CSL 728

Compiler Design: Project Description

Energy optimal SIMD widths

Submitted by:

Akshay Singhal 2010CS10207

Ankur Garg 2010CS10208

Problem Definition

Given a program containing multiple loops, give optimal SIMD widths (W_1, W_2, W_3 …) for each loop (L_1, L_2, L_3 …).

SIMD – Single Instruction Multiple Data

Compare the energy results of flexible SIMD width structure with fixed width architecture.

Constraints:

W_i can take any value between 1 and 16

Approach

SIMD width for any loop depends on many factors:

- Loop Bounds
- True Dependencies: Read and Write Operations are occurring on the same array.
- Access Pattern: Same array is being accessed multiple times in a single iteration.

For calculating the SIMD widths, we take into account both memory and computation energies.

Energy Values for different operations are taken as follows for all energy calculations in the project:

Addition: 1

Multiplication: 5

Register Access: 0.5

Memory Access: 30

For SIMD width *W*, we multiply the above energy values by *W*.

Algorithm

To find optimal SIMD widths, we look at the possible cases that can occur in a loop. We analyse these cases individually and find an approach specific to that case.

Case 1: No True Dependencies

This is the case in which the array isn't being read and written in an iteration. An example of this case is:

```
for (i = 0; i < N; i++) {
        A[i] = B[i+5] + 4;
}
```

The example above is for single access. We first analyse this case. The case for multiple accesses can be handled in the same way.

There are different kind of optimizations which can be used in this case:

Loop Unfolding:

In cases where, for any particular array, all accesses made to that array are contiguous, loop unfolding (or vectorization) can be used to reduce number of memory accesses.

This can be used efficiently only when accesses to any particular array are contiguous.

For example:

```
for (i = 0; i < N; i++) {
        A[i] = B[i];
        C[i] = D[i];
}
```

This method decreases the number of iterations for which the loop runs by executing operations on multiple array indexes in one iteration. For example in the above for loop example, if we execute 2 iterations in 1, A[i] and A[i+1] will be accessed in one iteration.

Two Functional Units will be used to make changes to the array index vector of size 2.

The above code will get converted to:

```
for (i = 0; i < N; i+=2) {
        A[i] = B[i];
        A[i+1] = B[i+1]
        C[i] = D[i];
        C[i+1] = D[i+1];
}
```

Now, this for loop will run for N/2 iterations instead of N and it will perform operations on both A[i], A[i+1] and C[i], C[i+1] in one iteration. These are performed using vector operations. Since both these memory locations are contiguous, we can access them in one memory access and save memory accesses.

In cases, where array accesses for any particular array are not contiguous, as shown in the example below, we use a technique called Array interleaving so that we can use vectorization in this case as well. We show that using an example:

Initial Code:

```
for (i = 0; i < N; i+=4) {
        A[i] = B[i];
        C[i] = D[i];
        E[i] = F[i];
}
```

Array interleaving:

This method is used to interleave the arrays being accessed so that immediate array accesses are contiguous.

In the above case, the interleaving would be as follows:

W: A[0], C[0], E[0], A[1], C[1], E[1], …

R: B[0], D[0], F[0], B[1], D[1], F[1]…

In this way memory access for read operations on all the arrays in one iteration can be done in a single memory access as all the array values are present in contiguous locations in the memory.

Here, Width = 3 (number of arrays being accessed in one iteration of the loop)

The code after applying array interleaving becomes:

```
for (i = 0; i < N*W; i+=4*W) {

        W[i] = R[i];

        W[i+1] = R[i+1];

        W[i+2] = R[i+2];

}
```

In this case, the both read operations on the R can occur in single memory access. Similarly, both write operations can occur in single memory access.

Note:

- This array interleaving process can be used only when there are multiple arrays being accessed in a single iteration. Otherwise, if there are non-contiguous array accesses and there is only single array being accessed, we can't apply any of the above two optimisations.
- The array interleaving process may not be applied directly in all cases. One of the major problems is that, there can be multiple loops present in a program. For each loop, an array may be needed to be interleaved with a different array. For example, A may be needed to be interleaved with B in one loop and C in other. In that case, we can't use the method directly.
  One of the solutions to this problem is that we make a copy of the array A. Then we use these two copies of A to interleave with different arrays. We calculate if the overhead of making another copy of the same array is more or less than the profit we make in terms of memory accesses and we decide to use this accordingly.

Case 2: True Dependency

This case involves read and write operations on the same array.

- Single Access

  For example.

  ```
  for (i = 0; i < N; i++) {

      A[i] = A[i-1] + B[i];

  }
  ```

In this case the array is dependent on itself for the computations but the writing is done only once.
This is a case of multiple reading, single writing for a single array in an FU.

Here we can't optimize the energy by lowering the loop bounds because we need to access other indexes at the same time and they may get changed if we merge the iterations.

Here if we increase the W to 2, then we can access A[i] and A[i-1] at the same time resulting lesser number of memory accesses. This improves the efficiency of array accesses.

Now the FU needs to do only 2 memory accesses one for the array A and one for the array B.
Now our Algorithm will be looking into the index of array and calculating the difference between the accessed indices. If there was A[i-2] instead of the A[i-1] in the body of the loop, the algorithm would have suggested width 3 instead of the 2 so that A[i] and A[i-1] are accessed simultaneously in a single memory access.

- Multiple Access

For example:

```
for (i = 0; i < N; i+=2) {
    A[i] = A[i-2] + B[i];
    A[i-1] = A[i-3] + B[i-1];
}
```

In this case not only that the array is dependent on itself but it is accessing multiple indexes in the same FU.
This is a case of multiple reading, multiple writing of a single array in an FU.

This case is handled like the previous case but the width is 4.
In this case, the A[i], A[i-1], A[-2], A[i-3] can be accessed in a single memory access. This will reduce the number of memory accesses by 4 and hence improve the efficiency.

References

- Namita Sharma et al., “Data Memory Optimization in LTE downlink”, ICASSP 2013.
- Wikipedia Entry on Vectorization in parallel computing
  http://en.wikipedia.org/wiki/Vectorization_(parallel_computing)
- Compiler Optimization by Array Interleaving by Manjeet Dahiya, IIT Delhi
  http://www.cse.iitd.ac.in/~dahiya/CompilerOptimization_by_ArrayInterleaving.pdf