction. Then, remove the branch instructions between blocks and replace them with prediction instructions. At last, the instructions in blocks are replaced by prediction instructions.

Different architecture has different support for prediction. Some architecture like DEC/Compaq alpha, SUN SPARC V9 only partly supports prediction in specific instructions. Some architecture fully supports prediction such as IA64.

BOSCC: Branch-on- Superword- Condition-Code (BOSCC) is proposed by Shin *et al.* (2009) which is a branch instruction that can be conditionally taken based on the comparison result of two vector variables. For example, the predicated vector instruction:

```
Vdst = vec_add; <Vpred>
```

Can be bypassed by introducing a BOSCC instruction as follows:

```
NotTaken = vec_any_ne (Vpred, ZeroVector)
if (NotTaken) { Vdst = vec add;}
```

The vec_any_ne instruction returns true if any field of Vpred does not equal to 0 in Zero Vector which contains false values in all fields, Not Taken will be set to false only when all fields of Vpred are false.

Predicate region refers to a sequence of consecutive instructions guarded by the same predicate and BOSCC region refer to the sequence of instructions enclosed by a BOSCC. The BOSCC regions could be bypassed whenever the guarding vector predicate has all false values, which is the only case when Not Taken is set to false. And BOSCC needs additional treatment in loop iteration (Shin, 2007).

Whole function vectorization in SSA: Traditional vectorization mainly concern loop structure which has better optimization potential. But when transform SPMD to multi-core platform, the whole function becomes the loop body, which has lots of unsuitable elements for traditional vectorization such as irregular variable use and complicated structure.

Under above conditions, Ralf Karrenberg proposes the whole function vectorization which implements data-parallel languages on machines with SIMD instruction sets. They describe an analysis based on a data-flow lattice approach. The thread instance in SPMD program becomes the scalar instance in SIMD vector.

The whole-function vectorization algorithm consist two main tasks, mask generation and CFG linearization.

Mask generation mainly treat with divergent control flow. Because a condition might be true for some scalar instances and false for others. In CFG, if a mask of a CFG edge is true, then the corresponding instance of the code took the edge branch. Thus, the mask denotes which elements in a vector contain valid data on the corresponding control-flow edge. Then select instructions are used to replace phi function in the original CFG. Because blend operations are inserted at control-flow join points.

CFG linearization put blocks into a sequence that preserves the execution order of the original CFG. Because after all mask and select operations are inserted, all control flow, except for loop backedges, is effectively encoded by data flow and can thus be removed.

Methods: Above all, we know that the whole function vectorization of SPMD programs has such characteristics:

- The different thread has little data dependence between others. Usually, threads communicate by explicit synchronization which exchange data in shared memory space. In addition, in some hardware the data dependence also is guaranteed by hardware execution, for example, the NVIDIA GPU. We don't concern implicit synchronization for they can be canceled when programming. In traditional vectorization, data dependence has to be concerned and detected first. From this point, it's very convenient to do vectorization.

- Threads have loose synchronization information in loops. The can execute loop iteration in different times. It's very different compared with traditional vectorization in which the iterations of loop is divided into several parts and every part execute same iteration times. This means whole function vectorization of SPMD programs needs more strict method to keep synchronization.

Based on the characteristic of whole function vectorization of SPMD program in SSA, we design the Branch- on-Superword- Condition-Code -in-SSA (BOSCCS) technology. This technology bases on the whole function vectorization method, improves the CFG linearization. There are three stages: mask relation analysis, construct bypass field and add jump operation.

Mask relation analysis: After CFG linearization in original vectorization, all basic block have in sequential order. We could identify the consecutive basic block or instructions which have same mask. It's possible to bypass there consecutive fields when all lanes in vector predication are false.

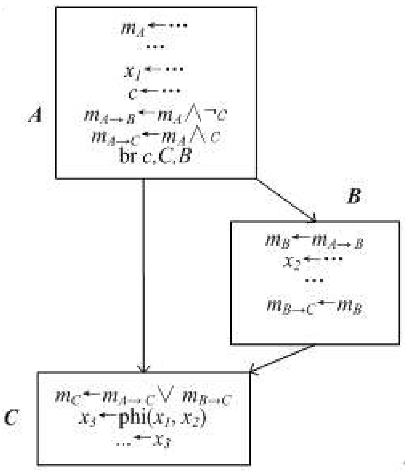
The challenge for consecutive fields identify is the mask relation analysis. Because in whole function vectorization, entry mask and edge mask are introduced for single block. As Fig. 1 shows, three block A, B, C have three entry mask m_A, m_B, m_C. Basic block B has only one entry edge $A\rightarrow B$, so $m_B \leftarrow m_{A\rightarrow B}$. And $m_{A\rightarrow B} \leftarrow m_A \wedge \neg c$, so $m_B \leftarrow m_A \wedge \neg c$. It means m_A is one element of m_B. Basic block C has two entry edge $A\rightarrow B$ and $B\rightarrow C$, so $m_C \leftarrow m_{A\rightarrow C} \vee m_{B\rightarrow C}$. And we know that $m_{A\rightarrow C} \leftarrow m_A \wedge c$ and $m_{B\rightarrow C} \leftarrow m_B$, $m_B \leftarrow m_{A\rightarrow B}$, so $m_C \leftarrow (m_A \wedge c) \vee (m_A \wedge \neg c)$. We transform m_C to $m_C \leftarrow m_A \wedge (c \vee \neg c)$. It also means m_A is one element of m_C. Based on the analysis, we can conclude that if m_A is false, then m_B and m_C must be false. A formalized description of this relation is:

$$\neg m_A \rightarrow \neg m_B \text{ and } \neg m_A \rightarrow \neg m_C$$

The above analysis is the no-loop situation. In the loop structure, mask relation becomes complicated for loop iteration.

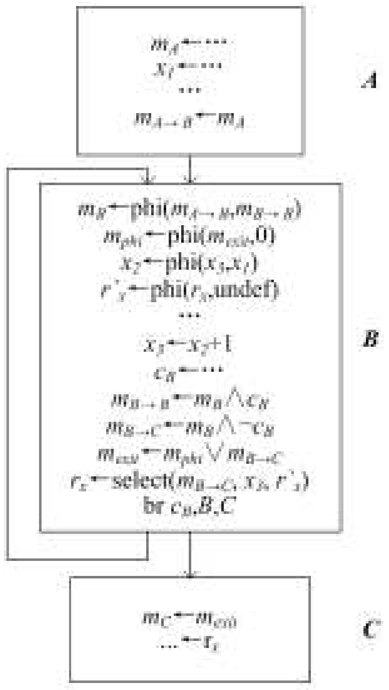
In loop structure, for whole function vectorization, loop mask m_{phi} is introduced to indicate which lane has exit from loop. But entry mask of block is still enough to dominate the vector execution in loop. In loop structure, m_{phi} and the parter m_{exit} both phi nodes. The m_{phi} just needs to be treated in the header block of loop. The m_{exit} just needs to be treated in the exit block of loop.

In Fig. 2 it is a simple loop structure which has irreducible CFG graph. Basic block A, B, C have the entry mask m_A, m_B, m_C. Block B is in loop structure, is has a loop mask m_{phi}. Because B is loop header, entry mask m_B indicate s the lane execution of vector. We add a m_{phi} to indicate which lane has exit the loop. To check if one iteration is executed in a lane, just to conform the corresponding bit in m_B.



A
$m_A \leftarrow \cdots$
$\cdots$
$x_1 \leftarrow \cdots$
$c \leftarrow \cdots$
$m_{A\rightarrow B} \leftarrow m_A \wedge \neg c$
$m_{A\rightarrow C} \leftarrow m_A \wedge c$
br c, C, B
B
$m_B \leftarrow m_{A\rightarrow B}$
$x_2 \leftarrow \cdots$
$\cdots$
$m_{B\rightarrow C} \leftarrow m_B$
C
$m_C \leftarrow m_{A\rightarrow C} \vee m_{B\rightarrow C}$
$x_3 \leftarrow$ phi(x_1, x_2)
$\ldots \leftarrow x_3$

Fig. 1: Entry mask for basic block



A
$m_A \leftarrow \cdots$
$x_1 \leftarrow \cdots$
$\cdots$
$m_{A\rightarrow B} \leftarrow m_A$
B
$m_B \leftarrow$ phi$(m_{A\rightarrow B}, m_{B\rightarrow B})$
$m_{phi} \leftarrow$ phi$(m_{exit}, 0)$
$x_2 \leftarrow$ phi(x_3, x_1)
$r'_x \leftarrow$ phi$(r_x$, undef$)$
$\cdots$
$x_3 \leftarrow x_2 + 1$
$c_B \leftarrow \cdots$
$m_{B\rightarrow B} \leftarrow m_B \wedge c_B$
$m_{B\rightarrow C} \leftarrow m_B \wedge \neg c_B$
$m_{exit} \leftarrow m_{phi} \vee m_{B\rightarrow C}$
$r_x \leftarrow$ select$(m_{B\rightarrow C}, x_3, r'_x)$
br c_B, B, C
C
$m_C \leftarrow m_{exit}$
$\ldots \leftarrow r_x$

Fig. 2: Entry mask for loop structure

Note that in block C, the entry mask is m_{exit} which is the disjunction of loop mask m_{phi} and edge mask $m_{B\rightarrow C}$. In each iteration, m_{exit} gets updated by the last m_{exit} and $m_{B\rightarrow C}$ in current iteration.

But in loop we also could conclude that:

$$\neg m_A \rightarrow \neg m_B \text{ and } \neg m_B \rightarrow \neg m_C$$

$\neg m_A \rightarrow \neg m_B$ is easy to confirm. We explain $\neg m_B \rightarrow \neg m_C$ in detail.

```
Algorithm ConstructBypassRegion(basic_block blocks)
  for each bb in blocks in fore order do
    if bb has entry mask $M$ then
      add instructions in bb to bb's bypass region
      bb1←bb.next in linear order
      while bb1 ≠ null
        if $\neg M \rightarrow \neg N$ ( $N$ is entry mask of bb1 ) then
          add instructions in bb1 to bb's bypass region
          bb1←bb1.next in linear order
        else
          break
        end if
      end while
    end if
  end for
```

Fig. 3: Construct bypass region

For we only concern the beginning of basic block, all lanes in m_B are false only take place after executing edge $A \rightarrow B$. If this situation happens after edge $B \rightarrow B$, that means all lanes in $m_{B \rightarrow B}$ are false in which condition edge $m_{B \rightarrow C}$ should be executed. So it's conflictive. And now $m_C \leftarrow m_{exit} \leftarrow m_{phi} \vee m_{B \rightarrow C} \leftarrow m_B \wedge \neg c_B$, so $\neg m_B \rightarrow \neg\, m_C$.

So we can see that in loop mask has same relation as no-loop structure. So a general method could be designed to identify the region of mask.

Construct bypass region: According to the relation, we design a linear scan algorithm to identify the consecutive instruction field for same mask. The algorithm is as in Fig. 3.

After the algorithm done, every entry mask of basic block contains a region which could be bypassed when every lane of mask is false (Fig. 3).

Note that we use conjunction transform to analyze the relation of different mask. When identify the relation of mask A and B, we just compare the equivalence of elements in A and B.

Add jump operation: Based on the consecutive region, we can add jump operation to the original code.

Although there are lots of branch instructions in most instruction set, in whole function vectorization of SSA we have remove the branch instructions in CFG linear stage. So the jump operation adding becomes easy.

In each basic block, after the place of mask creation, we add an operation to detect whether all lanes are false and a jump instruction. The target of jump is the end of mask's dominated region.

RESULTS AND DISCUSSION

We implement our BOSCCS algorithm based on Ralf Karrenberg's whole function vectorization method. After CFG linear stage, we apply BOSCCS algorithm on flattened blocks. Then we take the test on some cases from CUDA SDK (NVIDIA, 2008b).

Table 1: Benchmarks for testing

Case	Data set	Kernels	Block dimensions
MM	512*512	1	(16, 16, 1)
Transpose	512*512	1	(16, 16, 1)
Convolve	512*512	1	(16, 16, 1)
Histogram	2 M	2	(192, 1, 1), (256, 1, 1)
Mandelbrot	512*512	1	(16, 16, 1)

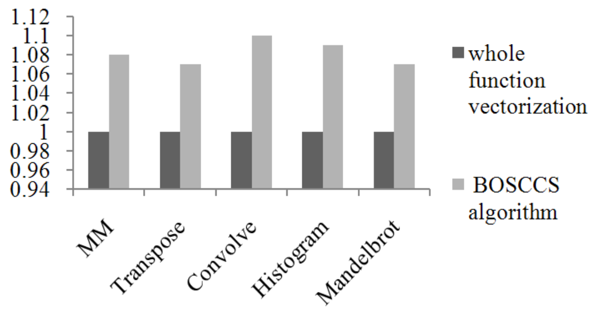


Fig. 4: The speedup results

Benchmarks: The test suit include a 2D image filter with 5×5 kernel (Convolve), Matrix Multiply (MM), 256-bin histogram (Histogram), fractal generation (Mandelbrot) and matrix transpose (Transpose). Table 1 lists data sizes and characteristics for all benchmarks.

Test methodology: With the above benchmarks, we get two versions of programs. One version is original whole function vectorization program. The second version is our BOSCCS algorithm.

The executing circumstance is Intel Pentium Dual CPU E2200 @2.2.0 Ghz, 1 GB DDR-1333 DRAM and Fedora 10 OS.

Figure 4 shows the speedup result. Compared with original whole function vectorization, the normalized speedup of our BOSCCS is obvious. The result shows that our optimization is effective for the performance could improve by 6.8%.

Nowadays, many compilers support SIMD instructions generation automatically. But the vector ability is limited by several factors, one of which is control flow. Usually, vectorization in the presence of control flow involves if-conversion followed by generating vector instrucions guarded by vector predicates.

Park and Schlansker developed an if-conversion technique that is optimal in the number of predicates used and in the number of predicate-defining instructions (Joseph *et al*., 1991). Shin *et al*. (2005) used this technique to generate SIMD instructions for modern microprocessors in the presence of arbitrarily complex cyclic control flow.

To overcome this overhead of executing all control paths, Shin *et al*. (2009) introduced a technique to generate Branches-on-Superword- Condition-Codes

(BOSCCs) automatically. A sequence of consecutive vector instructions guarded by the same vector predicate can be bypassed by a BOSCC if all fields of the guarding vector predicate are false. For the loops with complex control flow where if-statements are nested, a technique to nest BOSCC instructions so that multiple BOSCCs and the vector instructions enclosed within them can be bypassed by a single BOSCC instruction (Shin, 2007).

Rotem and Ben-Asher (2012) present an IF-conversion method for unifying basic blocks by merging similar instructions using operand selection, for reducing the number of predicated instructions in the code. They use a polynomial time algorithm for finding the optimal pairing between similar instructions that reduces the overall predication cost.

CONCLUSION

Whole function vectorization is a new vectorization technology which takes the thread instance in SPMD program as scalar instance in SIMD vector. In order to improve the IF-conversion in vectorization, we apply the similar technology to BOSCCs in SSA. By analyzing the entry mask relation, we identify the mask region conveniently. Then we add jump operation to the end of mask region. So if all field of entry mask are false, the mask region could be bypassed. The experiment also shows the effect of our algorithm.

ACKNOWLEDGMENT

R. B. G. thanks the sporting of State Key Laboratory of Mathematical Engineering and Advanced Computing and China National Digital Switching System Engineering and Technological Research Center. This project is supported by National High Technology Research and Development Program of China under Grant No. 2009AA012201 and the China Henan Municipal Science and Technology Project under Grant No. 092101210501.

REFERENCES

Joseph, C., H. Park and M. Schlansker, 1991. On predicated execution. Technical Report HPL-91-58, Software and Systems Laboratory.

Karrenberg, R. and S. Hack, 2011. Whole-function vectorization. Proceeding of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization. The IEEE Computer Society, pp: 141-150.

Lee, J., J. Kim, S. Seo, S. Kim, J. Park, H. Kim, T.T. Dao, Y. Cho, S.J. Seo, S.H. Lee, S.M. Cho, H.J. Song, S.B. Suh and J.D. Choi, 2010. An opencl framework for heterogeneous multicores with local memory. Proceeding of the 19th International Conference on Parallel Architectures and Compilation Techniques. ACM, New York, USA, pp: 193-204.

NVIDIA, 2008a. NVIDIA CUDA Compute Unified Device Architecture. 2nd Edn., NVIDIA Corporation, Santa Clara, California.

NVIDIA, 2008b. NVIDIA CUDA SDK 2.1. 2nd Edn., NVIDIA Corporation, Santa Clara, California.

Rotem, N. and Y. Ben-Asher, 2012. Block unification If-conversion for high performance architectures. IEEE Comput. Archit. Lett., 1(9): 1.

Shin, J., 2007. Introducing control flow into vectorized code. Proceeding of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07). IEEE Computer Society, pp: 280-291.

Shin, J., M.W. Hall and J. Chame, 2005. Superword-level parallelism in the presence of control flow. Proceeding of the International Symposium on Code Generation and Optimization, pp: 165-175.

Shin, J., M.W. Hall and J. Chame, 2009. Evaluating compiler technology for control-flow optimizations for multimedia extension architectures. Microprocess. Microsy., 33(4): 235-243.

Tyson, G. and M. Farrens, 1994. Evaluating the effects of predicated execution on branch prediction. Proceeding of the 27th International Symposium on Microarchitecture, pp: 196-206.