CMSC 180:
Introduction to Parallel Computing

Jaderick P. Pabico
Institute of Computer Science, College of Arts and Sciences
University of the Philippines Los Baños, College 4031, Laguna

2nd Semester 2020-2021

Improving Memory Latency Using Caches

- Caches are small and fast memory elements between the processor and DRAM.
- This memory acts as a low-latency, high-bandwidth storage
- If a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache

Improving Memory Latency Using Caches

- The fraction of data references satisfied by the cache is called the cache hit ratio of the computation of the system
- Cache hit ratio achieved by a cache on a memory system often determines its performance

Impact of caches

- Repeated references to the same data item correspond to temporal locality
- Lesson: Data reuse is critical for cache performance.

Impact of memory bandwidth

- Memory bandwidth is determined by the bandwidth of the memory bus and the memory units
- Memory bandwidth can be improved by increasing the size of memory blocks.
- The underlying system takes *l* time units to deliver *b* units of data
  - *l* is the latency of the system
  - *b* is the block size.

Impact of memory bandwidth

- Lesson: increasing block size does not change the latency
- In practice, wide buses are expensive
- What really is implemented? Consecutive words are sent on the memory bus on subsequent bus cycles after the first word is retrieved.

Impact of memory bandwidth

- Increased bandwidth results in higher peak computation rates
- The data layouts were assumbed to be such that consecutive data words in memory were used by successive instructions (spatial locality of reference)
- If we take a data-layout centric view, computations must be reordered to enhance spatial locality of reference.

Impact of memory bandwidth

- Consider the following code:

```
for (i = 0; i < 1000; i++)
        column_sum[i] = 0.0;
        for (j = 0; j < 1000; j++)
                column_sum[i] += b[j][i];
```

This code sums columns of the matrix `b` into a vector `column_sum`

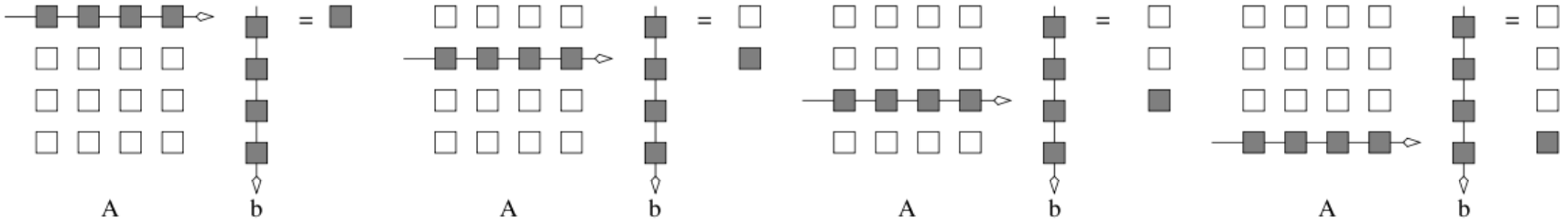
Impact of memory bandwidth

- The vector `column_sum` is small and easily fits into the cache
- The matrix `b` is accessed in a column order
- Stridded access results in very poor performance
  - Depends whether arrays are column major
  - Or row major

Impact of memory bandwidth

- Multiplying a matrix `A` with a vector `b`

Column major data access



Row major data access

Impact of memory bandwidth

- If instead we have this code:

```
for (i = 0; i < 1000; i++)
        column_sum[i] = 0.0;
for (j = 0; j < 1000; j++)
        for (i = 0; i < 1000; i++)
                column_sum[i] += b[j][i];
```

The matrix is traversed in a row-order and performance can be expected to be significantly better.

LESSONS: Memory System Performance

- Exploiting spatial and temporal locality in applications is critical for amortizing memory latency and increasing effective memory bandwidth
- The ratio of the number of operations to number of memory accesses is a good indicator of anticipated tolerance to memory bandwidth
- Memory layouts and organizing computation appropriately can make a significant impact on the spatial and temporal locality

Alternate approaches for hiding memory latency

- Consider the problem of browsing the web in a very slow network connection.
- Three possible solutions:
  - We anticipate which pages we are going to browse ahead of time and issue requests for them in advance.
  - We open multiple browsers and access different pages in each browser, thus while we are waiting for one page to laod, we could be reading others. or

Alternate approaches for hiding memory latency

- Consider the problem of browsing the web in a very slow network connection.
- Three po
  - We an going to brows uests for them i
  - We op ess differe s while we are waiting for one page to laod, we could be reading others. or

We access a whole bunch of pages in one go – amortizing the latency across various accesses.

Alternate approaches for hiding memory latency

- These approaches have names:
  - Prefetching
  - Multithreading
  - Spatial locality in accessing memory words

Multithreading

- Example:

```
for (i = 0; i < n; i++)
    c[i] = dot_product(get_row(a, i), b);
```

Each dot product is independent of the other, and thus represents a concurrent unit of execution. We can safely rewrite the above code as:

```
for (i = 0; i < n; i++)
    c[i] = create_thread(dot_product, get_row(a, i), b);
```

Multithreading

- In the code, the first instance of this function accesses a pair of vector elements and waits for them.
- In the meantime, the second instance of this function can access two other vector elements in the next cycle, and so on.
- After *l* units of time, where *l* is the latency, the first function instance gets the requested data from memory and can perform the required computation.

Multithreading

- In the next cycle, the data items for the next function instance arrive, and so on. In this way, in every clock cycle, we can perform a computation.
- The execution schedule in this example is predicated upon two assumptions:
  - The memory system is able to service multiple requests and
  - the processor is able to switch threads every cycle

Multithreading

- It also requires the program to have an explicit specification of concurrency in the form of threads