Restructuring Memory Access for Optimal Cache Performance

Prashanth Suresh(psuresh@andrew.cmu.edu),
Zhe Qian(zheq@andrew.cmu.edu),
Adu Bhandaru(akbhanda@andrew.cmu.edu)

http://www.andrew.cmu.edu/user/psuresh/Optimizing_compilers/

Project Description

Memory access is a major bottleneck in instruction execution by the processor. Caches enable this to be done with reduced access time. The predominant way by which cache access is optimized by the compilers is by choosing the optimal tile size to enable optimal cache utilization.

Consider the case of multiplication of 2D matrices; the elements are accessed in a sequential manner.

```plaintext
for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
        for(k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
```

(a) matmul

```plaintext
for (jT=0; jT< N/J; jT++)
    for (kT=0; kT< N/K; kT++)
        for (i = 0; i < N ; i++)
            for (k = K*kT ; k < K*kT + K-1 ; k++)
                /* vector loop */
                for (j = J*jT ; j < J*J + J-1 ; j++)
                    C[i][j] += A[i][k] * B[k][j];
```

(b) 2D-tiled matmul

Here misses occur for every element access of matrix B and comparatively lesser misses for the other 2. For the 2D tiled matmul the matrix have better hit rates since within the particular tile of size T, it has all hits as they belong to a single memory block in the cache. The optimization is done by choosing a tile size based on an algorithm that takes the matrix size and cache size as input.

Now the address allocation for the each of the matrices is done in a sequential manner for each element. Since the addressing of memory is done in an arbitrary fashion it may cause cache thrashing by using the same set for different matrix data. For example some elements of matrix A may need to share the same set and cache line with that of matrix B since cache sets are allocated based on the middle bits of the memory address. This may cause the cache to exhibit conflict misses and cause back-n-forth data movement or ping ponging of data in the cache.

We plan to approach this in 2 ways. The first way is by reorganizing memory such that we can
avoid access collisions within the same set in a cache. This method avoids cache thrashing between the 3 matrices used above. The second way is by making use of temporary registers so that we can make the maximum use of a particular cache, perform the operation and store them all simultaneously. Here the focus is on combining the memory operations on a particular matrix i.e loads and stores so that all blocks in a cache line are utilized.

In the example above, if the maximum number of registers used is R, then for the worst case scenario of all matrices sharing some cache lines, misses occur for every alternate access of A, B and C. Combining the loads and stores theoretically should reduce the miss count by a fraction of R over tile size.

Matrices are generally stored in memory in a row major order. Hence this can be utilized to access data in a contiguous way. Hence choosing the access pattern for a particular memory array operation can provide a significant improvement in cache-hit performance. This can be selected by using data locality and distribution information and classifying them accordingly. We plan to assess different features, which may contribute to determining the optimal access pattern for a particular array access. The compiler can appropriately unroll loops and form the access pattern.

Additionally we will try to modify the above to enable parallel cache access by different threads so that fetched data can be shared among the threads and pingponging does not occur. We can make use of the cache set associativity at different cache levels to effectively classify access patterns to guarantee spatial and temporal locality.

75% Goal- Analyze and eliminate cache line collisions and thrashing by different matrices.
100% Goal- List different features required for determining access patterns and create a pass to use them.
125% goal - Parallelized cache performance analysis.

**Logistics:**

**Plan of Attack and Schedule**

Week 1 (03/23/14 to 03/29/14) Form an algorithm for the cache collision and analyze extent of set overlap among matrices.

Week 2 & 3 (04/30 to 04/12) Naïve implementations of the findings from previous weeks’ and get the pass working. Simultaneously analyze different array access and the access patterns that might be favorable to each.

Week 4 (04/13 to 04/19) Implementation of final passes and a summarized model of the access patterns.

Week 5 (04/19 to 04/30) Final benchmarks and project write-up.

**Milestone**
By the milestone we hope to have a pass, which prevents cache set collision and a comprehensive model of the access patterns.

**Literature Search**  
Read different papers on tile size selection and algorithms used to calculate them. Additionally read about data distribution patterns and papers on compiler parallelization techniques and compiler managed memory systems.

**Resources Needed**  
Planning to basically use the llvm framework used in class assignments.

**Getting Started**  
Currently we are reading different papers on tile selection and tools that can be used to represent the cache.

**References**  
2. *Data Access Type Aware Replacement Policy for Cache Clustering Organization of Chip Multiprocessors* by Chongmin Li, Dongsheng Wang, Haixia Wang, Guohong Li, Yibo Xue.  
5. *Compiler Directed Data Locality Optimization for Multicore Architectures* by Wei Ding.