

Computer Architecture and Performance: Memory Impact; Instruction Execution

William Gropp



Importance of Memory in Performance Bounds

- We have seen:
 - ◆ Loads and stores can be as important as floating point operations
 - ◆ Simple models that look at just sustained memory bandwidth (and ignore details of cache effects) can provide useful *bounds* on performance
 - Recall the sparse matrix-multiply example
 - True for problems where the majority of data accesses are consecutive
 - ◆ Note that this is a bound, a guaranteed-not-to-exceed value for the performance



2

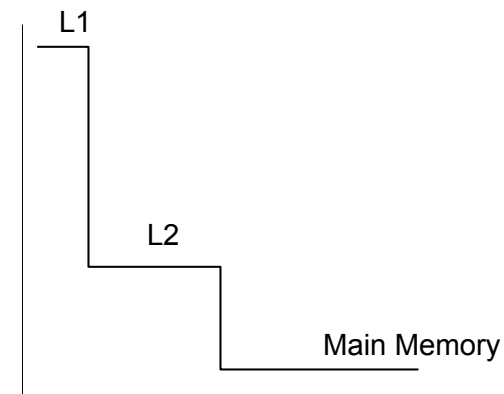
Refining the Bounds

- Achievable memory bandwidth depends on small, fast, cache memory
 - ◆ *Temporal locality* reduces need for sustained memory bandwidth
 - Data is present in faster memory
 - ◆ Leads to an important rule-of-thumb in performance modeling
 - Identify dominant terms
 - Architectures are too complex for any other approach
 - Similar to approaches for mathematical modeling
 - ◆ “Does it fit in cache” provides a first model
 - Assume all of cache is available
 - For multiple cache levels, use performance of the smallest (fastest) cache in which the data fits



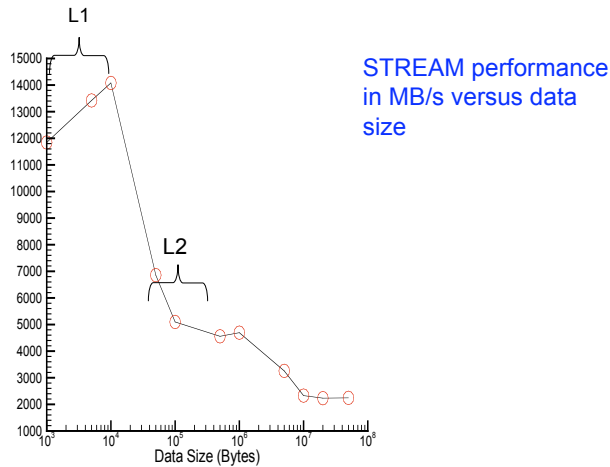
3

Memory Bandwidth vs Data Size



4

Impact of Memory Hierarchy



5

Refining the Bounds: Spatial Locality

- Non-consecutive memory accesses expose more details about the memory structure
 - ◆ Effective cache size reduces when not all of the data on the same cache line is used
 - *Spatial Locality* important to get full use of cache
 - Cache line size helps in refining performance bounds
 - Reduce effective cache size to same fraction of cache line used.
 - E.g., if 1 8 byte value used from a 128-byte cache line, or 1/16 of the line, the effective cache size is 1/16 of the full size.



6

Breaking the Model: TLB

- Adding virtual memory requires an extremely fast way to convert virtual addresses to physical addresses
 - ◆ The Translation Lookaside Buffer is a cache that performs this translation
 - ◆ However, with typical page sizes, the TLB does not provide fast translation for all memory in cache
 - Cost of occasional TLB miss in consecutive accesses (for data in memory and not on disk) is relatively small
 - Cost for non-consecutive addresses can be very large
 - ◆ Partial fix: Specify larger pages
 - No standard way to do this in language (or among flavors of Unix)
 - ◆ Algorithmic fix: Change order of accesses
 - No standard way to control in language
 - Depends on page and cache line size



7

What's Next?

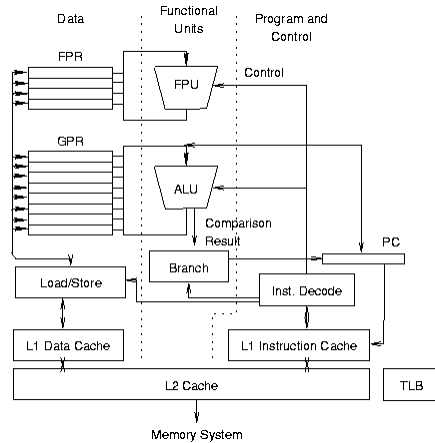
- There are many details that we've ignored
 - ◆ Can more than one operation take place at a time?
 - ◆ Does each assignment require a store into memory?
 - ◆ What about the other operations (loop counts and tests, array indexing, etc.)?
- Before answering these, let's revisit the CPU



8

Basic Processor Architecture

- A representative processor architecture
- Key points:
 - ◆ Multilayered memory system
 - ◆ Multiple functional units permit concurrent actions (e.g., loads and floating point operations)
 - ◆ Some operations (e.g., address translation) have hardware assist (TLB) but may fall back on software



9

More Details

- Can more than one operation take place at a time?
 - ◆ Yes, if they involve different functional units
 - ◆ Or if there are multiple units of the same type, as long as enough units are available
 - Architecture Feature: Quickest way to add to peak floating point performance is to add floating point units
 - Algorithm and Programming language must make use of these
 - Discussion Question: Are there natural ways to use and express this?



10

More Details (2)

- Does each assignment require a store into memory?
- Consider this code in C:


```
double sum = 0;
for (I=0; I < n; I++) {
    sum = sum + a[I];
}
```
- The value "sum" may be stored in register, requiring no load or store.
 - ◆ Making use of registers can be crucial in achieving high performance
 - ◆ Recall the CPU diagram: most operations take place between operands in register



11

More Details (3): Traps for the Unwary

- Consider these two codes in C:
 - ◆

```
void sum( double *total, double *a, int n ) {
    int I;
    for (I=0; I<n; I++) *total += a[I];
}
and
void sum2( double *total, double *a, int n ) {
    register double s = *total; int I;
    for (I=0; I<n; I++) s += a[I];
    *total = s;
}
```
 - ◆ Do these codes compute the same result?
 - The second version explicitly stores the partial result into a location that the programming language specifies may be in register



12

Perils of Aliasing

- They do not compute the same value!
- Consider this usage of the routines
 - ◆ `Sum(&a[2], a, 3);`
 - ◆ In the first case, the routine computes
 - $A[2] + A[0] + a[1] + a[2] + a[0] + a[1]$
 - Why?
 - ◆ In the second case, the routine computes
 - $A[0] + a[1] + a[2]$
- When two variables may describe overlapping memory regions, they are said to *alias* one another
 - ◆ Programming languages with pointers often permit aliasing (how can they prevent it)
 - ◆ The *potential* for aliasing can *force* the compiler to store a value (or in a different example, load it) even though the programmer does not intend to use aliased data
 - ◆ Discussion Question: Is this a flaw in the programming model? If so, how would you fix it?



13

More Details (4)

- What about the other operations (loop counts and tests, array indexing, etc.)?
 - ◆ Operations on integers are relatively fast in modern CPUs
 - Exceptions include integer divide and modulus
 - ◆ Branches (conditional jumps to other parts of the code, such as at a loop test) are also relatively expensive
 - ◆ However, most are still faster than an L2 cache miss



14

Can those Operations be Ignored in Performance Bounds?

- Not a priori - you should check
- To test whether they can be ignored, compute a *bound* on them:
 - ◆ Assume simple operations: add, integer multiply, compare, branch, etc.
 - ◆ Assume one cycle each
 - A very coarse assumption
 - ◆ Assume these can execute concurrently with loads, stores, and floating point operations
 - Remember the CPU diagram - each functional unit can run independently
 - ◆ In numerical calculations, it is *often* the case that the *sustained* load/store rate is the limiting step
 - But more computationally intensive calculations with complex control may be dominated by these "other" operations



15

Some Rules for Bounding Performance

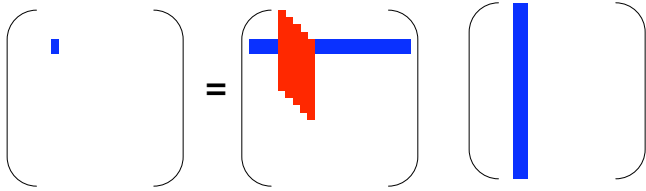
- Most importantly remember: the goal is to create an effective (but possibly approximate) bound on performance - *not* an estimate!
 - ◆ Discussion Question: What's the difference?
- Count the number of operations in each *functional unit category*:
 - ◆ Loads/Stores
 - ◆ Floating Point (add, subtract, multiply - divides are a special subcase)
 - ◆ Other operations (integer arithmetic, branches, comparisons, etc.)
- For each of these, compute the time they will take
- The bound on the time is the max of these three
 - ◆ Note: not really a bound because we've ignored any dependencies between the different operations
 - ◆ You can refine each of these by including more detail
 - Refine load/store by considering cache



16

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



17

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) 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 * $n >$ L1 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



18

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)
    
```

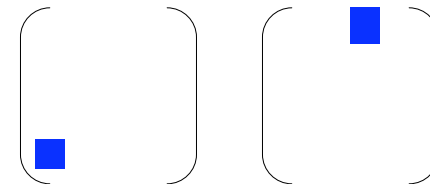
- Each $a(i,j)$ etc. used twice
 - Cuts the numbers of loads in half
 - But requires enough registers to hold all items
 - 4 registers for $a(I,k)$, $a(I+1,k)$, $b(k,j)$, $b(k,j+1)$ plus 2 registers for I , j , and 4 registers for address of $a(I,k)$, address of $b(k,j)$, address of $c(I,j)$, and address of $c(I,j+1)$.



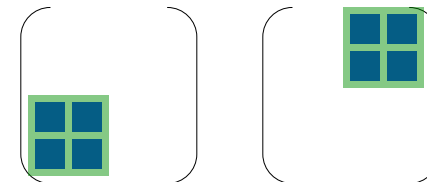
19

Blocking for Cache

- Reuse data in cache by blocking



Block for each level of memory hierarchy



20

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.



21

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



22

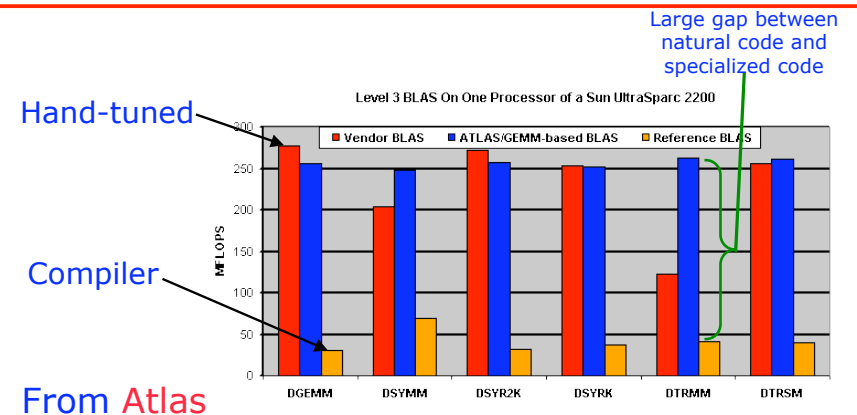
Why Don't Compilers Perform These Transformations?

- 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



23

Performance Gap in Compiled Code



Enormous effort required to get good performance

24

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



25

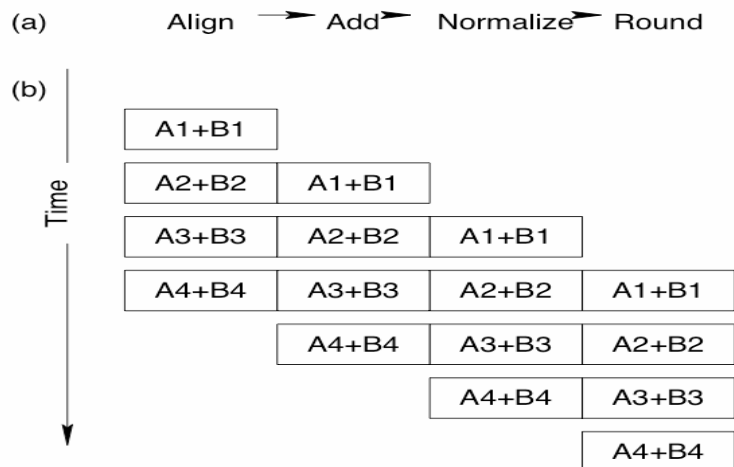
Yet More To Consider in Understanding Performance

- We have implicitly assumed that an operation takes one clock cycle.
 - ◆ This is rarely true!
- So why can we assume that instructions take one cycle?
 - ◆ Because most operations can be *started* every cycle
 - Each operation is divided into several steps
 - A different step is performed for each operation during a clock cycle
 - This approach is called *pipelining*
 - Not all instructions can be pipelined!
 - ◆ Impact on algorithms and programming models
 - Full performance requires multiple, concurrent (and independent) operations



26

Example: Floating Point Addition



27

Memory Bus Speeds Versus Sustained Memory Bandwidth

- The performance measured by the STREAM benchmark is different (and lower) than the "memory bus bandwidth". Why?
 - ◆ Memory bus bandwidth is simply the width of the memory bus (in bytes) times the clock rate of the bus
 - Instantaneous rate at which data can be transferred
 - ◆ Lets look at the STREAM code and see what it does



28

Understanding STREAM Performance

- Consider the simple case of memory copy:
 - Do $i=1, n$
 $a(i) = b(i)$
 - Suppose system memory bandwidth is 5.5GB/s. How fast will this loop execute?



29

Stream Performance Estimate

- Easy estimate: 11 GB/s = 2 * 5.5 GB/Sec to L3, 5.5 GB/Sec to main memory
 - Minimum link speed is 5.5 GB/s each way, Stream adds both
- Measured performance is 1.2 GB/s!
 - Why?
- Time to move each cache line
 - 5.5 GB/s ~ 8 bytes/cycle (memory bus bandwidth)
 - ~60 cycles L2 miss (latency)
 - 64 byte cache line = 8 cycles (bandwidth) + 60 cycles (latency) = 68 cycles or ~ 1byte/cycle (read)
 - Stream bytes read + bytes written / time, so stream estimate is 2 * 1 byte/cycle, or 1.4 GB/sec
- This is typical (if anything, better than many systems because L2 miss cost is low)



31

BG/L Node

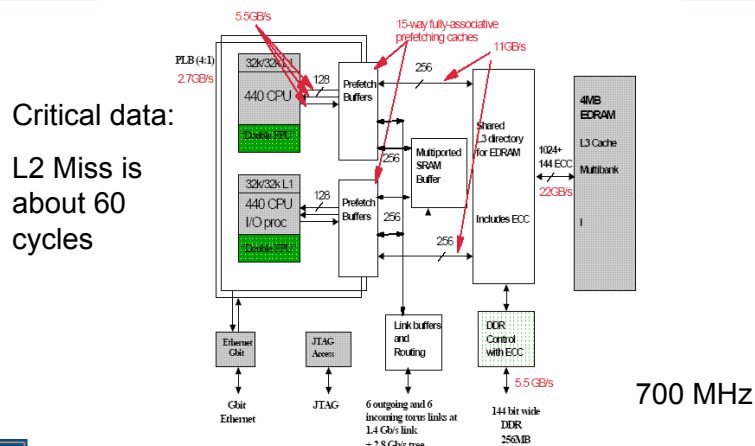


Figure 4: BlueGene/L Node Diagram. The bandwidths listed are targets.



30

Impact of Instruction Latency

- Programming languages usually present a model in which one line (or operation) completes before the next starts
 - This is not what happens in pipelined architectures (= real world)
- Algorithms often have the same feature
 - After all, often written as pseudo code with these features
- Discussion Questions:
 - Does this impact the algorithms that are considered?
 - Would a graphical description encourage more exposure of overlapping operations
 - As in the pipelined sum
 - Alternatively, should the input format (written by the programmer) and the output format (developed by the compiler) be different to reflect the existence of pipelining
 - Some vectorizing compilers took small steps in this direction



32