Compiler Optimization by Array Interleaving

Manjeet Dahiya
2012CSY7524
Indian Institute of Technology, Delhi

1 Introduction

If we interleave the arrays, a lot of vectorization can be made effective, that is not possible otherwise. Even for the code where vectorization is already there but suboptimal, array interleavings may make the vectorization more efficient.

This work presents such cases. Provides algorithms for locally finding these cases. We provide algorithms to handle the globally conflicting situation by duplication and/or heuristic based approach to solve Maximum Weighted Independent Set (MWIS).

We have implemented it in LLVM infrastructure. And tests show promising results for these cases under vector hardware.

Key contributions of this work are:
1. Locally finding cases where vectorization can be made effective by array interleaving.
2. Decision algorithms to handle interleaving conflicts based on duplication.

1.1 Array Interleaving

Arrays are stored contiguously in the memory e.g. an array $A$ of 10 integers $\text{int } A[10]$ will be stored in memory as:

\[A_0A_1A_2A_3A_4A_5A_6\ldots\ldotsA_9\]

If multiple arrays are involved, they will be stored independent of each other, contiguously at different memory locations. Array Interleaving is a data layout optimization technique in that multiple arrays are stored in memory in interleaved fashion e.g. two arrays $A$ and $B$ can be interleaved element by element alternatively in the memory the new layout will look like following:
When in loops operations to multiple arrays is involved we see some scope of optimization. *Array interleaving* may result in possibility of vectorization or making the vectorization more efficient.

### 1.2 Vectorization

Vector computer architectures are available which support parallel operations. For example vector computer architecture of width 4 can do 4 multiplications in one cycle. Two vectors containing 4 numbers each can all be multiplied in just one cycle. All four multiplications will be done in parallel.

Loading these *vector registers* is a challenge. If all the elements to be filled lie contiguously in memory then the loading takes less time in comparison to loading all elements from different locations.

```c
int A[N];
for (int i = 0; i < N; ++i)
{
}
```

In the above code the loop is being called $N$ times i.e. the stride is one. One operation is being done in each loop iteration. This code can be easily vectorized to do 4 operations at once, the stride will be 4. The loop will be called $N/4$ times. Following code is the equivalent vectorized code.

```c
int SIMD_W = 4;
int A[N];
for (int i = 0; i < N; i = i + 4)
{
  vec v = vec_ld(&A, i, SIMD_W);
  v = v + vec(1, 1, 1, 1);
}
```

### 2 Optimization Ideas

There are two ways to start on a compiler optimization, in first there are various examples and cases where optimization is possible, one tries to build a generic optimization algorithm based on it. Second way is starting from a general case itself from the beginning. This optimization is done the first way. We start with examples where optimization is possible by using *array interleaving* in the context of *vector registers*. 2
Let us discuss two cases in the same family of loops involving arrays. The difference is that the array accesses in the first case is contiguous i.e. that stride of the loop is 1. In second it is other way, the stride is more than 1. We will see that the first case is already being handled by todays compilers [3, 4] whereas for the second case our optimization idea can help.

Case 1: Contiguous array accesses, stride = 1

Vectorization comes into picture when dealing with arrays in loops. For example in the following code segment we have a for loop and we load [store] from [to] some arrays. The stride of the loop is 1 and we can see that the access to the arrays are made contiguously. This makes vector loads and stores cheaper than the scalar counter parts.

Compilers generally have the ability to vectorize this type of code [3, 4].

Listing 3: Vector operations

```
for(int i = 0; i < N; ++i)
{
    // we load from some arrays
    // A1, B1, ... A2, B2, ...
    // and we store to some arrays
    // A, B, ...

    B[i] = B1[i] + B2[i];
    C[i] = C1[i] + C2[i];
    D[i] = D1[i] + D2[i];
}

// after vectorization by compiler
for(int i = 0; i < N; i = i + 4)
{
    vec vA1 = vec_ld(&A1, i, SIMD_W);
    vec vB1 = vec_ld(&B1, i, SIMD_W);
    vec vC1 = vec_ld(&C1, i, SIMD_W);
    vec vD1 = vec_ld(&D1, i, SIMD_W);

    vec vA2 = vec_ld(&A2, i, SIMD_W);
    vec vB2 = vec_ld(&B2, i, SIMD_W);
    vec vC2 = vec_ld(&C2, i, SIMD_W);
    vec vD2 = vec_ld(&D2, i, SIMD_W);

    vec_st(&A, i, vA1 + vA2);
    vec_st(&B, i, vA1 + vA2);
    vec_st(&C, i, vA1 + vA2);
    vec_st(&D, i, vA1 + vA2);
}
```
Case 1: Non-contiguous array accesses, stride > 1

Above method doesn’t result in effective vectorization when the stride is not equal to 1 i.e. when jumps are present or non-contiguous access. Vector loads some elements which are not needed for computation hence results in poor vectorization.

For example changing the stride to 8 in the example of last subsection will give us:

Listing 4: Vector operations

```c
for (int i = 0; i < N; i = i + 8)
{
    B[i] = B1[i] + B2[i];
    C[i] = C1[i] + C2[i];
    D[i] = D1[i] + D2[i];
}
```

Vectorization is not effective here. During every vector load, store or vector ALU operation we will be wasting work of 7 elements (for SIMD Width = 8).

One can’t be optimize using vectorization. However array interleaving can enable vectorization for these family of cases.

On interleaving the above arrays in following manner enables effective vectorization:

W1 : A₀B₀C₀D₀A₁B₁C₁D₁, .......
R1 : A₁₀B₁₀C₁₀D₁₀A₁₁B₁₁C₁₁D₁₁, .......
R2 : A₂₀B₂₀C₂₀D₂₀A₂₁B₂₁C₂₁D₂₁, .......

We can rewrite the code to:

Listing 5: Vector operations

```c
for (int i = 0; i < N*4; i = i + 8*4)
{
    vec vR1 = vec_ld(&R1, i, SIMD_W);
    vec vR2 = vec_ld(&R2, i, SIMD_W);
    vec_st(&W1, i, vR1 + vR2);
}
```

3 Interleaving Issues

3.1 Multiple Accesses

Apart from the loop in question, the arrays might be accessed anywhere else also. If we interleave the arrays then we need to fix the all the accesses of
the arrays that have been interleaved. This will maintain the correctness of
the program.

Listing 6: Multiple array access

```c
int A[64];
int B[64];
for(int i = 0; i < 64; ++i)
{
    A[i] = B[i] + 1;
}

int get_ith_A(int i)
{
    return A[i];
}
```

The correct conversion should look at all the access of the arrays that
are interleaved.

Listing 7: Correct Impl of Multiple array access

```c
struct AB
{
    int a;
    int b;
};

struct AB intrlv_AB[64];
for(int i = 0; i < 64; ++i)
{
    intrlv_AB[i].a = intrlv_AB[i].b + 1;
}

int get_ith_A(int i)
{
    intrlv_AB[i].a = i;
}
```

3.2 Interleaving Conflicts

We can have a conflicting situations where multiple loops will demand in-
terleaving of arrays such that common arrays are present e.g. say Loop1
demands arrays A & B to be interleaved whereas Loop2 wants A & C to
be interleaved. This will result in a conflict A can be interleaved with only
one of B or C.

One must handle these cases by a suitable cost model. One can also
duplicate the common array so that same array can be interleaved with
multiple arrays. Again, one must do the analysis whether the benefit is
more than the cost of duplicating and synchronizing the array data.
3.3 Multiple Loops
We may have multiple loops accessing the arrays. Interleaving arrays for a particular loop may optimize for that loop. For other loops accessing these array it may lead to performance poor than the normal situation. For example in we have two same loops, one has stride > 1 and other has stride = 1. If we interleave for the first one the second one gets affected. It will disable vectorization for second.

Again one has to design a suitable cost model for the same. If there are more number of calls to the first loop then one must interleave.

3.4 Loops Under Conditionals
What is loop is present in a branch e.g. in an if-else block? It may or may not be called. How should these loops be taken in account into the cost model?

4 Algorithms
The program will have multiple loops accessing multiple arrays. The implementation should choose interleavings that are best globally.

A cost model can be defined for each loop and its array interleavings. The algorithm will find the set of array interleaving which has the optimized cost overall.

4.1 Profit Per Interleaving
These interleavings are enabling code vectorization. We will have to calculate the benefit we get over scalar code.

Memory
Memory access time is less if accesses are contiguous. One can write following formula for memory access time:
\[
\text{time} = b + K
\]
\[
b = \text{number of blocks}
\]
\[
K = \text{constant}
\]

ALU Operations
If the width of SIMD is 8. One can do 8 scalar operations in one go. One can greatly reduce the time required to do computations.
4.2 Global Profit Maximization

There can be conflicting interleavings as mentioned in subsection 3.2. To resolve these, one can duplicate the arrays if profitable. Otherwise one has to remove some interleavings to resolve conflicts.

We discuss both the cases in following two subsections.

4.2.1 Duplicating The Arrays

Say we have an interleaving and it requires some number of arrays to be duplicated. We can find the cost of duplicating these arrays by finding the number of memory writes to these arrays at other places in the code.

\[ M = \text{number of arrays to be duplicated} \]
\[ P = \text{profit of this interleaving} \]

\[ \text{Duplication cost} = \text{total number of memory writes to these } M \text{ arrays at other places in the code} \]

Duplication is done if Duplication cost is less than \( P \).

Algorithm

Listing 8: Duplication Decision Algorithm

```c
void duplication()
{
    bool change = true;
    while(change) // keep doing until there is no change
    {
        change = false;
        for(itr over conflicting interleavings)
        {
            if(duplicationCost(itr) < profit(itr))
            {
                duplicateConflictingArrays(itr);
                change = true;
            }
        }
    }
}
```

4.2.2 Conflict Graph

There may be conflicts left after the duplication because duplication may not be possible for all the cases. Among these conflicting interleavings one will have to choose the ones to get globally optimal profit without any conflicts. This turns out to be finding the independent set of a graph.
We can represent conflicts and interleavings as a graph. Nodes represent the interleavings with the weights as the *profit*. The edges between nodes represent conflicts. Our goal is to maximize the total profit such that there are no conflicts.

Figure 1: Conflict graph of interleavings

In the figure 1 $w_i$ represent profits and edges connecting nodes represent conflicts. $W_5, W_6$ and $W_9$ don’t conflict with any interleaving and will always be chosen. Among other nodes we have to decide which nodes to delete so that there are no conflicts (edges) and profit is maximum.

This problem is same as finding the maximum independent set in weighted graph. The problem is called Maximum Weighted Independent Set (*MWIS*). This problem is NP-Hard i.e. no polynomial time algorithm exist. There doesn’t exist any approximation algorithm either [5, 6, 7].

**Algorithm**

We want to have a polynomial time algorithm for *MWIS*. We propose an algorithm based on heuristics which is polynomial. We define a quantity *deletion weight*.

\[ \text{deletionWeight} = \frac{\text{degree}}{\text{profit}} \]

*Deletion weight* is proportional to degree because we want many edges to be deleted. So that fewer conflicts are left. And it is inversely proportional to profit because we don’t want to delete nodes with high profit.

The conflicting node with the highest value for this quantity will be picked and deleted. And then we apply the duplication algorithm again because some duplications may now be feasible after deleting this node. And we keep repeating these two steps until all the nodes with *degree* > 0 are deleted.
void globalConflictResolution()
{
    while (il = getMaximumDeletionWeightNode())
    {
        delete_node(il);
        duplicationDecision();
    }
}

node* getMaximumDeletionWeightNode()
{
    double maxDeletionWeight = 0;
    node* result = NULL;
    for (itr over conflicting nodes)
    {
        double profit = getProfit(itr);
        double degree = getDegree(itr);
        double deletionWeight = degree / profit;
        if (maxDeletionWeight < deletionWeight)
        {
            maxDeletionWeight = deletionWeight;
            result = itr;
        }
    }
    return result;
}

References


- S. Basagni: Telecommunication Systems Volume 18, Issue 1-3, pp 155-168