Optimization and Profiling on Vector Computer CONVEX 3820

SLIDES

Kandó Kálmán Polytechnic
Institute of Computer Technology, Székesfehérvár
and
University of Athens

Dr. Rezső Nagy, Associate Professor
Magda Kovács, Senior Lecturer

OPTIMIZATION

Optimization improves the performance of programs.

CONVEX FORTRAN compiler performs these functions:

- Eliminates unnecessary operations
- Arranges operations in the most efficient order
- Replaces slow operations with faster equivalents
- Takes full advantage of CONVEX architectures

OPTIMIZATION OPTIONS

Option	Description
-no	Machine-dependent scalar optimization. This option is default
-O0	Basic block machine-independent scalar optimization
-O1	Basic block and program unit machine-independent scalar optimization
-O2	Vector optimization
-O3	Parallel optimization

VECTOR OPTIMIZATION

Example:

```
DO I = 1, 100
   A(I) = B(I) + C(I)
ENDDO
```

At optimization level –O2, the compiler generates vector code to load 100 elements of B and 100 elements of C into vector registers, add them simultaneously, and store 100 resulting elements in A.

Think of the vector code as a pseudocode statement involving only four instructions:

A(1:100) = B(1:100) + C(1:100)

Where A(1:100) means A(1) trough A(1:100).

Strip mining

The vector registers hold up to 128 elements. When the iterations of a vectorizable loop are unknown or exceed 128, the compiler strip mines the loop before vectorizing it.

Strip mining replaces the original loop with two loops. The inner loop has an iteration count that never exceeds 128. The outer loop controls the number of times the inner loop is executed. The following example shows a loop that can be strip mined.

Original loop:

```
DO I = 1, N
   A(I) = B(I) + C(I)
ENDDO
```

In the vectorized loop code shown below, `IOUT` is a variable that the compiler uses to count the number of elements remaining to be processed, and the vector operations are shown using above. `I` is the starting index for each vector operation.

Vectorized loop:

```
I = 1
DO IOUT = N, 0, -128
   K = I + MIN(127, IOUT -1)
   A(I:K:1) = B(I:K:1) + C(I:K:1)
   I = I + 128
ENDDO
```

If N equals 300, IOUT is tested four times. For each comparison of IOUT to zero, the table below shows values of I and IOUT and the elements of array A that are calculated.

I	IOUT	Elements processed
1	300	1...128
129	172	129...256
257	44	257...300
385	-84	

The fourth test of IOUT fails, so the loop is not executed and no elements of A are processed.

Loop distribution

Vectorization is only done on simple loop nests. A simple loop nest is one in which all calculations are done in the innermost loop.

Nested loops, however, can be vectorized by distributing the outermost loop and vectorizing each of the resulting loops or loop nests. Consider the loop in the following example.

Original loop:

```
DO I = 1, N
   B(I, 1) = 0
   DO J = 1, M
      A(I) = A(I) + B(I, J)*C(I, J)
   ENDDO
   D(I) = E(I) + A(I)
ENDDO
```

Vectorized loop:

```
DO I = 1, N
   B(I, 1) = 0
ENDDO
DO I = 1, N
   DO J = 1, M
      A(I) = A(I) + B(I, J)*C(I, J)
   ENDDO
ENDDO
DO I = 1, N
   D(I) = E(I) + A(I)
ENDDO
```

C./ Loop interchange

The compiler interchanges nested loops for the following reasons:

- To make the loop that is the most profitable to vectorize the innermost loop
-
- To make the loop that is the most profitable to parallelize the outermost loop
-
- To make memory accesses to consecutive word in memory
-
- To make a loop with long vector length (iteration count) inside a loop with short vector length

Consider the matrix addition shown below:

Original loop:

```
DO I = 1, N
   DO J = 1, M
      A(I, J) = B(I, J) + C(I, J)
   ENDDO
ENDDO
```

Vectorized loop:

```
DO J = 1, M
   DO I = 1, N
      A(I, J) = B(I, J) + C(I, J)
   ENDDO
ENDDO
```

Parallel optimization

As an example of the transformations the compiler performs at optimization level -O3, consider the matrix multiplication shown below.

Example

```
DO I = 1, N
    DO J = 1, N
       C(I, J) = 0.0
       DO K = 1, N
          C(I, J) = C(I, J) + A(I, K) * B(K, J)
       ENDDO
    ENDDO
ENDDO
```

The compiler processes this loop nest by distributing the loop nest containing the I and J loops, as shown in the following example.

Example

```
DO I = 1, N
   DO J = 1, N
      C(I, J) = 0.0
   ENDDO
ENDDO
DO I = 1, N
   DO J = 1, N
      DO K = 1, N
         C(I, J) =  C(I, J) + A(I, K) * B(K, J)
      ENDDO
   ENDDO
ENDDO
```

The compiler moves the I loop to the innermost position in each nest, as shown below, so that it can retrieve contiguous elements on successive iterations.

Example

```
DO J = 1, N
   DO I = 1, N
      C(I, J) = 0.0
   ENDDO
ENDDO
DO J = 1, N
   DO K = 1, N
      DO I = 1, N
         C(I, J) = C(I, J) + A(I, K) * B(K, J)
      ENDDO
   ENDDO
ENDDO
```

The compiler strip mines both I loops to the optimal vector length, a function of the loop upper bound (N). In the following examples, MVSL represents that function, and V0 and V1 represent vector registers that can contain up to 128 64-bit elements.

Example

```
M = MVSL(N)
DO J = 1, N
   DO IOUTER = 1, N, M
     C(IOUTER:MIN(N, IOUTER + M - 1), J) = 0.0
   ENDDO
ENDDO
DO J = 1, N
   DO K = 1, N
      DO IOUTER = 1, N, M
        V0 = C(IOUTER:MIN(N, IOUTER + M - 1), J)
        V1 = A(IOUTER:MIN(N, IOUTER + M - 1), K)
        V0 = V0 + V1 * B(K, J)
        C(IOUTER:MIN(N, IOUTER + M - 1), J) = V0
       ENDDO
    ENDDO
 ENDDO
```

In the second nest, the compiler interchanges the IOUTER strip-mine loop outside of the K loop.

Example

```
M = MVSL(N)
DO J = 1, N
   DO IOUTER = 1, N, M
     C(IOUTER:MIN(N, IOUTER + M - 1), J) = 0.0
   ENDDO
ENDDO
DO J = 1, N
   DO IOUTER = 1,N, M
      DO K = 1, N
        V0 = C(IOUTER:MIN(N, IOUTER + M - 1), J)
        V1 = A(IOUTER:MIN(N, IOUTER + M - 1), K)
        V0 = V0 + V1 * B(K, J)
        C(IOUTER:MIN(N, IOUTER + M - 1), J) = V0
      ENDDO
   ENDDO
ENDDO
```

In the following example, PARALLEL DO represents a loop that can be processed by multiple CPUs.

Example

```
M = MVSL(N)
PARALLEL DO J = 1, N
   DO IOUTER = 1, N, M
     C(IOUTER:MIN(N, IOUTER + M - 1), J) = 0.0
   ENDDO
ENDDO
PARALLEL DO J = 1, N
   DO IOUTER = 1, N, M
      DO K = 1, N
        V0 = C(IOUTER:MIN(N, IOUTER + M - 1), J)
        V1 = A(IOUTER:MIN(N, IOUTER + M - 1), K)
        V0 = V0 + V1 * B(K, J)
        C(IOUTER:MIN(N, IOUTER + M - 1), J) = V0
      ENDDO
   ENDDO
ENDDO
```

As shown below, the compiler hoists a vector load and sinks a vector store out of the K loop. The remaining reference to vector V1 chains with the vector addition and vector multiplication in the next statement, resulting in even faster execution.

Example

```
M = MVSL(N)
PARALLEL DO J = 1, N
   DO IOUTER = 1, N, M
     C(IOUTER:MIN(N, IOUTER + M - 1), J) = 0.0
   ENDDO
ENDDO
PARALLEL DO J = 1, N
   DO IOUTER = 1, N, M
     V0 = C(IOUTER:MIN(N, IOUTER + M - 1), J)
      DO K = 1, N
        V1 = A(IOUTER:MIN(N, IOUTER + M - 1), K)
        V0 = V0 + V1 * B(K, J)
      ENDDO
     C(IOUTER:MIN(N, IOUTER + M - 1), J) = V0
    ENDDO
ENDDO
```

Profiling

A profiler is a software tool that measures the run-time performance of a program. Profiling typically measures the total time it takes each portion of your program to execute. Additional information may be measured as well, such as the number of times routines or loops are executed.

Running your program under a profiler's control enables the profiler to collect performance measurements. After the measurements are analyzed, the profiler reports the performance data.

Cxpa measure a program's entire execution time, reporting the total time spent in individual routines, loops, parallel regions, and basic blocks.

With Cxpa the program's executable must be modified for use with the profiler. When a program is compiled with a special compiler option, the resulting executable contains extra information that the profiler uses to collect performance data.

Once your program is compiled, you can run it under the profiler's control. The profiler gathers performance data during execution and reports the results.

Why use a profiler?

Profiling a program is different from most other development steps because it typically happens after a program is working correctly.

A good profiler can help you learn how your program executes and where that execution is spent. Using the information Cxpa provides, you can learn which routines and loops are slowest, and which are fastest.

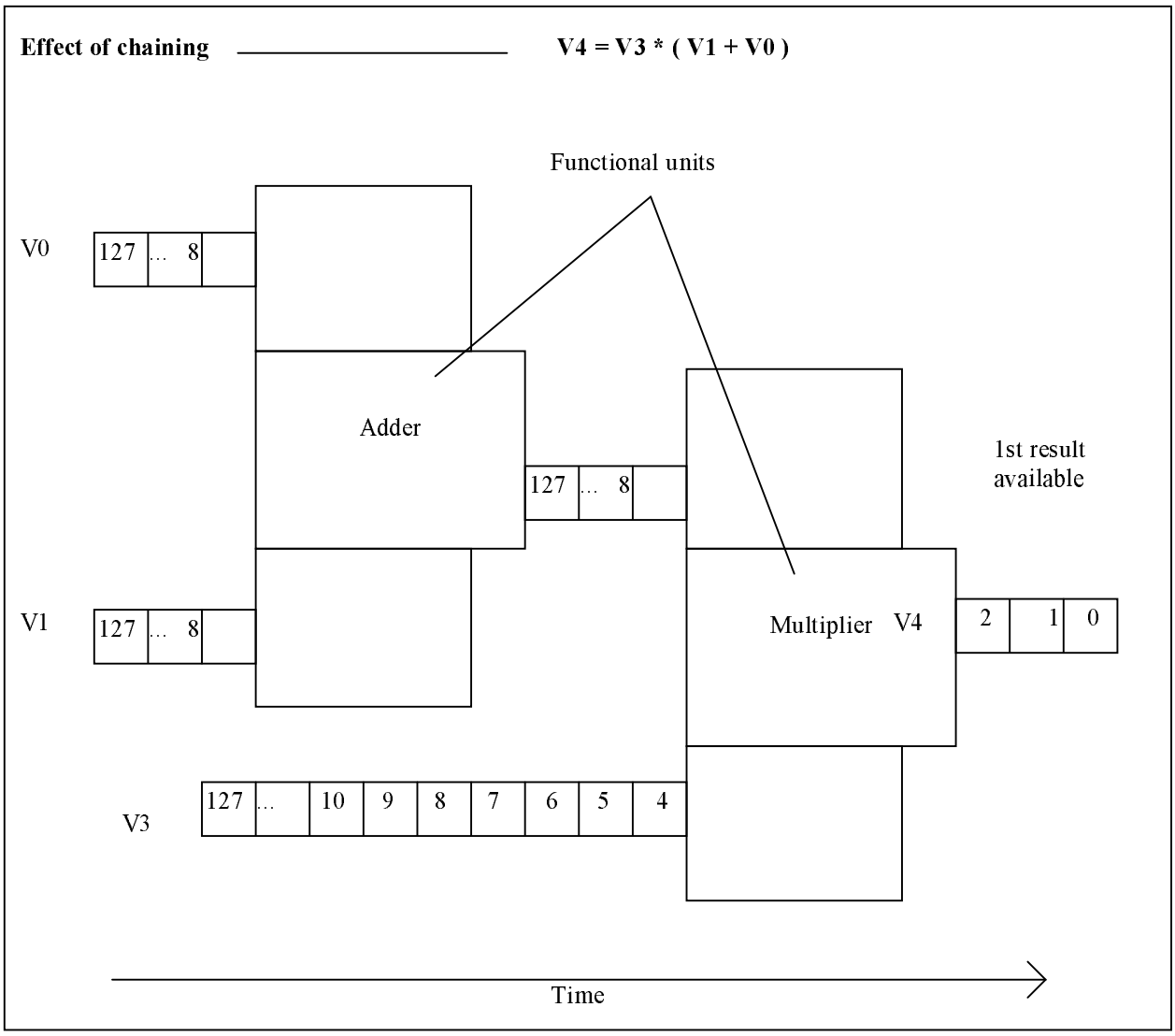
Profiling your program compiled at different optimization levels can provide insight into the types of optimizations that work best for given situations.

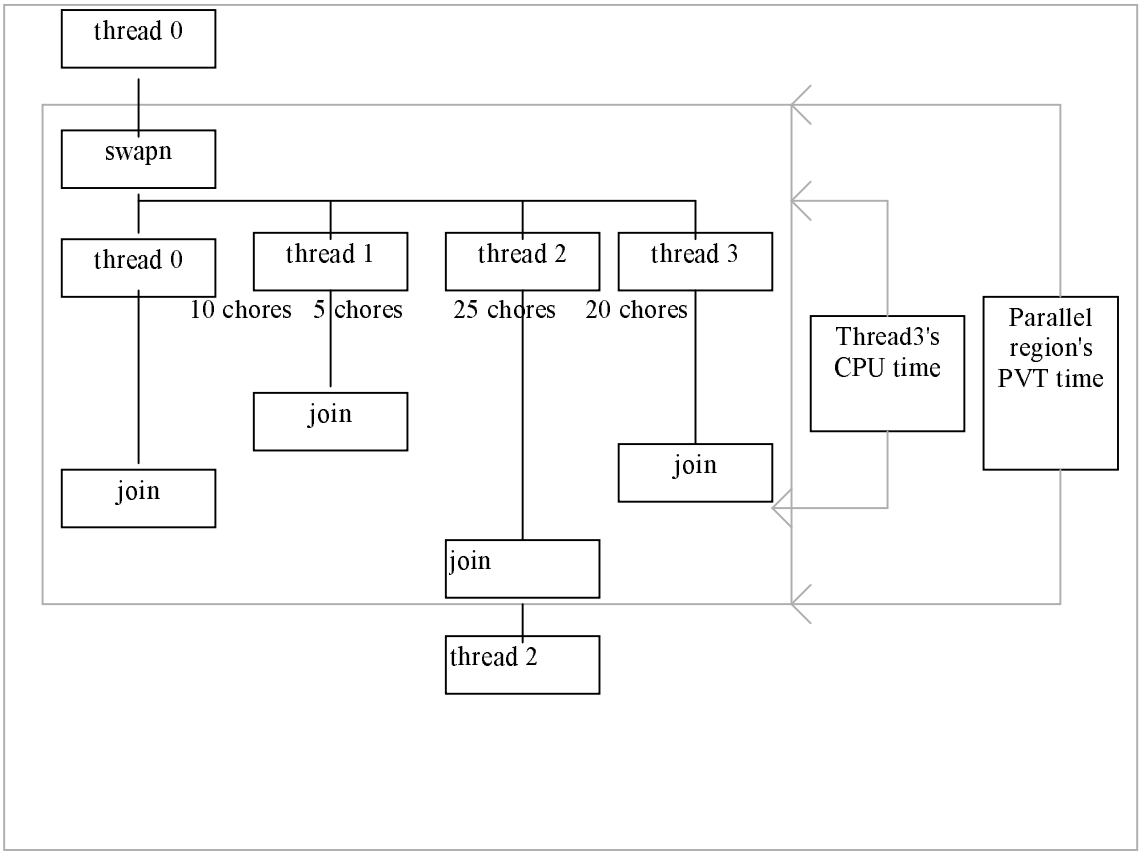
Vector chaining

Consider the following two vector instructions:

V2 = V1 + V0
V4 = V3 * V2

In this example, each element of V0 is added to the corresponding element of V1, and the result is stored into the corresponding element of V2. Then each element of V2 is multiplied with the corresponding element of V3, and the result is stored in V4. Without vector chaining, all of the elements of the first operation (the addition) wold have to be completed before the multiplication operation could begin.
With vector chining, The output of the addition operation can be fed directly into the multiplication operation. this is possible because the add operation and the multiply operation are calculated by two different functional units. This could be represented as:
V4 = V3 * (V1 + V0)




thread 0
swapn
thread 0
thread 1
thread 2
thread 3
10 chores
5 chores
25 chores
20 chores
join
join
join
join
thread 2
Thread3's
CPU time
Parallel
region's
PVT time