Optimizing Array Locality via Memory Layout Reorganization

Xuezhi Wang (xuezhiw@cs.cmu.edu), Junchen Jiang (junchenj@cs.cmu.edu)

Project webpage: www.cs.cmu.edu/~junchenj/15745-proj/index.html

Project description:

When optimizing array locality for cache performance in a loop, a conventional approach is tiling which keeps the current memory layout of array but change execution order of instructions so that cached content will be reused for future execution before they are replaced. However, this scheme may fail when the execution reordering is unavailable due to data dependency or other constraints.

In this project, we propose an alternative approach for optimizing array locality through reorganizing memory layout of arrays. The main idea is that by analyzing the access pattern of array elements, we can re-organize the array memory layout such that the new memory layout can achieve a better locality even with no changes on execution order or prefetching (though it can potentially be used together with tiling to further improve the performance).

A simple example of using the new method is following:

```
for(i=0; i<n; i++){
  A[i][0] += 1;
  A[i][1] += 2;
}
```

In this program, assuming cache can hold 2 elements, a naive execution (no tiling) will cause cache miss on every access. In our method, we can reorganize the 2-d array `A[n][2]` into 1-d array `B[2n]` such that `B[0]=A[0][0], B[1]=A[0][1], B[2]=A[1][0], …` and the cache miss rate will be reduced to half (100% to 50%) of that in a naive execution with no change on the order of execution.

In case there are multiple access patterns (e.g., multiple loops for the same array), we consider the following two options:
(1) For each access pattern, create a suitable memory layout and keep this newest layout(s) (keep multiple copies only when necessary, e.g. for branches);
(2) Find the most suitable memory layout for all access patterns, and hopefully it should be better than the original layout (which does not take access patterns into consideration).
It is even possible that which option is better is dynamically decided by the access patterns in

the code.

The key difference between tiling and our method is, tiling changes the execution order of instructions, but our method changes the memory layout and can also be used in the case that instruction order should be fixed during program execution. Certainly our method can be used in the cases where tiling is used. Another example where our method might be more useful would be, in a program involving regular expression matching, if we use an automaton to store the transition states, the order of state transition in actual execution cannot be changed (i.e., tiling is not applicable), but we can reorganize the order of state storage by analyzing the degree/transition relations of each state.

75% goal: Finish the codebase which can detect access patterns and reorganize array memory layout. Finish evaluation of the proposed scheme based on the basic analysis.

100% goal: Finish the codebase which can detect access patterns and reorganize integer array memory layout. Finish analysis for multiple access patterns in the code, decide which option is better (or should be decided dynamically based on the code). Finish evaluation of the proposed scheme on the benchmark.

125% goal: Based on 100% goal, extend the application to other type of arrays (e.g., struct array).

Logistics

Plan of attack and schedule

Mar 27-Apr 10: Implement code to detect access patterns of arrays in programs. [Junchen, Xuezhi]
Apr 10-Apr 15: Implement code to reorganize memory layout based on different access patterns [Junchen, Xuezhi].
Apr 16-Apr 18: Use benchmark for basic test. Finish milestone report. [Junchen]
Apr 19-Apr 27: Implement code for programs that have multiple access patterns. [Junchen, Xuezhi]. Test the two options on the benchmarks. [Xuezhi]
Apr 27-Apr 30: Write-up final report. [Junchen, Xuezhi]

Milestone

A reasonable goal of Apr. 18 will be to complete code to detect access patterns, and finish code for memory reorganization based on the access patterns analyzed. Also we should finish some basic analyses/tests (cache miss rate, efficiency) using the benchmark.

Literature search

Beginning from [1], many related works have been published on tiling optimization for loops. Recent developments mainly focus on tiling size selection (e.g., [3, 4]) and specific hardware support for blocked optimization (e.g., [2]). There are also works on optimization of loop performance using hardware support without tiling (e.g., [5]). Another paradigm is through exploiting array and loop parallelism through dynamic analysis (e.g., [6]) and speculative execution(e.g., [7]).

Resources needed

We will basically use LLVM on the same Ubuntu image as in the homework. We will use SPEC benchmark (http://www.spec.org/benchmarks.html) to test our proposed methods.

Getting started

We have read several research papers regarding this topic and we already investigated what work has been done in this area.

References

[1] M. J. Wolfe. More iteration space tiling.
[2] A Compiler Framework for Optimization of Afne Loop Nests for GPGPUs, L Zuck, A Pnueli, B Goldberg, C Barrett, Y Fang
[3] Automatic Creation of Tile Size Selection Models, Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre Eichenberger and Kevin O'Brien.
[4] Parameterized Tiling Revisited, Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Tom Henretty, J Ramanujam, and P Sadayappan
[5] Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. S Ryoo, CI Rodrigues, SS Baghsorkhi
[6] Logical inference techniques for loop parallelization. Cosmin E. Oancea and Lawrence Rauchwerger.
[7] Speculative separation for privatization and reductions. Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August.