CSC766: Project - Fall 2010
Vivek Deshpande and Kishor Kharbas
Auto-vectorizating compiler for pseudo language

Table of contents

1. Introduction

Modern CPUs are equipped with SIMD instruction extensions like Intel's MMX, SSE and ARM's NEON. This means that the CPU can perform a computation on an array of data values in a single instruction cycle. Programs can leverage this capability by using vector instructions in their programs to gain better performance.

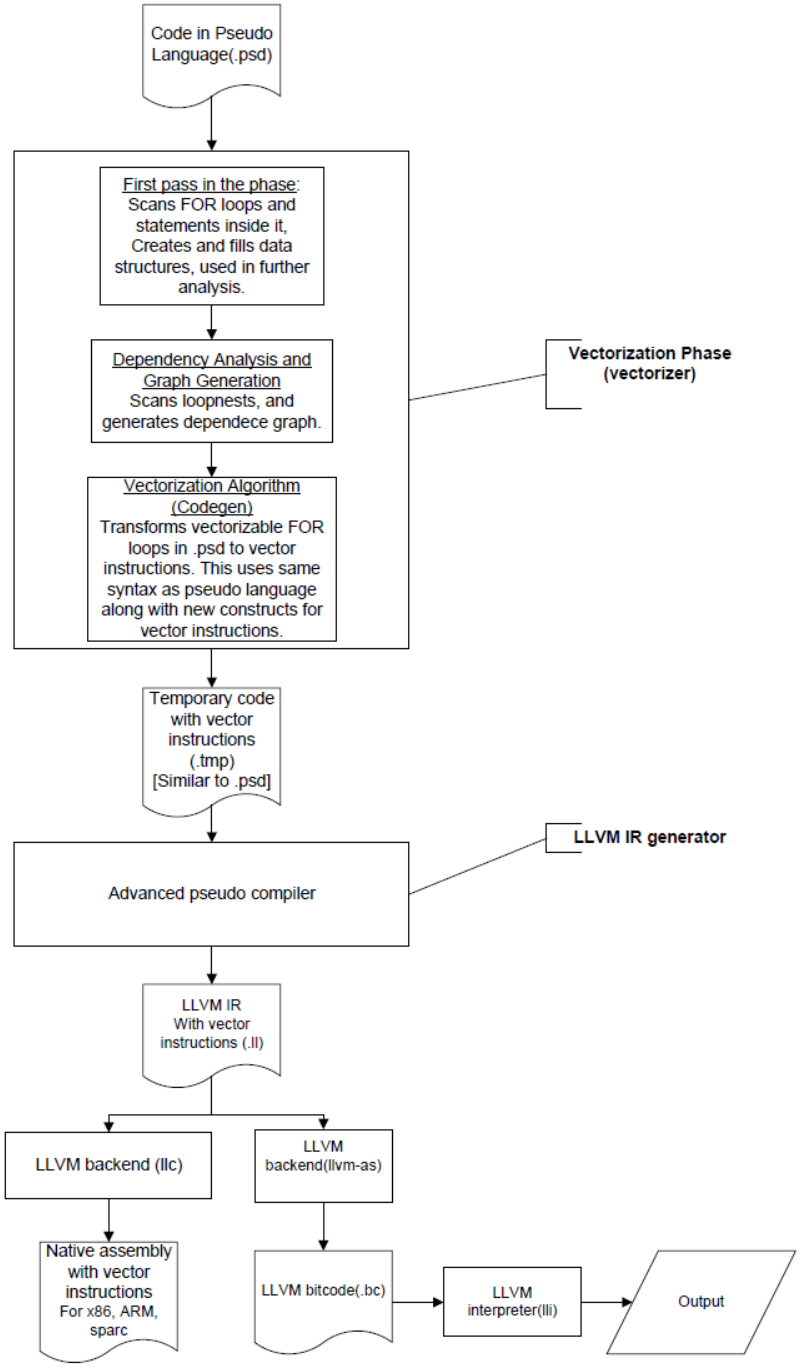
To make use of these instructions, we developed a compiler for the pseudo language introduced through class assignments with auto vectorization capability.

Following are the highlights of the compiler

- Automatic detection of vectorizable instructions
- Detects and vectorizes ZIV and SIV dependencies
- Safe transformations in case of MIV and other non-linear array subscripts
- Transforms the inner-most loop nest
- Loop distributions when partial loop can be vectorized
- Use LLVM backend to generate target code, which supports multiple target architectures

2. Implementation details

2.1. High Level Design



2.2. Data Structures Used in Implementation

- The properties of loop are stored in the data structure 'loop' such as upper bound lower bound, index pointer to first instruction in the loop and so on. The data structure is shown below,

```
typedef struct loop
{
      char index[10];
      int isnormal;
      int depth;                  //depth is equivalent to level
      int rdepth;
      int lB,uB,step_size;
      struct stmnt_or_loop *stmnt_start;
      struct loop *parent_loop;
}loop_t;
```

- The properties of statement are stored in the data structure stmnt such as variable is scalar or array reference and so on.

```
typedef struct stmnt
{
      int type;              //0- arithmetic ; 2 - other
      struct v_node *lhs;     //v_node ptr to a scalar or array variable
      struct v_node *rhs;     //variables used in this statement
      char *src_stat;
      struct loop *parent_loop;
}stmnt_t;
```

- A very useful v_node data structure is used to represent the elements of the statement in the form of tree which is shown below. This data strcture forms the elements in the tree. Such tree is formed for each element.

```
typedef struct v_node
{
      int type;              //0-leaf 1-non-leaf
      int subtype;           //for leaf which type of operand ; for non-leaf which operator
      char sym[10];          //if leaf, this is the variable
      struct v_node *left;
      struct v_node *right;
      struct v_node *scrpt_exprs[2];    //for arrays this is expr for max. 2 subscripts
      double value;
}v_node_t;
```

2.3. Vector extensions and advanced pseudo compiler

For intermediate representation of the instructions that are transformed, vector constructs were added to the pseudo language.

- An array reference can be a range in the form lb:ub where ‘lb’ and ‘ub’ are arithmetic expressions which represent the lower and upper bounds respectively. So array references can be of the form a[10:20] or a[10*i:20*i]

- Arithmetic operators for vector operands take the ‘range’ in the form ‘op:range’ where range is an integer specifying the length of the operands. For example +:16 specifies ‘+’ operation on two operands of size 16 doubles

- Assignment involving vector data must also be appended with size of data as in, a[11:20] **:=:10** b[11:20] +:10 c[11:20]

    - range ← aexpr : aexpr
    - vexpr ← ID[range]
      ← vexpr ARITH_OP :INT vexpr
    - assignment ← ID[range] :=:INT vexpr

The **advanced compiler** transforms code with normal statements + vector statements to LLVM IR code. In our processing sequence, it takes the intermediate .tmp file as input which is generated by vectorization phase and translates it into LLVM IR vector operations. LLVM IR supports vector operations in the form of vector data types and all the operations work on vector data types. More details of LLVM vector instruction are given in [1].

Abstract Syntax Tree
The advanced pseudo compiler uses abstract syntax tree for the generation of LLVM IR. Since the range of a vector operation is associated with the operator and not with the operands, while parsing the operands this range is not known. So while parsing, all expressions are parsed into an AST which is transformed to LLVM IR operations later when the entire expression is parsed. This also facilitates generation of unnamed temporary variables in program order which is a requirement in LLVM IR. Abstract syntax tree also gives reusability and flexibility.

2.4. Dependency Analysis

It is a task of determining whether the statements within a loop body form data dependence with respect to array references. It consists of subscript analysis then followed by different tests to determine the dependencies within the statements.

Subscript Analysis
/*The array references in pseudo language are single dimensional also pseudo language does not perform aliasing.*/
In our program, the subscripts for array references are analyzed and classified as Zero Index Variable (ZIV), Single Index Variable (SIV). In SIV those are further classified as strong SIV $<ai+c_1, ai+c_2>$, weak zero SIV $<ai+c_1, c_2>$, weak crossing SIV $<ai+c_1,-ai+c_2>$ and further as of general SIV $<a_1i+c_1, a_2i+c_2>$.
Currently the indices are not classified for the Multiple Index Variable (MIV) category.
Also, since pseudo language has single dimensional arrays, there is no need of analysis of coupling.

Single subscript Tests
For each subscript single subscript tests are applied such as ZIV, SIV and variants of SIV tests. The dependencies within statements are identified such as loop carried or loop independent dependencies. Also the distance and direction within subscripts of the array reference is obtained.

The dependency graph is generated with statements in the loop as vertices and dependencies as edges. The properties of edge are - Loop carried (with level) or loop independent along with direction and distance vectors.

2.5. Vector code generation

This phase operates on the loop data structures and the dependence graph generated in previous pass. It analyses the dependences within the statements in the loops by applying the 'codegen' algorithm discussed in class.

These are the steps performed:
- Find strongly connected components and sort them in topological order
- Analyze each SCC, if its cyclic emit out the k-level FOR loop.
  and remove k-level dependencies and analyze further.
- If statement is not cyclic generate vector statement.

3. Test cases

Testing is divided into two parts:

1. We first test whether vectorizable instructions are identified and are correctly transformed into vector instructions.
2. Then we test whether the .tmp file generated(which contains vector additions) is correctly transformed to LLVM IR(.ll file)

1. For the 1st part, following are some of the test cases been tested and their outcomes. Other test cases with more complicated array references and statements are provided in a separate file

Functionality tested	Input source code(.psd)	Expected output	Generated .tmp file

ZIV subscripts	FOR i:=1 TO 10 DO
	a[j] := b[i] + c[1];
	c[j] := a[1] + a[j-1];
	ENDFOR;
		FOR i:=1 TO 10 DO
		a[j] := b[i] + c[i];
		c[i] := a[1] + a[j - 1];
		ENDFOR;
			FOR i:=1 TO 10 DO
			a[j] := b[i] + c[i];
			c[i] := a[1] + a[j - 1];
			ENDFOR;

Strong SIV	FOR i:=1 TO 32 DO
	a[i] := a[i-1] + c[i];
	ENDFOR;
		FOR i:=1 TO 32 DO
		a[i] := a[i - 1] + c[i];
		ENDFOR;
			FOR i:=1 TO 32 DO
			a[i] := a[i - 1] + c[i];
			ENDFOR;

Strong SIV ('d' > loop-bounds)	FOR i:=1 TO 32 DO
	a[i] := a[i+32] + c[i];
	ENDFOR;
		a[1:32] :=:32 a[1+32:32+32] +:32 c[1:32];	a[1:32] :=:32 a[1+32:32+32] +:32 c[1:32];

Strong SIV(non-integer 'd')	FOR i:=1 TO 32 DO
	a[3*i+1] := a[3*i+32] + c[i];
	ENDFOR;
		a[3*1+1:3*32+1] :=:32 a[3*1+32:3*32+32] +:32 c[1:32];	a[3*1+1:3*32+1] :=:32 a[3*1+32:3*32+32] +:32 c[1:32];

Strong SIV	FOR i:=1 TO 32 DO
	a[3 * i + 1] := a[3 * i + 61] + c[i];
	ENDFOR;
		FOR i:=1 TO 32 DO
		a[3 * i + 1] := a[3 * i + 61] + c[i];
		ENDFOR;
			FOR i:=1 TO 32 DO
			a[3 * i + 1] := a[3 * i + 61] + c[i];
			ENDFOR;

Weak zero SIV	FOR i:=1 TO 32 DO
	a[i] := a[i] + c[i];
	d[i] := a[10] + c[i];
	ENDFOR;
		FOR i:=1 TO 32 DO
		a[i] := a[i] + c[i];
		d[i] := a[10] + c[i];
		ENDFOR;
			FOR i:=1 TO 32 DO
			a[i] := a[i] + c[i];
			d[i] := a[10] + c[i];
			ENDFOR;

Weak zero SIV ('d' outside array bounds)	FOR i:=1 TO 32 DO
	a[i] := a[i] + c[i];
	d[i] := a[80] + c[i];
	ENDFOR;
		a[1:32] :=:32 a[1:32] +:32 c[1:32];
		FOR i:=1 TO 32 DO
		d[i] := a[80] + c[i];
		ENDFOR;
			FOR i:=1 TO 32 DO
			d[i] := a[80] + c[i];
			ENDFOR;
			a[1:32] :=:32 a[1:32] +:32 c[1:32];

Weak zero SIV ('d' not an integer)	FOR i:=1 TO 32 DO
	a[2*i] := a[2*i] + c[i];
	d[i] := a[9] + c[i];
	ENDFOR;
		a[2*1:2*32] :=:32 a[2*1:2*32] +:32 c[1:32];
		FOR i:=1 TO 32 DO
		d[i] := a[9] + c[i];
			FOR i:=1 TO 32 DO
			d[i] := a[9] + c[i];
			ENDFOR;
			a[2*1:2*32] :=:32 a[2*1:2*32]

		ENDFOR;	+:32 c[1:32];

Weak crossing SIV('d' is integer)	FOR i:=1 TO 32 DO
	a[-2*i+1] := a[2*i+13] + c[i];
	ENDFOR;
		FOR i:=1 TO 32 DO
		a[-2 * i + 1] := a[2 * i + 13] + c[i];
		ENDFOR;
			FOR i:=1 TO 32 DO
			a[-2 * i + 1] := a[2 * i + 13] + c[i];
			ENDFOR;

Weak crossing SIV('d' is integer+1/2)	FOR i:=1 TO 32 DO
	a[-2*i+1] := a[2*i+15] + c[i];
	ENDFOR;
		FOR i:=1 TO 32 DO
		a[-2 * i + 1] := a[2 * i + 15] + c[i];
		ENDFOR;
			FOR i:=1 TO 32 DO
			a[-2 * i + 1] := a[2 * i + 15] + c[i];
			ENDFOR;

Weak crossing SIV('d' outside loop bounds)	FOR i:=1 TO 32 DO
	a[-2*i+1] := a[2*i+257] + c[i];
	ENDFOR;
		a[-2*1+1:-2*32+1] :=:32
		a[2*1+257:2*32+257] +:32
		c[1:32];
			a[-2*1+1:-2*32+1] :=:32
			a[2*1+257:2*32+257] +:32
			c[1:32];

Weak crossing SIV('d' not a integer or int1/2)	FOR i:=1 TO 32 DO
	a[-2*i+1] := a[2*i+14] + c[i];
	ENDFOR;
		a[-2*1+1:-2*32+1] :=:32
		a[2*1+14:2*32+14] +:32 c[1:32];
			a[-2*1+1:-2*32+1] :=:32
			a[2*1+14:2*32+14] +:32 c[1:32];

General SIV	FOR i:=1 TO 32 DO
	a[2*i+1] := a[3*i+15] + c[i];
	ENDFOR;
		a[2*1+1:2*32+1] :=:32
		a[3*1+15:3*32+15] +:32 c[1:32];
			a[2*1+1:2*32+1] :=:32
			a[3*1+15:3*32+15] +:32 c[1:32];

General SIV	FOR i:=1 TO 32 DO
	a[2*i+15] := a[3*i+1] + c[i];
	ENDFOR;
		FOR i:=1 TO 32 DO
		a[2 * i + 15] := a[3 * i + 1] + c[i];
		ENDFOR;
			FOR i:=1 TO 32 DO
			a[2 * i + 15] := a[3 * i + 1] + c[i];
			ENDFOR;

2. For the 2nd part, we test whether the .tmp files are correctly translated by our advanced pseudo compiler to LLVM IR using vector instruction. These test cases are provided in the file test_adv_pseudo.tmp

4. Performance Analysis

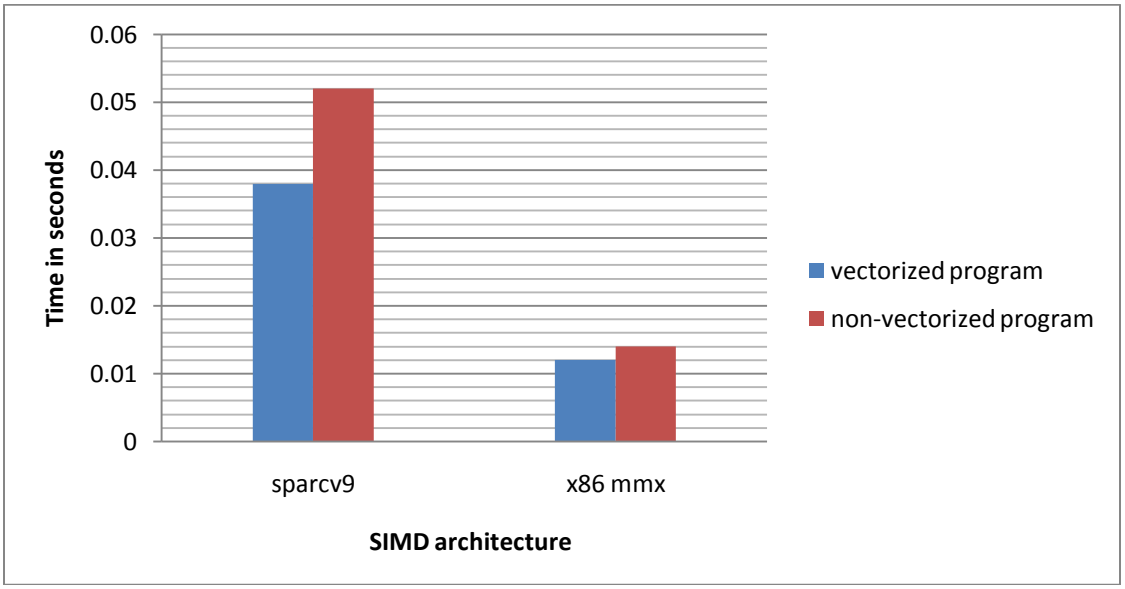
We tested our auto vectorizing compiler on sparcv9(with VIS) and x86 (with SSE) platforms. According to theoretical analysis, for intel SSE, two instructions on double values execute in one clock cycle, which should ideally give double or 100% speed up. But the presence of serial components and also optimizations performed by llvm compiler (with and without vectorization) makes performance evaluation difficult. Still for our simple test case we noticed a performance gain.
In our simple benchmark code, there is a simple FOR loop with 100000 iterations containing single statement of addition which could be vectorized, and there are 2 FOR loops of 100 iterations to initialize and write respectively.

```
FOR i:=0 TO 999 DO
        b[i]:=i;
        c[i]:=i;
ENDFOR;

FOR i:=0 TO 99999 DO
        a[i]:=b[i]+c[i];
ENDFOR;

FOR i:=0 TO 999 DO
        WRITE(a[i]);
ENDFOR;
```

The time in seconds for execution of vectorized and non-vectorized benchmark program for the two SIMD architectures are as shown below,



Thus considering the serial or non vectorizable component in the program and the also other optimizations applied by LLVM compiler, we found a speed up of 25.5%

5. Task Division

Vivek Deshpande	Kishor Kharbas
Phase 1 • Studying and understanding LLVM IR, tools • Abstract syntax tree generation and evaluation. • Action associated with “for loop” of parser • Study of dependence analysis and vectorization theory	**Phase 1** • Studying and understanding LLVM IR, tools • Implementing actions associated with rules in parser(except for loop) • Study of dependence analysis and vectorization theory
Phase 2 • Implementation of Dependency analysis and graph generation. • Abstract Syntax Tree and grammar for vector constructs in Pseudo compiler. • Implementation of Actions and for Array access in pseudo compiler • Performance Analysis and Testing	**Phase 2** • Implementation of basic compiler for loop scanning and data structure creation. • Implementation of Vectorization Algorithm, along with graph analysis and graph operations such as finding SCC and topological sorting. • Implementation of productions, grammar for vector constructs in pseudo compiler • Testing and Performance Analysis

6. Open Issues and Future Work

- Handle dependency tests for references having MIV subscripts.
- Vectorization with loop transformations.
- Currently innermost loop is considered for vectorization; this could be extended for arbitrary loop nest.
- The pseudo language could be extended for more than one dimension arrays and such array references could be analyzed.
- Few tests resulted in segmentation faults, because of issues with the memory alignment required for vector instructions.

7. Conclusion:

Successfully implemented an auto-vectorizing compiler for pseudo language with the help of LLVM backend.

References

1]
http://www.celinuxforum.org/CelfPubWiki/ELC2009Presentations?action=AttachFile&do=get&target=LLVM-ELC2009.pdf
2]
http://www.zap.org.au/elec2041-cdrom/reference/arm-thumb-instructions-quickref.pdf
3] LLVM design
http://llvm.org/docs/#llvmdesign
4] LLVM passes
http://llvm.org/docs/Passes.html
5] LLVM IR specification
http://llvm.org/docs/LangRef.html
6] The Stony Brook Algorithm Repository
http://www.cs.sunysb.edu/~algorith/files/dfs-bfs.shtml