Lecture 11: Matrix-Matrix Multiply

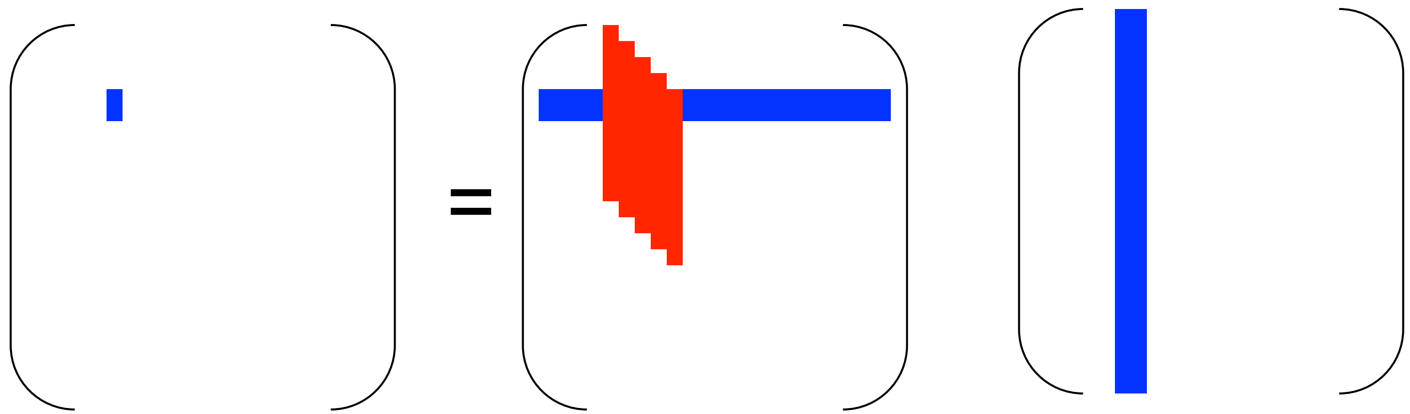
William Gropp
www.cs.illinois.edu/~wgropp

Performance for a Common Calculation

- Combine memory issues with computations
  - ♦ Spatial and Temporal locality
  - ♦ Dependencies on computation
- Dense matrix-matrix multiply a good example
  - ♦ Lots of potential to avoid extra memory operations
  - ♦ Lots of potential to arrange computation for better performance

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 two new features:
  - Perform a calculation (we'll see why this is important later)
  - Reuse of data: n^2 data used for n^3 operations

Memory Locality for Matrix-Matrix Multiply

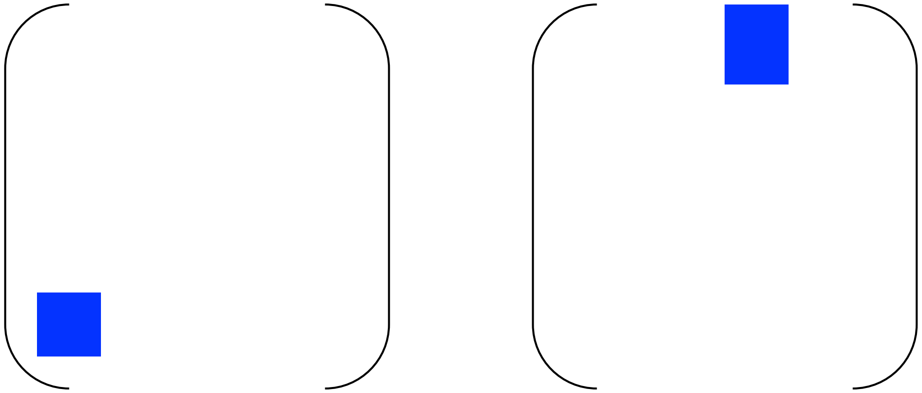
- Problems:
  - Only one value in register reused (C(i,j))
  - If cache line size * n > L1 cache size, there is a miss on every load of A
  - Every cache line size (in doubles) may incurs a long delay as each cacheline is loaded
- How problems are addressed
  - Can reuse values in C, A, and B
  - Can block matrix A
  - May be able to *prefetch* (more later)

Reusing Data

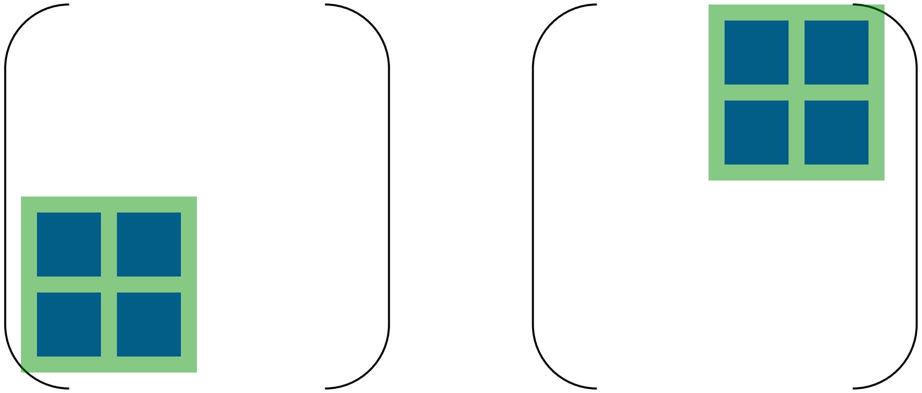
- Load data into register
- Use several times (each load, even from cache, is at least a cycle)
- Use *loop unrolling* to expose register use
  - …
    ```
    c(i,j)     += a(i,k)   * b(k,j)
    c(i+1,j)   += a(i+1,k) * b(k,j)
    c(i,j+1)   += a(i,k)   * b(k,j+1)
    c(i+1,j+1) += a(i+1,k) * b(k,j+1)
    ```
- Each a(i,j) etc. used twice
  - Cuts the numbers of loads in half
  - But requires enough registers to hold all items
    - 4 registers for a(I,k), a(I+1,k), b(k,j), b(k,j+1) plus 2 registers for I, j, and 4 registers for address of a(I,k), address of b(k,j), address of c(I,j), and address of c(I,j +1).

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 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)
  ```
- This is only a first step. Achieving good performance for this simple operation requires blocking for each level of cache, available registers, (and TLB – for huge problems).
  ```

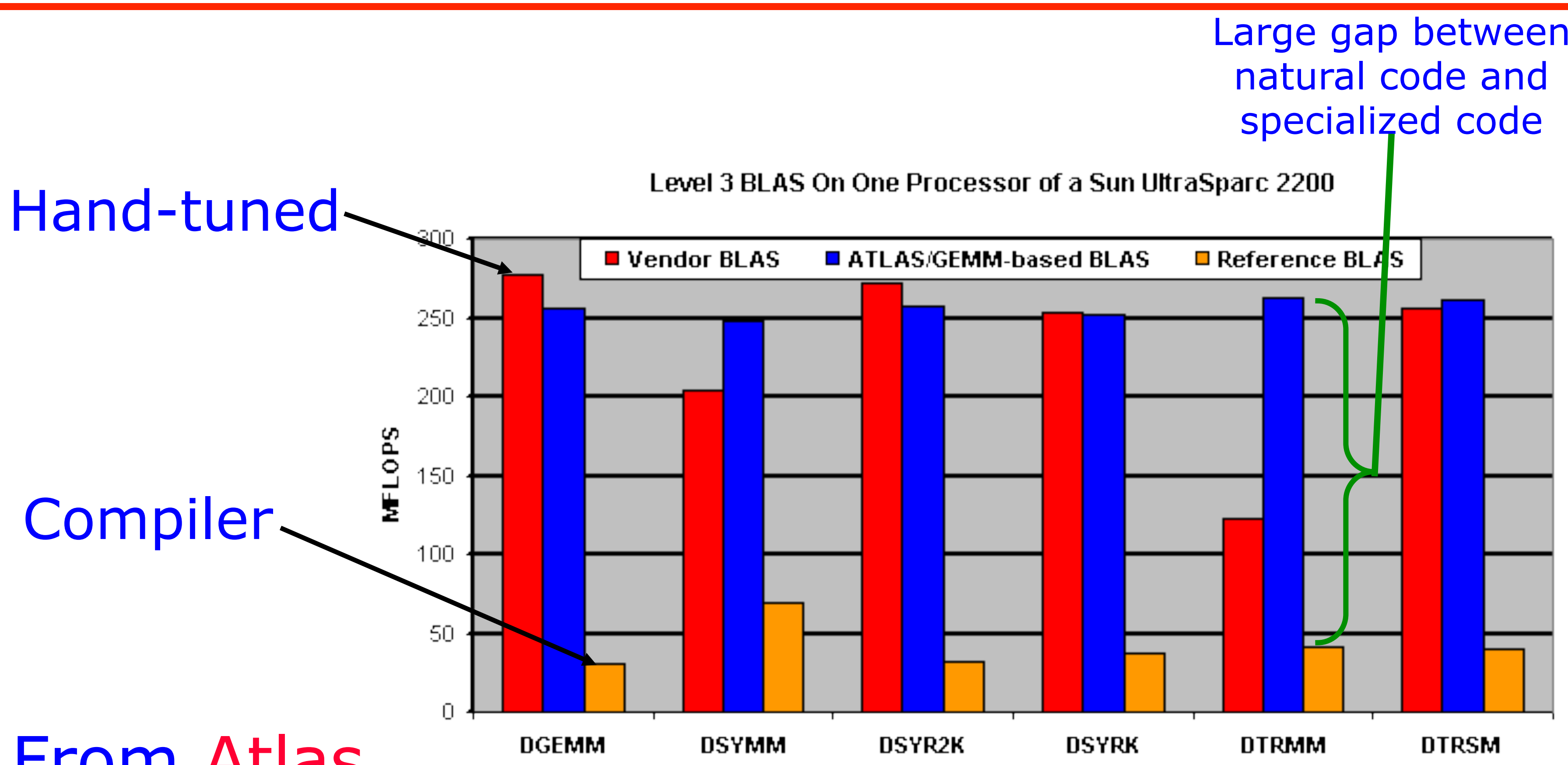
Considerations for Blocking

- 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 - we'll see this later)
- 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



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

Comments

- Very high quality compilers can perform many of these transformations
  - Note that some are *not exact* for floating point arithmetic
  - High levels of optimization may assume floating point arithmetic is associative
- Some even detect matrix-matrix multiply
  - Performance for similar-looking operations may not be as good

Matrix-Matrix Multiply Performance

- There are many things to take into account in creating a fast matrix-matrix multiply routine
  - We've just touched on a few to illustrate performance issues and models
  - You can find more information, including tutorials, focused on this and similar dense matrix operations