Parallel Computing Platforms

Chieh-Sen (Jason) Huang

Department of Applied Mathematics

National Sun Yat-sen University

Thank Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar for providing slides.

Topic Overview

- Implicit Parallelism: Trends in Microprocessor Architectures
- Limitations of Memory System Performance

Scope of Parallelism

- Conventional architectures coarsely comprise of a processor, memory system, and the datapath.
- Each of these components present significant performance bottlenecks.
- Parallelism addresses each of these components in significant ways.
- Different applications utilize different aspects of parallelism – e.g., data itensive applications utilize high aggregate throughput, server applications utilize high aggregate network bandwidth, and scientific applications typically utilize high processing and memory system performance.
- It is important to understand each of these performance bottlenecks.

Implicit Parallelism: Trends in Microprocessor Architectures

- Microprocessor clock speeds have posted impressive gains over the past two decades (two to three orders of magnitude).
- Higher levels of device integration have made available a large number of transistors.
- The question of how best to utilize these resources is an important one.
- Current processors use these resources in multiple functional units and execute multiple instructions in the same cycle.
- The precise manner in which these instructions are selected and executed provides impressive diversity in architectures.
- We shall not discuss any further on these topics. (Details, see Grama's slides.)

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 (2000×100ns).

- 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., $200 + 16\mu$s.

- This corresponds to a peak computation rate of 64K FLOP$/216\mu s$ or 303 MFLOPS.

Impact of Caches

- 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.

- Repeated references to the same data item correspond to temporal locality.

- Aggressive caching $\left\{ \begin{array}{l} \text{Spatial locality} \\ \text{Temperal locality} \end{array} \right.$

  1. Spatial locality : Access data in blocks.
  2. Temperal locality : Reuse data that is already loaded.

Impact of Caches

- Loop unrolling

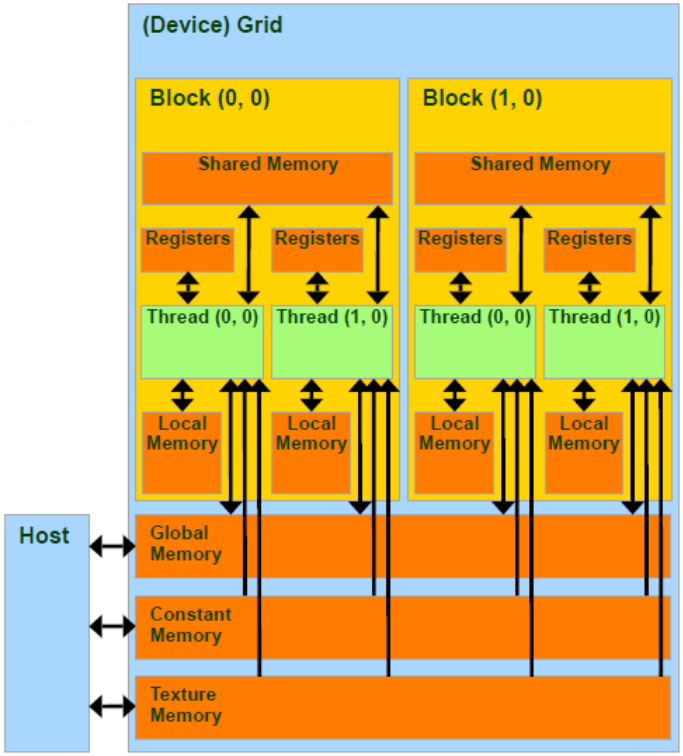
```
Do i = 1, n, 4
   sum1 = sum1 + a[i]*b[i]
   sum2 = sum2 + a[i+1]*b[i+1]
   sum3 = sum3 + a[i+2]*b[i+2]
   sum4 = sum4 + a[i+3]*b[i+3]
End do

sum = sum1 + sum2 + sum3 + sum4
```

- Blocked matrix multiplication

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} \begin{bmatrix} A_1B_1 + A_2B_3 & A_1B_2 + A_2B_4 \\ A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{bmatrix}$$

- Homework: Compute the FLOPS of the loop unrolling and blocked matrix multiplication examples.

Impact of Memory Bandwidth



The architecture of GPU.

- 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.

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

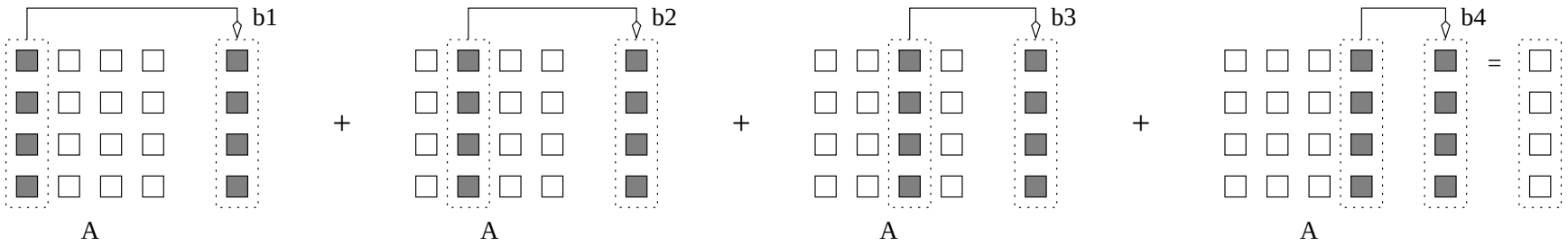
The code fragment sums columns of the matrix `b` into a vector `column_sum`.

```
  int a[2][3];
  for(int i=0;i<2;i++){
    for (int j=0;j<3;j++)
      cout<<'\t'<<&a[i][j];
    cout<<endl;
  }
```

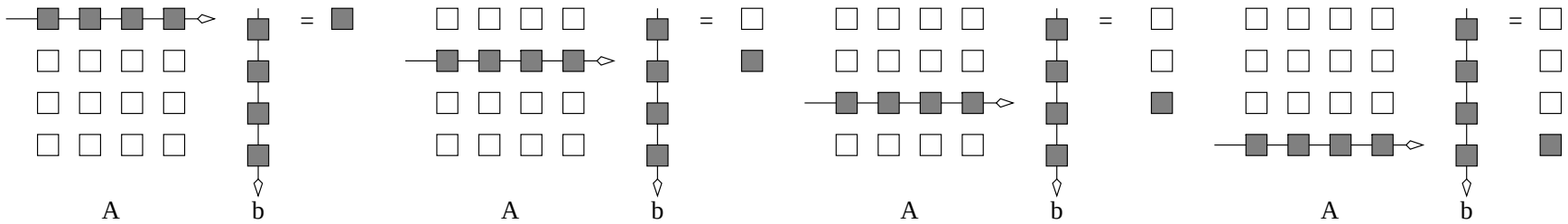
```
-----------------------------------------------------------
        0x7fff9987c700  0x7fff9987c704  0x7fff9987c708
        0x7fff9987c70c  0x7fff9987c710  0x7fff9987c714
```

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.



(a) Column major data access



(b) Row major data access.

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.