Limitations of Memory System Performance

- Memory system, and not processor speed, is often the bottleneck for many applications.
- Memory system performance is largely captured by two parameters, latency and bandwidth.
- Latency is the time from the issue of a memory request to the time the data is available at the processor.
- Bandwidth is the rate at which data can be pumped to the processor by the memory system.

Memory System Performance: Bandwidth and Latency

- It is very important to understand the difference between latency and bandwidth.
- Consider the example of a fire-hose. If the water comes out of the hose two seconds after the hydrant is turned on, the latency of the system is two seconds.
- Once the water starts flowing, if the hydrant delivers water at the rate of 5 gallons/second, the bandwidth of the system is 5 gallons/second.
- If you want immediate response from the hydrant, it is important to reduce latency.
- If you want to fight big fires, you want high bandwidth.

Memory Latency: An Example

Consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches). Assume that the processor has two multiply-add units and is capable of executing four instructions in each cycle of 1 ns. The following observations follow:

- The peak processor rating is 4 GFLOPS.
- Since the memory latency is equal to 100 cycles and block size is one word, every time a memory request is made, the processor must wait 100 cycles before it can process the data.

Memory Latency: An Example

On the above architecture, consider the problem of computing a dot-product of two vectors.

- A dot-product computation performs one multiply-add on a single pair of vector elements, i.e., each floating point operation requires one data fetch.
- It follows that the peak speed of this computation is limited to one floating point operation every 100 ns, or a speed of 10 MFLOPS, a very small fraction of the peak processor rating!

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

Impact of Caches: Example

Consider the architecture from the previous example. In this case, we introduce a cache of size 32 KB with a latency of 1 ns or one cycle. We use this setup to multiply two matrices A and B of dimensions 32×32. We have carefully chosen these numbers so that the cache is large enough to store matrices A and B, as well as the result matrix C.

Impact of Caches: Example (continued)

The following observations can be made about the problem:

- Fetching the two matrices into the cache corresponds to fetching 2K words, which takes approximately 200 μs.
- Multiplying two $n \times n$ matrices takes $2n^3$ operations. For our problem, this corresponds to 64K operations, which can be performed in 16K cycles (or 16 μs) at four instructions per cycle.
- The total time for the computation is therefore approximately the sum of time for load/store operations and the time for the computation itself, i.e., $200 + 16$ μs.
- This corresponds to a peak computation rate of $64\mathrm{K}/216$ or 303 MFLOPS.

Impact of Caches

- Repeated references to the same data item correspond to temporal locality.
- In our example, we had $O(n^2)$ data accesses and $O(n^3)$ computation. This asymptotic difference makes the above example particularly desirable for caches.
- Data reuse is critical for cache performance.

Impact of Memory Bandwidth

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

Impact of Memory Bandwidth: Example

Consider the same setup as before, except in this case, the block size is 4 words instead of 1 word. We repeat the dot-product computation in this scenario:

- Assuming that the vectors are laid out linearly in memory, eight FLOPs (four multiply-adds) can be performed in 200 cycles.
- This is because a single memory access fetches four consecutive words in the vector.
- Therefore, two accesses can fetch four elements of each of the vectors. This corresponds to a FLOP every 25 ns, for a peak speed of 40 MFLOPS.

Impact of Memory Bandwidth

- It is important to note that increasing block size does not change latency of the system.
- Physically, the scenario illustrated here can be viewed as a wide data bus (4 words or 128 bits) connected to multiple memory banks.
- In practice, such wide buses are expensive to construct.
- In a more practical system, consecutive words are sent on the memory bus on subsequent bus cycles after the first word is retrieved.

Impact of Memory Bandwidth

- The above examples clearly illustrate how increased bandwidth results in higher peak computation rates.
- The data layouts were assumed 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: Example

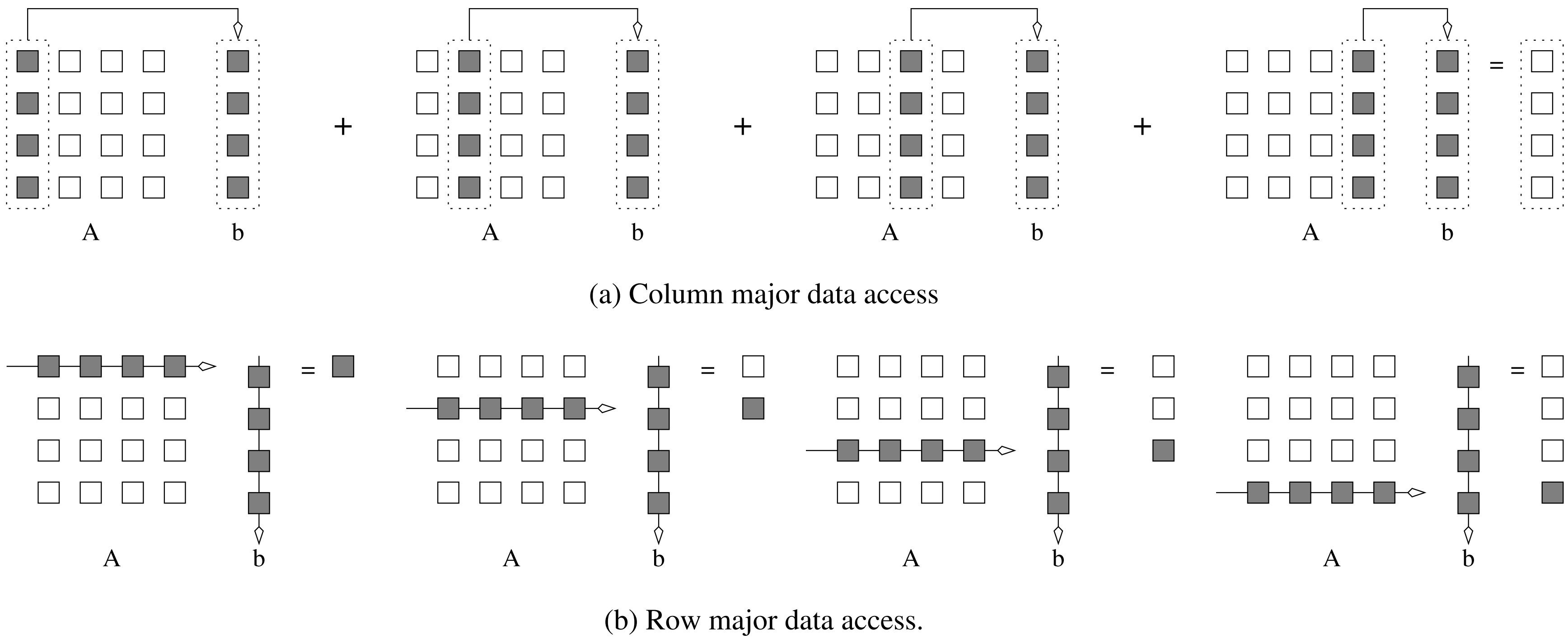
Consider the following code fragment:

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

The code fragment sums columns of the matrix `b` into a vector `column_sum`.

Impact of Memory Bandwidth: Example

- The vector `column_sum` is small and easily fits into the cache
- The matrix `b` is accessed in a column order.
- The strided access results in very poor performance.



Multiplying a matrix with a vector: (a) multiplying column-by-column, keeping a running sum; (b) computing each element of the result as a dot product of a row of the matrix with the vector.

Impact of Memory Bandwidth: Example

We can fix the above code as follows:

```
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];
```

In this case, the matrix is traversed in a row-order and performance can be expected to be significantly better.

Memory System Performance: Summary

The series of examples presented in this section illustrate the following concepts:

- 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 on a very slow network connection. We deal with the problem in one of three possible ways:

- 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 load, we could be reading others; or
- we access a whole bunch of pages in one go – amortizing the latency across various accesses.

The first approach is called *prefetching*, the second *multithreading*, and the third one corresponds to spatial locality in accessing memory words.

Multithreading for Latency Hiding

A thread is a single stream of control in the flow of a program. We illustrate threads with a simple 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 therefore represents a concurrent unit of execution. We can safely rewrite the above code segment as:

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

Multithreading for Latency Hiding: Example

- 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 of the memory system, the first function instance gets the requested data from memory and can perform the required computation.

- 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.

Multithreading for Latency Hiding

- The execution schedule in the previous example is predicated upon two assumptions: the memory system is capable of servicing multiple outstanding requests, and the processor is capable of switching threads at every cycle.

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

- Machines such as the HEP and Tera rely on multithreaded processors that can switch the context of execution in every cycle. Consequently, they are able to hide latency effectively.

Prefetching for Latency Hiding

- Misses on loads cause programs to stall.
- Why not advance the loads so that by the time the data is actually needed, it is already there!
- The only drawback is that you might need more space to store advanced loads.
- However, if the advanced loads are overwritten, we are no worse than before!