Computer Architecture and Performance: Virtual Memory

William Gropp

Virtual Memory

- So far, we've assumed that the process is addressing "memory"
- In most systems, (user) processes use "virtual" addresses
  - ♦ Gives the process the illusion that it directly addresses all real memory
  - ♦ Gives the process the illusion that there is more real memory than is really available

How Virtual Memory Works

- Memory is divided into blocks called *pages*
  - Each address has two parts
    - Low bits: location of item within a page
    - High bits: page number
  - Pages are mapped to different parts of the real memory *or* stored on denser (but slower) media (typically disk)

Paging Example

High bits in address | Low bits

Page Table

Memory

Memory page

. . .

All of memory is divided into pages

A page table entry is required for each memory page

Low bits in address give location within page

Implementing Paging

- Virtual memory introduces some costs because the virtual address must be translated to a physical address
- Consider this case:
  - ♦ Let each page contain 4k bytes
    - A common size
  - ♦ Address uses lower 12 bits to represent location in the page
  - ♦ Upper bits give page number
    - For a 32-bit address space (4GB of memory), use the top 20 bits
- For each page number, there is a corresponding location
  - ♦ Either in physical (real) memory
  - ♦ On “backing store” (in the *swap* file on disk)

Page Tables

- Each page number requires a *mapping* to real storage
  - ♦ If all 4GB memory is real, require 2^{20} entries of 4 bytes each, or 2^{22} bytes of memory
    - This is 4MB
    - As large or larger than many cache memories
      - Using a single table to map all of virtual memory to real memory will be slow (memory lookup is latency bound)
    - A common solution is to use a cache
      - Every address used by the program must be translated from a virtual address to a physical address
      - Must be *very* fast
      - Typically small (e.g., 64 pages (entries))

Paging Example With Cache

Translation Lookaside Buffer (TLB)

- The page mapping cache is called a Translation Lookaside Buffer (TLB)
  - ♦ Lookup is not easy when it has to be very fast
  - ♦ As a result, TLBs are often small but fast enough to return physical address quickly
- What happens on a page miss (entry is not in the TLB)?
  - ♦ Fetch entry from memory (the whole page table isn't big relative to main (DRAM) memory
    - Main memory latency cost

When Virtual Memory Exceeds Physical Memory

- Virtual memory allows the memory size (apparently) available to a process to exceed the available physical memory
  - ♦ If multiple processes are sharing memory, virtual memory allows each process to act as if it has all of the memory
- Where is the data actually stored (it has to be somewhere)?
  - ♦ Typically stored on disk in a *swapfile* (also called *secondary storage*). May be stored in other slow but high-density memory
- The page table indicates whether the location of the memory is in physical (fast) memory or if it is in secondary storage

TLB Revisited

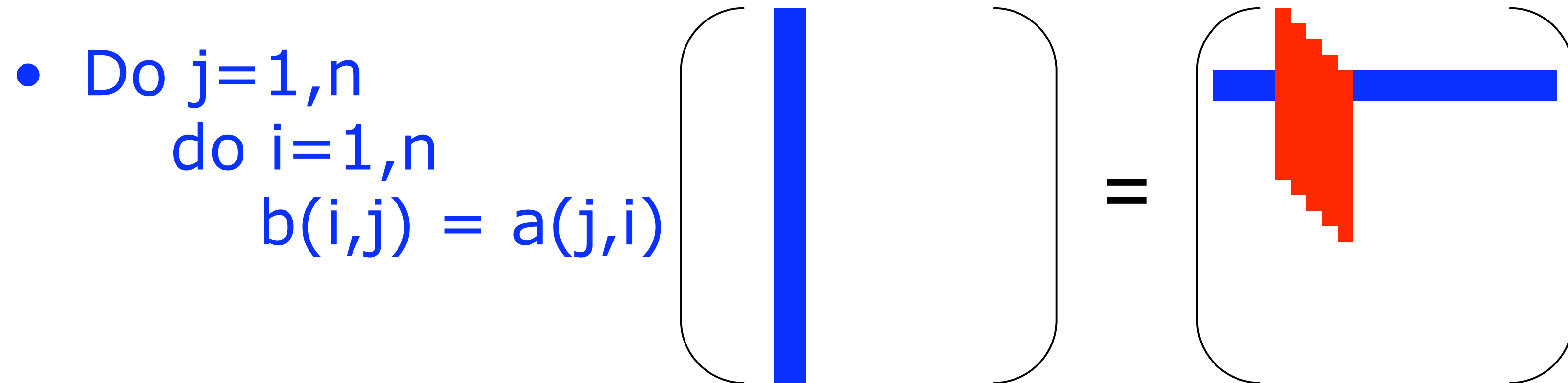
- When an page location is not found in the TLB, first find the entry in the page table
  - ♦ Requires a memory read - latencies of 20 to 100s of cycles.
- Determine if the page is stored in the main memory (resident) or has been moved to slower disk storage
  - ♦ If resident, replace a TLB entry with the location of this page and return the physical address
  - ♦ If not resident, transfer control to the operating system to handle a page fault
    - A page fault has latencies in milliseconds (time to find and read data from disk)

Impact on Algorithms

- Large cost if data outside of TLB set is accessed frequently
- Consider the transpose example with a 2048 x 2048 matrix and a TLB with 64 entries
- Each entry an 8-byte double precision value

Simple Example: Matrix Transpose (From TLB Viewpoint)

- Do j=1,n
  do i=1,n
  b(i,j) = a(j,i)



- No temporal locality (data used once)
- Spatial locality only if (words/~~cacheline~~ Pagesize) * n fits in ~~cache~~ TLB
  - Otherwise, each column of a may cause a TLB miss

Transpose with 4K pages:

- Each column of the matrix requires 4 pages
  - A page is mapped for stores every 512 rows
  - A page is mapped for loads on every column:
    - Use only a single entry from a page before going to the next one
    - Process 2k-1 pages before returning to a previous page
    - *Every* load incurs a TLB miss

Transpose with 64k pages

- 4 columns per page
  - ♦ It takes 512 pages to cover one row of the matrix
  - ♦ But get 4 values out of each page
    - Every fourth load incurs a TLB miss

Transpose with 1MB pages

- 64 columns per page
  - It takes 32 pages to cover a row
  - Only compulsory TLB misses
    - Compulsory misses are the ones that cannot be avoided - they occur the first time that a page address is used

Observations

- Note that the TLB and the L1/L2/L3 cache have different behavior
  - For example, consider 512 separate cache lines of 128 bytes each
  - Only 64K bytes of storage
  - But if they are in 512 different pages, each reference may incure a TLB miss, even though data fits within cache!
- If a page is located in secondary media, performance may be orders of magnitude lower
  - Drop in performance is severe and sudden
- Large pages can give modest (several loads satisfied from each page) or large improvements in performance (no extra TLB misses)

Discussion Questions

- Architecture Issues
  - ♦ TLB is often very small
  - ♦ Even regular accesses (as in the strided accesses in transpose) can cause problems
    - Can hardware effectively predict pages and preload a guess at the next TLB entry?
    - Can alternative approaches be used?
      - If there was more or different information from the program, would other architectural solutions be practical?
- Programming Model Issues
  - ♦ Optimizing the transpose code appears simple
    - Blocking for cache and TLB is straightforward
    - Why don't compilers (usually) generate good code for this case?

Out of Core Algorithms

- Some algorithms have been designed to work well when the problem size exceeds available (fast) memory
- Many of these dated from before the invention of virtual memory or from the days when 1 MB of memory was a lot of memory
- These were called “out of core” algorithms (main memory was once made from magnetic cores)
- An important feature of these algorithms is that they *explicitly* managed the motion of data from slow storage to fast (working) storage

Double Buffering and Asynchronous I/O

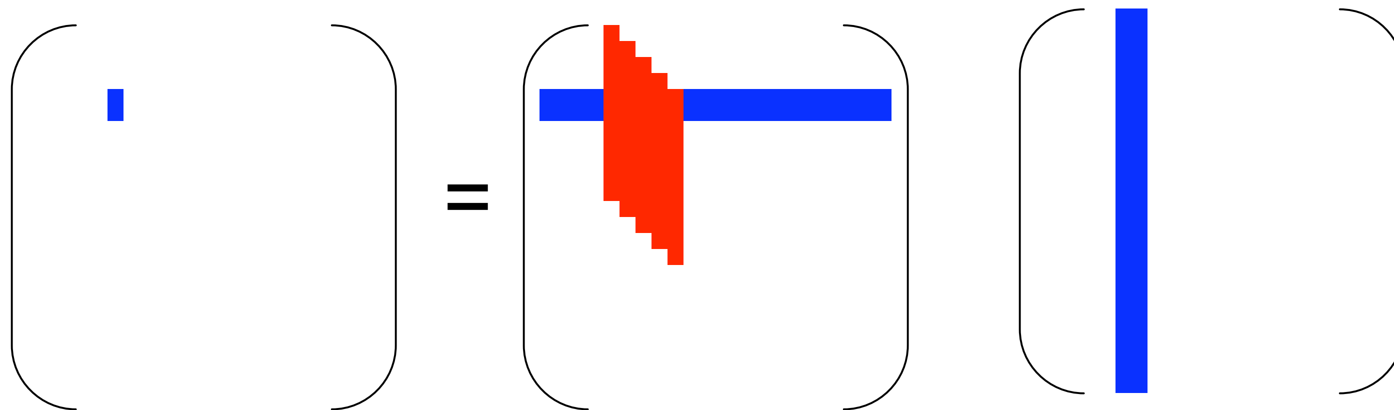
- Out of core algorithms replied on double buffering. Pseudo code looks like this:
- Load A with data
  Initiated nonblocking load of B with data to be used later
  ```
  while (not done) {
      work on data in A
      initiate nonblocking load of A with data to be used later
      wait for load of B to complete
      swap pointers to A and B
  }
  ```
- These *algorithms* can address problems with TLB misses, even to secondary storage
- But they are hard to implement in practice. Why?

Challenges in Implementing Out of Core Algorithms

- Most programming models provide no support for asynchronous operations
  - It is nearly impossible to robustly use nonblocking operations in Fortran because of the language design
    - Compiler may "optimize" around calls to library routines that implement nonblocking or asynchronous operations
- A key part of the algorithm is performing work while the "other" buffer is filled with data
  - How much work?
  - Does the work (computation) overlap (take place at the same time) with filling the buffer (communication)?
    - Programming models and hardware may *support* the operation without making it *efficient*

Another Example: Matrix-Matrix Multiply (ddot form)

- ```
  do i=1,n
      do j=1,n
          do k=1,n
              c(i,j) = c(i,j) + a(i,k) * b(k,j)
  ```



- Like transpose, but a new feature:
  - Reuse of data: n^2 data used for n^3 operations

  ```

# Memory Locality for Matrix-Matrix Multiply

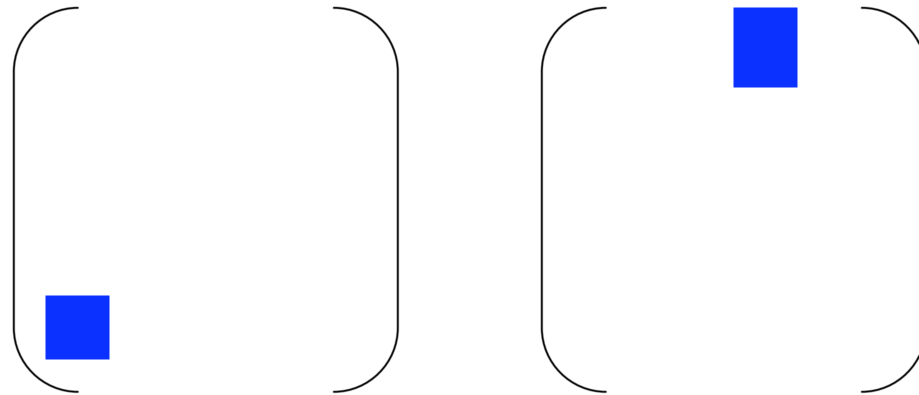
- Problems:
  - Only one value in register reused (C(i,j))
  - If cache line size * Acols > cache size, there is a miss on every load of A
  - Every cache line size (in doubles) incurs a long delay as each cacheline is loaded
- How problems are addressed
  - Can reuse values in C, A, and B
  - If cache line size * Acols > cache size, there is a miss on every load of A
  - Every cache line size (in doubles), incurs a long delay as each cacheline is loaded

# Reusing Data

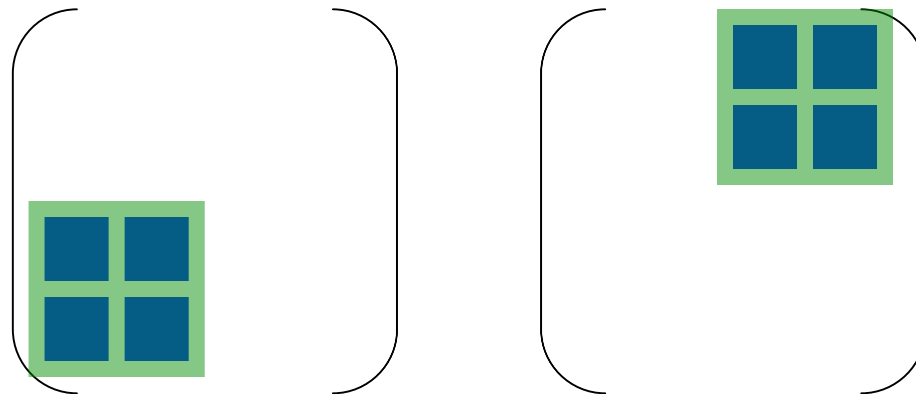
- Load data into register
- Use several times (each load, even from cache, is at least a cycle)
- Use *loop unrolling* to expose register use
  - …
    ```
    c(i,j)   += a(i,k)   * b(k,j)
    c(i+1,j)   += a(i+1,k) * b(k,j)
    c(i,j+1)   += a(i,k)   * b(k,j+1)
    c(i+1,j+1) += a(i+1,k) * b(k,j+1)
    ```
- Each a(i,j) etc. used twice
  - Cuts the numbers of loads in half
  - Requires enough registers to hold all items

# Blocking for Cache

- Reuse data in cache by blocking



Block for each level of memory hierarchy



# Blocked, Unrolled MxM (one level only)

- Do kk=1,n,stride

```
Do kk=1,n,stride
    do ii=1,n,stride
        do j=1,n-2,2
            do i=ii,min(n,ii+stride-1),2
                do k=kk,min(n,kk+stride-1)
                    c(i,j)         += a(i,k)    * b(k,j)
                    c(i+1,j)       += a(i+1,k)* b(k,j)
                    c(i,j+1)       += a(i,k)    * b(k,j+1)
                    c(i+1,j+1) += a(i+1,k)* b(k,j+1)
```

Considerations for Blocking

- Block for Registers
  - ♦ Be careful not to exceed the number of available floating point registers
- Block for load-store/floating point ratio
  - ♦ Loop over cache blocks
  - ♦ Choose size to allow load latency to be hidden by floating point work
- Block for cache size
- Block for cache bandwidth
  - ♦ To match time to move data between memory/cache to the time spent operating on data within the cache

Why Don't Compilers Perform These Transformations?

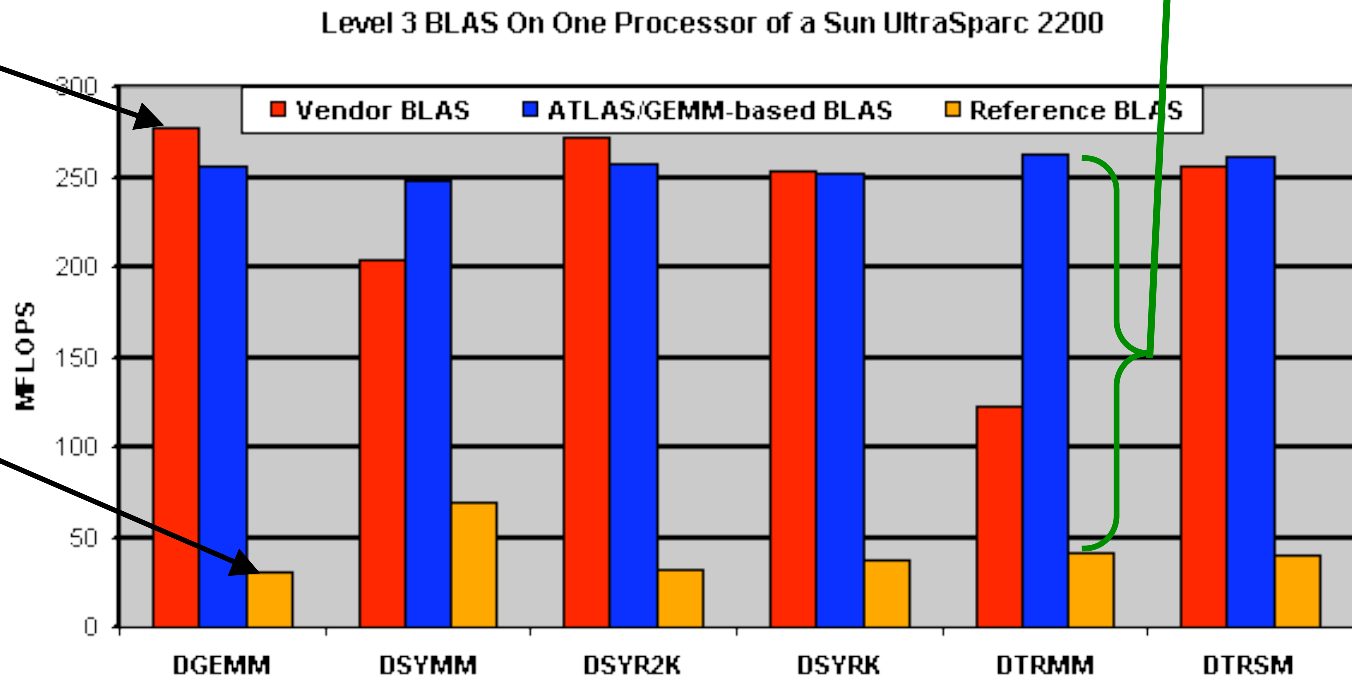
- Dense Matrix-Matrix Product
  - Most studied numerical program by compiler writers
  - Core of some important applications
  - More importantly, the core operation in High Performance Linpack
    - Benchmark used to "rate" the top 500 fastest systems
  - Should give optimal performance…
- But
  - Blocking changes the order of evaluation; floating point arithmetic is not associative
    - Thus it is wrong for the compiler to perform blocking transformations
  - While loop unrolling safe for most matrix sizes, blocking is appropriate only for large matrices (e.g., don't block for cache for 4x4 or 16x16 matrices).
    - If the matrices are smaller, the blocked code can be *slower*
- The result is a gap between performance realized by compiled code and the achievable performance

Performance Gap in Compiled Code

Large gap between natural code and specialized code

Hand-tuned

Compiler



From Atlas

Enormous effort required to get good performance

Comments

- Memory motion dominates the performance of many operations
- Sustained memory bandwidth can provide a better guide to performance
- But hardware architecture introduces features important for performance that are not visible in the programming language
  - A good thing most of the time
  - Not a good thing when performance is important

Today’s Seminar

- 3:00 p.m. 2405 Siebel Center
- Title: Memory Models: The Case for Rethinking Parallel Languages and Hardware
- Professor Sarita Adve
- Abstract: The memory model defines the legal values that a load can return, and forms the heart of the concurrency semantics of any (shared-memory) parallel language or hardware. It has typically involved a tradeoff between programmability and performance, and has arguably been one of the most challenging and contentious areas in shared-memory specification. The last few years have finally seen a convergence in this debate. Thanks to broad community-scale efforts, popular languages such as Java and C++ and most hardware vendors have now (almost) published compatible memory model specifications. Although this convergence is a dramatic improvement, it has exposed fundamental shortcomings in current popular languages and systems. …