Contents of Lecture 11

- Unimodular Transformations
- Inner Loop Parallelization
- SIMD Vectorization

Unimodular Transformations

- A **unimodular transformation** is a loop transformation completely expressed as a unimodular matrix $\mathbf{U}$.
- A loop nest $\mathbf{L}$ is changed to a new new loop nest $\mathbf{L_U}$ with loop index variables:

$$\mathbf{K} = \mathbf{IU}$$
$$\mathbf{I} = \mathbf{KU}^{-1}$$

- The same iterations are executed but in a different order.
- A new iteration order might make parallel execution possible.
- Before generating code for the new loop, the loop bounds for $\mathbf{K}$ must be computed from the original bounds:

$$\left.\begin{array}{rcl} \mathbf{p_0} & \leq & \mathbf{IP} \\ & & \mathbf{IQ} \leq \mathbf{q_0} \end{array}\right\}$$

Computing the New Index Variables

- With

$$\left.\begin{array}{rcl} \mathbf{p_0} & \leq & \mathbf{IP} \\ & & \mathbf{IQ} \leq \mathbf{q_0} \end{array}\right\} \quad (1)$$

$$\mathbf{I} = \mathbf{KU}^{-1} \quad (2)$$

We use Fourier-Motzkin elimination to find the loop bounds from

$$\left.\begin{array}{rcl} \mathbf{p_0} & \leq & \mathbf{KU}^{-1}\mathbf{P} \\ & & \mathbf{KU}^{-1}\mathbf{Q} \leq \mathbf{q_0} \end{array}\right\} \quad (3)$$

- The bounds are found starting with k_1, k_2 etc.
- This is the reason why we want to have an invertible transformation matrix.

New Array References

- All array references are rewritten to use the new index variables.
- Conceptually we could calculate, at the beginning of each loop iteration,

$$\mathbf{I} = \mathbf{K}\mathbf{U}^{-1}$$

  and then use this vector $\mathbf{I}$ in the original references, on the form:

$$x[\mathbf{I}\mathbf{A} + \mathbf{a_0}]$$

- We don't do that of course and instead replace each reference with

$$x[\mathbf{K}\mathbf{U}^{-1}\mathbf{A} + \mathbf{a_0}]$$

- Here $\mathbf{K}\mathbf{U}^{-1}\mathbf{A} + \mathbf{a_0}$ is be calculated at compile-time.
- For instance the **lmpcc** compiler has a function `make_ref` which takes an array reference and a transformation matrix, and produces new three-address code with the new index variables.

The Distance Matrix

- The set of all vectors of dependence distances is represented by the **distance matrix D**.
- We are free to swap the rows of **D** since it really is a set of dependencies.
- Unimodular transformations require that all dependencies are uniform, i.e. with known constants.
- Consider a uniform dependence vector $\mathbf{d} = \mathbf{j} - \mathbf{i}$.
- With the **K** index variables we have $\mathbf{d_U} = \mathbf{jU} - \mathbf{iU} = \mathbf{dU}$.
- Therefore, given a dependence matrix **D** and a unimodular transformation **U**, the dependencies in the new loop $\mathbf{L_U}$ become:

$$\mathbf{D_U} = \mathbf{DU}$$

Valid Distance Matrices

- The sign, **lexicographically**, of a vector is the sign of the first nonzero element.
- A distance vector can never be lexicographically negative since it would mean that some iteration would depend on a future iteration.
- Therefore row in the new distance matrix $\mathbf{D_U} = \mathbf{DU}$ may be lexicographically negative.
- If we would discover a lexicographically negative row in $\mathbf{D_U}$, that loop transformation is invalid, such as the second row of the following $\mathbf{D_U}$:

$$D_U = \begin{pmatrix} 1 & 2 \\ -1 & 1 \end{pmatrix}$$

Inner Loop Parallelization

- By **inner loops** is meant all loops except the outermost loop.
- We can always find a unimodular matrix through which we can parallelize the inner loops, but the program might run slower...
- To parallelize the inner loops, we need to assure that all loop carried dependencies are carried at the outermost loop.
- In other words, the leftmost column of the distance matrix $\mathbf{D_U}$ simply should consist only of positive numbers!

Inner Loop Parallelization Example

- Assume we have the distance matrix $\mathbf{D}$ defined as:

$$\mathbf{D} = \begin{pmatrix} 0 & 0 & 2 \\ 0 & 3 & -4 \\ 0 & 2 & -1 \\ 4 & 0 & 4 \end{pmatrix}$$

- With this distance matrix, no loop can be executed in parallel.
- We want a $\mathbf{D_U}$ with positive first column:

$$\mathbf{D_U} = \begin{pmatrix} \geq 1 & ? & ? \\ \geq 1 & ? & ? \\ \geq 1 & ? & ? \\ \geq 1 & ? & ? \end{pmatrix} = \begin{pmatrix} 0 & 0 & 2 \\ 0 & 3 & -4 \\ 0 & 2 & -1 \\ 4 & 0 & 4 \end{pmatrix} \begin{pmatrix} u_1 & ? & ? \\ u_2 & ? & ? \\ u_3 & ? & ? \end{pmatrix}$$

Continued example

- Searching for a transformation, we get the following system of inequalities:

$$\begin{array}{rcrcl} & & 2u_3 & \geq & 1 \\ 3u_2 & - & 4u_3 & \geq & 1 \\ 2u_2 & - & u_3 & \geq & 1 \\ 4u_1 & + & 4u_3 & \geq & 1 \end{array}$$

- Since there are no upper bounds on u_i, there are infinitely many solutions to this equation. We will choose the smallest integer u_i which satisfies the inequalities.

Continued example

- So, u_3 is chosen as $\lceil 1/2 \rceil = 1$.
- Then we proceed with u_2, for which there are two inequalities: $u_2 \geq \lceil (1 + 4u_3)/3 \rceil = 2$ and $u_2 \geq \lceil (1 + u_3)/2 \rceil = 1$, so u_2 is chosen as the maximum of these, or $u_2 \leftarrow 2$.
- Finally, $u_1 \geq \lceil (1 - 4u_3)/4 \rceil = 0$, so $u_1 \leftarrow 0$. We get

$$\mathbf{U} = \begin{pmatrix} 0 & 1 & 0 \\ 2 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

and

$$\mathbf{D} \times \mathbf{U} = \begin{pmatrix} 2 & 0 & 0 \\ 2 & 0 & 3 \\ 3 & 2 & 2 \\ 4 & 0 & 0 \end{pmatrix}.$$

- The new loop nest $\mathbf{L_U}$ thus carries all dependencies in the outermost loop L_1, with the consequence that L_2 and L_3 can be vectorized.

Continued example

function *hyperplane_method* (**D**)

/* Group rows with leading element in position r together. */

for ($r \leftarrow 1; r \leq m; r \leftarrow r + 1$) {

$D_r \leftarrow \{\mathbf{d} \in \mathbf{D} : \mathbf{d} \prec_r \mathbf{0}\}$

}

for ($r \leftarrow m; r \geq 1; r \leftarrow r - 1$) {

if ($D_r = \emptyset$)

$u_r \leftarrow 0$

else

$u_r \leftarrow \lceil \max_{\mathbf{d} \in \mathbf{D}} \{(1 - d_{r+1}u_{r+1} - d_{r+2}u_{r+2} - \ldots - d_m u_m)/d_r\} \rceil^+$

}

/* now $\gcd(u_1, u_2, \ldots, u_m) = 1$. */

/* and $d_1 u_1 + d_2 u_2 + \ldots + d_m u_m \geq 1 \quad (\mathbf{d} \in \mathbf{D})$. */

$\mathbf{u} \leftarrow (u_1, u_2, \ldots, u_m)$

$k \leftarrow$ the first nonzero element in the sequence $u_m, u_{m-1}, \ldots, u_1$

let U be an $m \times m$ unimodular matrix such that

(1) the first column is **u**

(2) the k^{th} row is $(1, 0, \ldots, 0)$

(3) The matrix obtained by deleting column 1 and row k of **U** is I_{m-1}

return U

end

SIMD Vectorization of DSP Codes

- The following will describe SIMD Vectorization in the **lmpcc** compiler, which essentially is similar to that in other compilers.
- The vectorizations are done on SSA Form.
- The most important steps are data dependence analysis and identifying expressions suitable for AltiVec SIMD instructions.
- Other optimizations such as SSAPRE make it easier to perform vectorization since certain address calculations are automatically moved out to the proper location.

An Example: a FIR filter 1(3)

$$y(n) = \sum_{i=n-LENGTH}^{n} h(i) \cdot x(i)$$

- *LENGTH* is the number of past input samples remembered
- $y(n)$ is the output at time n — represented as a scalar
- x are the input samples — represented as a vector of length L
- the coefficients h is also a vector of length L
- after one step, x is shifted right one position and a new sample is stored in $x(0)$
- the main operation is the multiply-add called multiply-accumulate, or MAC

An example: a FIR filter 2(3)

$$y(n) = \sum_{i=n-LENGTH}^{n} h(i) \cdot x(i)$$

- the multiplications can be performed concurrently
- we will see later that the accumulation to $y(n)$ can be done concurrently on AltiVec
- moving x right one position is trivial
- so why does Hennessy and Patterson have the following pitfall in the third edition of their computer architecture book ?
  - "Pitfall: Expecting to get good performance from a compiler for DSPs"
  - Writing a FIR filter in assembler gives 11.5 times better performance than using TI's compiler for the TMS320C54D DSP processor

An example: a FIR filter 3(3)

- DSP codes written in C use pointers a lot (intentionally in DSP stone)
- The FIR filter from DSP stone used by Hennessy and Patterson:

```
float y, h[LENGTH], x[LENGTH], *px, *px2, *ph;

px = &x[LENGTH-1]; px2 = &x[LENGTH-2]; ph = &h[LENGTH-1];
y = 0;
for (i = 0; i < LENGTH - 1; i++) {
    y += *ph-- * *px ;
    *px-- = *px2-- ;
}
y += *ph * *px ;
*px = x0 ;
```

Another example from DSP stone: N complex updates

```
for (i = 0 ; i < N ; i++, p_a++)
  {
    *p_d    = *p_c++ + *p_a++ * *p_b++ ;
    *p_d++ -=          *p_a   * *p_b-- ;
    *p_d    = *p_c++ + *p_a-- * *p_b++ ;
    *p_d++ +=          *p_a++ * *p_b++ ;
  }
```

- The TI compiler produces 9.5 times slower code than hand-optimized assembler
- DSP programmers often assume that code should be hand-optimized for very simple processors and naive compilers
- This coding style often confuses compilers

PowerPC/AltiVec SIMD Vector Processor

- Available from Freescale and IBM
- Based on superscalar PowerPC designs
- 32 16-byte vector registers
- 162 new instructions
- programmable data prefetch engines
- float; signed/unsigned int/short/char both normal and saturated; bool
- no double precision floating point

Examples of AltiVec instructions: Vector Permute

- `vperm vd, va, vb, vc`
- Permutes the contents of source vectors VA and VB according to VC
- Bytes in VA are called 00, 01, 02, ..., 0f
- Bytes in VB are called 10, 11, 12 ..., 1f
- Eg: if byte k in VC contains 12 then byte 2 of VB is stored in byte k in VD
- Vector permute is very useful and is executed by a separate functional unit

AltiVec Instructions: load/store 1(2)

- Loads or stores a vector register
- `lvx vd, ra, rb` fetches 16 bytes at address ra+rb into vd
- Memory accesses discard the four last bits of the virtual address
  - the compiler either must know that a reference is aligned, or
  - two vectors must be loaded and shifted appropriately
  - next slides shows how this is done
- AltiVec suffers a penalty if the compiler cannot control the alignment of arrays

AltiVec instructions: load/store 2(2)

- Assume we have the reference `a[i]` and we don't know about its alignment
- Assume `&a` is in `r1` and an offset is in `r2`.
- We want to load `a[i]`, `a[i+1]`, `a[i+2]`, and `a[i+3]` into `v3`.

```
lvx     v0, r1, r2      // load lower part
addi    r3, r1, 16      // address of upper part
lvx     v1, r3, r2      // load upper part
lvsl    v2, r1, r2      // produce vector for vperm
vperm   v3, v0, v1, v2  // extract a[i..i+3] from of v0 # v1
                        // where # means concatenation
```

Sometimes the lvsl and one load can be moved out of the loop

AltiVec Instructions: vadd

`vaddfp`	`vd, va, vb`	vector add single-precision floating point
`vaddsbs`	`vd, va, vb`	vector add signed byte saturate
`vaddshs`	`vd, va, vb`	vector add signed half saturate
`vaddsws`	`vd, va, vb`	vector add signed word saturate
`vaddubm`	`vd, va, vb`	vector add unsigned byte modulo
`vaddubs`	`vd, va, vb`	vector add unsigned byte saturate
`vadduhm`	`vd, va, vb`	vector add unsigned half modulo
`vadduhs`	`vd, va, vb`	vector add unsigned half saturate
`vadduwm`	`vd, va, vb`	vector add unsigned word modulo
`vadduws`	`vd, va, vb`	vector add unsigned word saturate

There are numerous arithmetic instructions for different data types.

Vectorization on SSA Form in the LMPCC compiler

1. Initial optimizations on SSA form, eg copy and constant propagation
2. Construct the loop tree from the control flow graph
3. Rewrite pointer expressions to array references using the SSA graph
4. Identify statements in each basic block
5. Construct the data dependence graph and the dependence matrix
6. Perform unimodular transformations if useful.

Rewriting pointer references to array references 1(2)

```
for (i = 0; i < LENGTH - 1; i++) {
      y += *ph-- * *px ;
      *px-- = *px2-- ;
}

for (i = 0 ; i < LENGTH - 1; i++) {
      y += h[ -1 * i + 15 ] * x[ -1 * i + 15 ];
      x[ -1 * i + 15 ] = x[ -1 * i + 14 ];
```

Rewriting pointer references to array references 2(2)

```
for (i = 0 ; i < N ; i++, p_a++) {
      *p_d    = *p_c++ + *p_a++ * *p_b++ ;
      *p_d++ -=          *p_a   * *p_b-- ;
      *p_d    = *p_c++ + *p_a-- * *p_b++ ;
      *p_d++ +=          *p_a++ * *p_b++ ;
}
```

```
for (i = 0 ; i < N ; i++) {
      d[ 2 * i ]     = c[ 2 * i ]  + a [ 2 * i ] * b [ 2 * i ];
      d[ 2 * i ]     = d[ 2 * i ]  - a [ 2 * i + 1] * b [ 2 * i + 1];
      d[ 2 * i + 1 ] = c[ 2 * i + 1 ]  + a [ 2 * i + 1] * b [ 2 * i ];
      d[ 2 * i + 1 ] = d[ 2 * i + 1 ]  + a [ 2 * i ] * b [ 2 * i + 1 ];
}
```

The LMPCC SIMD Vectorizer

- All arrays (except struct members) are aligned on 16 byte boundaries — simplifies loading and storing of vectors
- For non-aligned references still only one vector load is used — the loading of two vectors is pipelined so that the upper vector becomes the lower in the next iteration (or, vice versa, depending on the direction of the array traversal)
- When possible, LVSR/LVSL instructions are moved out of loops by SSAPRE — the address of the first array element accessed is used instead of the next element; this way the address is constant in the loop
- Rewriting of references to make alignment explicit — this way more redundancy is exposed