Cache/locality - aware software

Review: Locality mechanism of CPU

Load/store architecture

- Only load and store instructions can access memory.
- Once data is loaded from memory to registers, and then processors typically use them multiple times. (temporal locality)
- When loading data from main memory, processors read data size of one cache line at once. (spatial locality). Temporal locality, i.e., reuse, increases arithmetic intensity, but spatial locality just reduces latency (and has other benefits we'll discuss another time).

Modern CPUs have several levels of hierarchy. Why? Speed vs. Size.

- Registers – Caches (multiple levels inside it) – Main memory (external) – Disks.
- Higher level (ex. Register): Lower latency, but smaller storage size.
- Lower level (ex. Disk): Higher latency, but larger storage size.
- Exploiting locality is the way to use the hierarchical structure more efficiently.

Performance metrics to measure the speed.

- Latency: Closer device (higher level of hierarchy) has lower latency. Placement is important.
- Bandwidth: Related to the capability of the interface. Bandwidth of off-chip devices are limited by capability of interface, typically wires, so there is a big difference between on-chip devices and off-chip devices. However, within a category, there is no significant of difference.

Locality Metrics

$$Locality = \frac{number\ of\ words\ accessed\ locally}{number\ of\ words\ accessed\ in\ total}$$ (the value is between 0 and 1)

$$Arithmetic\ Intensity = \frac{number\ of\ operations}{number\ of\ words\ from\ non{-}local\ storage}$$

$$\textit{Cache Hit Metrics} = \frac{\textit{number of accesses that hit}}{\textit{number of total accesses of cache}}$$

- Our locality metric might go down with large cache lines, but the cache hit rate is still high
    - If we have large cache lines, but only access the first element within the line, then the hit rate is high, but the locality metric is low.
- Hit rate is a measure of latency
- Locality is a measure of bandwidth/volume
- Locality increases as capacity goes up. However, locality eventually levels off after a certain storage capacity.

Can also consider metrics directly related to *physical locality* as distance, although usually a hierarchical approach is easier and sufficient. However, depending on the topology of the interconnect between the different components, physical locality is sometimes quite complex. More on that in later lectures.

We also talked about *correlation locality* which is a generalization of traditional spatial locality as it refers to the memory hierarchy. An interesting paper discussing this in some detail is Peled et al., "Semantic locality and context-based prefetching using reinforcement learning", ISCA 2015 .

Cache-aware software

- To obtain performance, the program should be aware of the cache architecture and parameters.

Example: Dense Matrix-Matrix Multiplication

- Problem to solve: C (MxN matrix) = A (MxK matrix) * B (KxN matrix)
- Assumption 1: For simplicity, all matrices are NxN square matrices.
- Assumption 2: Matrices are stored in row major order (as in C/C++, some languages like Fortran use column major order).

Original program.

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

- Because one cache line is typically 64 or 128 bytes long, one cache line contains multiple elements of a given matrix (spatial locality).
- Lots of cache misses in this code. Locality (arithmetic intensity) is ops / misses * line-size.
    - Computation is roughly 2*N^3 C-arithmetic-operations (many more instructions after compilation, but "C ops" is a nice easy way to count.
- Miss analysis:
    - Matrix C exploits temporal locality: Calculation for a particular element of C is completely done at one time. Matrix C also exploits spatial locality because update is done in the order it stored.
    - Matrix A benefits from cache hits because of spatial locality .
    - However, B[k][j] and B[k+1][j] are separated by N, so every iteration of the inner-most loop incurs a cache miss. Data in a newly loaded cache line is not reused, but just wastes cache memory, and also evicts data of A.

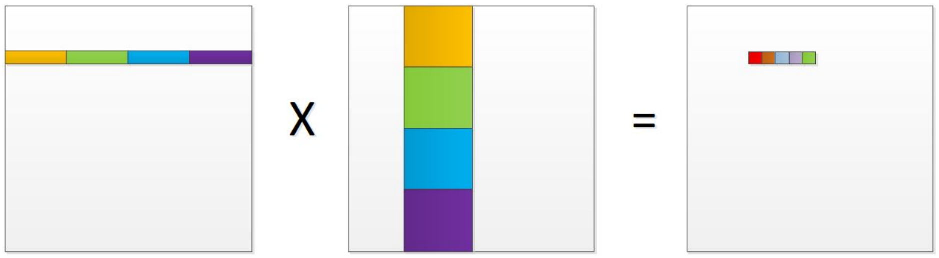
Solution #1: Swapping the order of loops to exploit locality.

```
// assume C initialized to all zeros

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

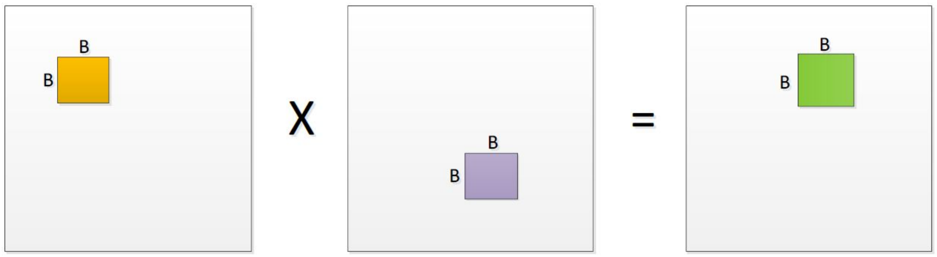
- Matrix A exploits temporal locality as well as spatial locality.
- Matrix B also benefits from spatial locality because its access pattern is changed from column direction to row direction.
- Matrix C loses temporal locality, but holds spatial locality.
- However, if the row size is much larger than a cache line size, it does not work that well.

Solution #2: Divide a row or a column into small-sized blocks, which fit into a given cache size, and then calculate its partial result.



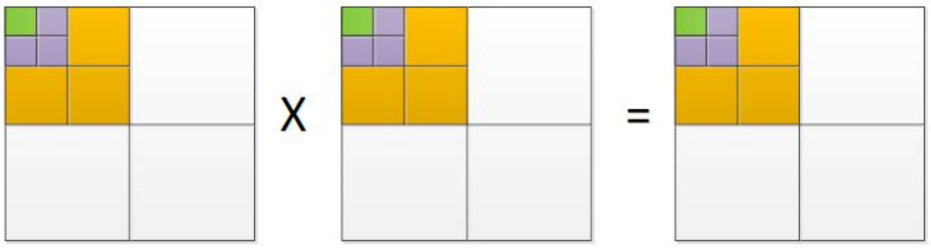
- The final result can be obtained by accumulating the partial results. Break-up is started from the most inner loop to exploit locality.

Solution #3: Break up the original matrix into several small-sized matrices.



- It is possible to reuse each small-sized chunk.
- Assumed the block size is BxB, then total number of cache accesses becomes, $(B^2 * N/B * 2 + B^2) * N^2/B^2$
- With the original code, there are N3-times MADD (Multiply and Add) operations, thus number of cache accesses becomes $4 * N^3$. (D = A * B + C, A/B/C/D are requested to read from memory)
- Therefore, the arithmetic intensity is $(2 * N^3) / (((B^2 * N/B * 2 + B^2) * N^2/B^2) \approx 2B$, if N is large enough. It means large B is beneficial to achieve higher hit rate.
- B can be large up to sqrt(cache size/3) in the ideal condition. ($3 * B^2 <$ cache size) However, cache is non-ideal, means that not fully associative, cache line, so B should be much smaller.

Solution #4: Blocks within a block.



- Latency is important to achieve high performance, so block size should be small enough to insert in a L1 cache. However, if block size is determined by L1 cache size, which is relatively small, it needs more off-chip bandwidth, so block size should be large enough to fit into LLC size as well. These conflicting requirements lead to multiple levels of blocking that match the locality hierarchy.
- Using blocks in a block is good way to exploit both latency and bandwidth. While loading a big sized block into LLC to reduce off-chip bandwidth, CPU can compute with smaller sized block fit into L1 cache to exploit its lower latency.

Cache-oblivious Software

Cache-aware software is highly dependent on the details of the machine, for example, cache size of each level, and number of registers. To avoid it, oblivious algorithm can be used.

A cache-oblivious algorithm asymptotically-matches (Big-O) the locality of the best cache-aware algorithm on the same system.

This algorithm recursively breaks down the block into very small blocks. Recursive algorithms are often a good example of cache-oblivious algorithms. The cache-oblivious and cache-aware matrix-multiplication programs access matrices in a different order, but a different order of instructions is legal since addition is associative and commutative (not necessarily true for floating point)

The cache-oblivious algorithm recursively decomposes the input matrices into smaller matrices, by a procedure called leaf-unfolding.

```
void recur(int i0, int i1,int j0, int j1, int k0, int k1)
    {
        int i, j, k, di = i1 - i0, dj = j1 - j0, dk = k1 - k0;
        const int CUTOFF = 8;
        if (di >= dj && di >= dk && di > CUTOFF) {
            int im = (i0 + i1) / 2;
            recur(i0,im,j0,j1,k0,k1);
            recur(im,i1,j0,j1,k0,k1);
        } else if (dj >= dk && dj > CUTOFF) {
            int jm = (j0 + j1) / 2;
            recur(i0,i1,j0,jm,k0,k1);
            recur(i0,i1,jm,j1,k0,k1);
        } else if (dk > CUTOFF) {
            int km = (k0 + k1) / 2;
            recur(i0,i1,j0,j1,k0,km);
            recur(i0,i1,j0,j1,km,k1);
        } else {
            for (i = i0; i < i1; ++i)
                for (j = j0; j < j1; ++j)
                    for (k = k0; k < k1; ++k)
                        C[i][j] += A[i][k] * B[k][j];
        }
    }
```

[$[Get Code]]

The various levels of the function call tree are listed below.

Level 1

When recur() is invoked with the given arguments, this statement executes first, making two calls to recur() -

```
if (di >= dj && di >= dk && di > CUTOFF) {
            int im = (i0 + i1) / 2;
            recur(i0,im,j0,j1,k0,k1);
            recur(im,i1,j0,j1,k0,k1);
    }
```

[$[Get Code]]

Level2

Each of the two calls to recur() executes this portion of code -

```
else if (dj >= dk && dj > CUTOFF) {
            int jm = (j0 + j1) / 2;
            recur(i0,i1,j0,jm,k0,k1);
            recur(i0,i1,jm,j1,k0,k1);
    }
```

[$[Get Code]]

Level 3

Each of the 4 calls to recur() executes this portion of code -

```
else if (dk > CUTOFF) {
            int km = (k0 + k1) / 2;
            recur(i0,i1,j0,j1,k0,km);
            recur(i0,i1,j0,j1,km,k1);
    }
```

[$[Get Code]]

Level 4

Now each of the 8 calls to recur() executes the base code, as we had defined a CUTOFF of 2 -

```
else {
          for (i = i0; i < i1; ++i)
              for (j = j0; j < j1; ++j)
                  for (k = k0; k < k1; ++k)
                      C[i][j] += A[i][k] * B[k][j];
    }
```

[$[Get Code]]

At this level, matrix multiplication is performed for matrices of dimensions 2 x 2. The actual computation re-uses elements from the matrix periodically, and since the size of the matrices is small, there is a lesser chance of matrix elements being evicted from cache.

More somewhat unorganized notes

Ideal Cache

- Fully Associative
- Optimal Replacement Policy
- Size Z
- Write Back Policy
- 1 word per line (not necessary, but easy to analyze)

Cache Capacity Lemma
For each sequence of instructions that access M memory locations, there will be at least M - Z misses where Z is the cache capacity.

Competitiveness of LRU

Theorem (Sleator and Tarjan ’85)
LRU running on a cache of size 2Z incurs at most twice as many cache misses as optimal replacement running on a cache of size Z.
Corollary
Cache oblivious algorithms are asymptotically optimal on an LRU cache.

Assume:
* LRU cache of size 2Z

- Optimal cache of size Z

Proof:
* Divide execution into epochs where in each epoch the program accesses 2Z distinct locations.

- LRU incurs at most 2Z misses per epoch.
- By the capacity lemma, Optimal incurs at least 2Z - Z misses per epoch.
- Thus, LRU misses no more than twice Optimal misses.

Locality of cache oblivious matrix multiplication

- Q(N) $\leq$ 8Q($\frac{N}{2}$)
    - The total number of misses in the above algorithm is less than the maximum number of misses due to data reuse. Here Q(k) represents k misses.

Even with the ideal blocking for the cache-aware algorithm, the number of misses is $\Omega\left(\frac{N^3}{\sqrt{Z}} +\right.$. So, the cache-aware and cache-oblivious achieves the same upper bound.

Here, Ω represents the upper bound.
An informal derivation of this bound goes like this:

Q(N) $\leq$ 8Q($\frac{N}{2}$)

After we recurse r times

Q(N) $\leq$ 8^rQ($\frac{N}{2^r}$)

When the problem being solved fits in the cache, additional recursion levels don’t change the locality. This happens when: $\frac{N}{2^r} \leq \sqrt{\frac{Z}{3}}$.

State another way, no additional misses occur after we recurse r=$\log_2 \sqrt{\frac{3}{Z}}N$ times. At this point the number of misses for the subproblem is less than Z (that’s how we chose r).

Now plugging back in to Q(N):

Q(N) $\leq$ 8^rZ

Q(N) $\leq$ 2^{3^r}Z

Q(N) $\leq$ $\left(\sqrt{\frac{3}{Z}}N\right)^3$ Z

And when accounting for the fact that we have at least $3N^2$ cold misses on the matrices:

Q(N) $\leq$ $\Omega\left(\frac{N^3}{\sqrt{Z}} + N^2\right)$