Analyzing Software Cache Configuration for In-line Data Compression

Sansriti Ranjan, Jon Calhoun (Advisor) Holcombe Department of Electrical and Computer Engineering - Clemson University

Why do we need caching for inline-compression?

Large HPC applications use compression techniques to reduce the memory requirements for storing large allocated arrays. Naive inline compression compresses and decompresses for each memory access which significantly hurts performance. Caching the decompressed values in a software managed cache limits the number of compression and decompression operations. This helps in improving the performance of the application.

Experimental Setup

Cache Simulator

Parameters

- Cache Size
- Block Size
- Cache Policy
- Cache Hit
- Hit Rate
- Miss Rate

Applications

- Matrix-Matrix multiplication (Matmul)
- Fast Fourier Transform (FFT)
- Jacobi
- Dijkstra
- Sparse Matrix-Vector Multiplication (SpMV)

The Interface

- Load function: Implemented for each read action being performed for the array index
- Store function: Implemented for each write action being performed for the array index
- Dimensions – n dimensions for k elements in N blocks

Workflow

- Address trace generated by running the applications & getting array index being accessed
- For each index accessed from the decompressed array, the appropriate action – load/store is implemented and the cache hit, or cache miss is updated

Cache Simulator

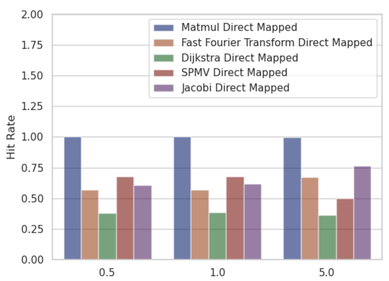
Current hardware caches operate on linearized memory arrays.
Our simulator incorporates software caches for compression with multi-dimensional arrays.

Goal: Use the Cache Simulator to understand what cache configuration works best for each application in terms of performance.

Improving the Performance by tuning the Cache Size and Policy

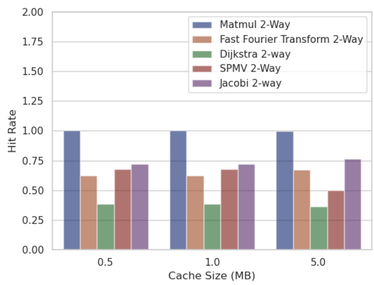
How does performance vary as the parameter such as cache size is changed?

Direct Mapped Cache



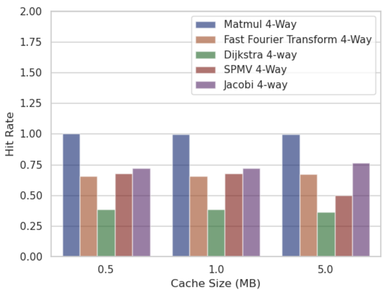
For a **direct mapped cache**, **Matmul** and **Jacobi** give the best performance, and with an increase in cache size, their performance improves but **Dijkstra** and **SpMV** reduce in performance.
Dijkstra and SpMV have low spatial locality and random-access patterns leading to **aliasing** and consequently a drop in performance.

Two-Way Set Associative Cache



With an increase in cache policy to a **two-way set associative cache,** performance for **Dijkstra** and **SpMV** improves relatively but reduces with an increase in cache size. This cache policy helps attain an improvement in performance by 5.3% for **Matmul, FFT** and **Jacobi**.
For Dijkstra and SpMV, we see an improvement of 2%.

Four-Way Set Associative Cache



With a further increase in cache policy to a **four-way set associative cache, Matmul** sees the best improvement in performance (7%) followed by **Jacobi** (6.8%). **Dijkstra** and **SpMV** reduce in performance on increasing the cache size.

Conclusion

Optimal Cache configuration using Cache Size and Cache Policy can be obtained for each application depending on the application's dimensionality and access patterns. Improvement in performance by increasing the Cache Size and Cache Policy. Certain applications require a trade-off between the two to optimize performance.

Future Work

Quantify the speedup further in the application by using the best cache Configuration and improve the performance for Applications with aliasing and random-access patterns.

More results and videos
http://jonccal.people.clemson.edu/compression/compression.php

3https://github.com/FTHPC/Compressed_Cache_Simulator