CS4230 Parallel Programming

Lecture 8: Dense Linear Algebra and Locality Optimizations

Mary Hall
September 13, 2012

Administrative

- I will be on travel again Tuesday, September 18
- Axel will provide the algorithm description for Singular Value Decomposition, which is our next programming assignment

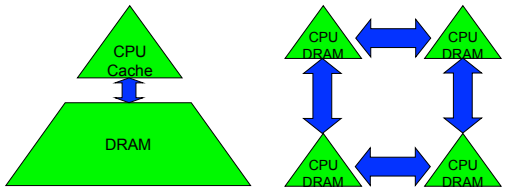
Today's Lecture

- Dense Linear Algebra, from video
- Locality
  - Data reuse vs. data locality
  - Reordering transformations for locality
- Sources for this lecture:
  - Notes on website

Back to basics: Why avoiding communication is important (1/2)

Algorithms have two costs:

1. Arithmetic (FLOPS)
2. Communication: moving data between
   - levels of a memory hierarchy (sequential case)
   - processors over a network (parallel case).



Slide source: Jim Demmel, CS267 CS267 Lecture 11 4

Why avoiding communication is important (2/2)

- Running time of an algorithm is sum of 3 terms:
  - # flops * time_per_flop
  - # words moved / bandwidth
  - # messages * latency

  } communication

- Time_per_flop << 1/ bandwidth << latency
  - Gaps growing exponentially with time

Annual improvements			
Time_per_flop		Bandwidth	Latency
59%	Network	26%	15%
	DRAM	23%	5%

- Goal : organize linear algebra to *avoid* communication
  - Between all memory hierarchy levels
    - L1 ↔ L2 ↔ DRAM ↔ network, etc
  - Not just *hiding* communication (overlap with arith) (speedup ≤ 2x)
  - Arbitrary speedups possible

Slide source: Jim Demmel, CS267

THE UNIVERSITY OF UTAH

Review: Naïve Sequential MatMul: C = C + A*B

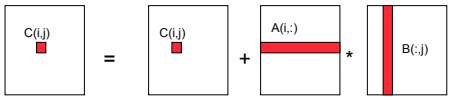
```
for i = 1 to n

  for i = 1 to n

    {read column j of B into fast memory, n³ reads}
    for k = 1 to n

    {write C(i,j) back to slow memory, n² writes}
```

$n^3 + O(n^2)$ reads/writes altogether



Slide source: Jim Demmel, CS267 CS267 Lecture 11

THE UNIVERSITY OF UTAH

Less Communication with Blocked Matrix Multiply

- Blocked Matmul C = A·B explicitly refers to subblocks of A, B and C of dimensions that depend on cache size

… Break A^{nxn}, B^{nxn}, C^{nxn} into bxb blocks labeled A(i,j), etc
… b chosen so 3 bxb blocks fit in cache
for i = 1 to n/b, for j=1 to n/b, for k=1 to n/b
C(i,j) = C(i,j) + A(i,k)·B(k,j) … b x b matmul, $4b^2$ reads/writes

- $(n/b)^3 \cdot 4b^2 = 4n^3/b$ reads/writes altogether
- Minimized when $3b^2$ = cache size = M, yielding $O(n^3/M^{1/2})$ reads/writes

- What if we had more levels of memory? (L1, L2, cache etc)?
  - Would need 3 more nested loops per level

Slide source: Jim Demmel, CS267 CS267 Lecture 11

THE UNIVERSITY OF UTAH

Blocked vs Cache-Oblivious Algorithms

- Blocked Matmul C = A·B explicitly refers to subblocks of A, B and C of dimensions that depend on cache size

… Break A^{nxn}, B^{nxn}, C^{nxn} into bxb blocks labeled A(i,j), etc
… b chosen so 3 bxb blocks fit in cache
for i = 1 to n/b, for j=1 to n/b, for k=1 to n/b
C(i,j) = C(i,j) + A(i,k)·B(k,j) … b x b matmul
… another level of memory would need 3 more loops

- Cache-oblivious Matmul C = A·B is independent of cache

Function C = RMM(A,B) … R for recursive
If A and B are 1x1
C = A · B
else … Break A^{nxn}, B^{nxn}, C^{nxn} into (n/2)x(n/2) blocks labeled A(i,j), etc
for i = 1 to 2, for j = 1 to 2, for k = 1 to 2
C(i,j) = C(i,j) + RMM(A(i,k), B(k,j)) … n/2 x n/2 matmul

Slide source: Jim Demmel, CS267 CS267 Lecture 11

THE UNIVERSITY OF UTAH

Communication Lower Bounds: Prior Work on Matmul

- Assume n^3 algorithm (i.e. not Strassen-like)
- Sequential case, with fast memory of size M
  - Lower bound on #words moved to/from slow memory = $\Omega(n^3 / M^{1/2})$ [Hong, Kung, 81]
  - Attained using blocked or cache-oblivious algorithms
- Parallel case on P processors:
  - Let M be memory per processor; assume load balanced
  - Lower bound on #words moved = $\Omega(n^3 / (p \cdot M^{1/2}))$ [Irony, Tiskin, Toledo, 04]
  - If M = $3n^2/p$ (one copy of each matrix), then lower bound = $\Omega(n^2 / p^{1/2})$
  - Attained by SUMMA, Cannon's algorithm

New lower bound for all "direct" linear algebra

Let M = "fast" memory size per processor
= cache size (sequential case) or $O(n^2/p)$ (parallel case)
#flops = number of flops done per processor

#words_moved per processor = $\Omega(\#flops / M^{1/2})$

#messages_sent per processor = $\Omega(\#flops / M^{3/2})$

- Holds for
  - Matmul, BLAS, LU, QR, eig, SVD, tensor contractions, …
  - Some whole programs (sequences of these operations, no matter how they are interleaved, eg computing A^k)
  - Dense *and* sparse matrices (where #flops << n^3)
  - Sequential *and* parallel algorithms
  - Some graph-theoretic algorithms (eg Floyd-Warshall)
- Proof later

New lower bound for all "direct" linear algebra

Let M = "fast" memory size per processor
= cache size (sequential case) or $O(n^2/p)$ (parallel case)
#flops = number of flops done per processor
#words_moved per processor = $\Omega(\#flops / M^{1/2})$
#messages_sent per processor = $\Omega(\#flops / M^{3/2})$

- Sequential case, dense n x n matrices, so $O(n^3)$ flops
  - #words_moved = $\Omega(n^3 / M^{1/2})$
  - #messages_sent = $\Omega(n^3 / M^{3/2})$
- Parallel case, dense n x n matrices
  - Load balanced, so $O(n^3/p)$ flops processor
  - One copy of data, load balanced, so M = $O(n^2/p)$ per processor
  - #words_moved = $\Omega(n^2 / p^{1/2})$
  - #messages_sent = $\Omega(p^{1/2})$
  -

Can we attain these lower bounds?

- Do conventional dense algorithms as implemented in LAPACK and ScaLAPACK attain these bounds?
  - Mostly not
- If not, are there other algorithms that do?
  - Yes
- Goals for algorithms:
  - Minimize #words_moved
  - Minimize #messages_sent
    - Need new data structures
  - Minimize for multiple memory hierarchy levels
    - Cache-oblivious algorithms would be simplest
  - Fewest flops when matrix fits in fastest memory
    - Cache-oblivious algorithms don't always attain this
- Attainable for nearly all dense linear algebra
  - Just a few prototype implementations so far (class projects!)
  - Only a few sparse algorithms so far (eg Cholesky)

Data Dependence and Related Definitions

- **Definition:**
  Two memory accesses are involved in a data dependence if they may refer to the same memory location and one of the references is a write.

  A data dependence can either be between two distinct program statements or two different dynamic executions of the same program statement.

- **Source:**
  - "Optimizing Compilers for Modern Architectures: A Dependence-Based Approach", Allen and Kennedy, 2002, Ch. 2.

Fundamental Theorem of Dependence

- **Theorem 2.2:**
  - Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.

In this course, we consider two kinds of reordering transformations

- Parallelization
  - Computations that execute in parallel between synchronization points are potentially reordered. Is that reordering safe? According to our definition, it is safe if it preserves the dependences in the code.
- Locality optimizations
  - Suppose we want to modify the order in which a computation accesses memory so that it is more likely to be in cache. This is also a reordering transformation, and it is safe if it preserves the dependences in the code.
- Reduction computations
  - We have to relax this rule for reductions. It is safe to reorder reductions for commutative and associative operations.

Targets of Memory Hierarchy Optimizations

- Reduce ***memory latency***
  - The latency of a memory access is the time (usually in cycles) between a memory request and its completion
- Maximize ***memory bandwidth***
  - Bandwidth is the amount of useful data that can be retrieved over a time interval
- Manage overhead
  - Cost of performing optimization (e.g., copying) should be less than anticipated gain

Reuse and Locality

- Consider how data is accessed
  - ***Data reuse:***
    - Same or nearby data used multiple times
    - Intrinsic in computation
  - ***Data locality:***
    - Data is reused and is present in "fast memory"
    - Same data or same data transfer
- If a computation has reuse, what can we do to get locality?
  - Appropriate data placement and layout
  - Code reordering transformations

Exploiting Reuse: Locality optimizations

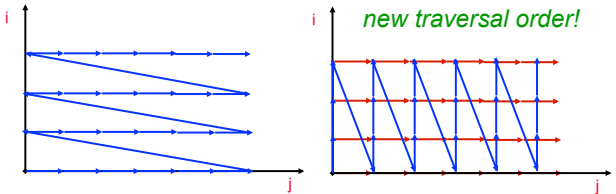
- We will study a few loop transformations that reorder memory accesses to improve locality.
- These transformations are also useful for parallelization too (to be discussed later).
- Two key questions:
  - Safety:
    - Does the transformation preserve dependences?
  - Profitability:
    - Is the transformation likely to be profitable?
    - Will the gain be greater than the overheads (if any) associated with the transformation?

Loop Transformations: Loop Permutation

Permute the order of the loops to modify the traversal order

```
for (i= 0; i<3; i++)
 for (j=0; j<6; j++)
   A[i][j+1]=A[i][j]+B[j];
```

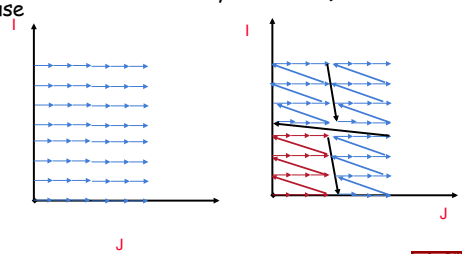
```
for (j=0; j<6; j++)
 for (i= 0; i<3; i++)
   A[i][j+1]=A[i][j]+B[j];
```



NOTE: C multi-dimensional arrays are stored in row-major order, Fortran in column major

Tiling (Blocking): Another Loop Reordering Transformation

- Blocking reorders loop iterations to bring iterations that reuse data closer in time
- Goal is to retain in cache/register/scratchpad (or other constrained memory structure) between reuse



Tiling Example

```
for (j=1; j<M; j++)
   for (i=1; i<N; i++)
     D[i] = D[i] +B[j,i]
```

Strip mine

```
for (j=1; j<M; j++)
   for (ii=1; ii<N; ii+=s)
      for (i=ii; i<min(ii+s-1,N); i++)
         D[i] = D[i] +B[j,i]
```

Permute

```
for (ii=1; ii<N; ii+=s)
   for (j=1; j<M; j++)
      for (i=ii; i<min(ii+s-1,N); i++)
         D[i] = D[i] +B[j,i]
```


Unroll, Unroll-and-Jam

- Unroll simply replicates the statements in a loop, with the number of copies called the unroll factor
- As long as the copies don't go past the iterations in the original loop, it is always safe
  - May require "cleanup" code
- Unroll-and-jam involves unrolling an outer loop and fusing together the copies of the inner loop (not always safe)
- One of the most effective optimizations there is, but there is a danger in unrolling too much

Original:
```
for (i=0; i<4; i++)
 for (j=0; j<8; j++)
  A[i][j] = B[j+1][i];
```

Unroll j
```
for (i=0; i<4; i++)
 for (j=0; j<8; j+=2)
  A[i][j] = B[j+1][i];
  A[i][j+1] = B[j+2][i];
```

Unroll-and-jam i
```
for (i= 0; i<4; i+=2)
 for (j=0; j<8; j++)
   A[i][j] = B[j+1][i];
   A[i+1][j] = B[j+1][i+1];
```


How does Unroll-and-Jam benefit locality?

Original:
```
for (i=0; i<4; i++)
 for (j=0; j<8; j++)
  A[i][j] = B[j+1][i] + B[j+1][i+1];
```

Unroll-and-jam i and j loops
```
for (i=0; i<4; i+=2)
 for (j=0; j<8; j+=2) {
  A[i][j]   = B[j+1][i] + B[j+1][i+1];
  A[i+1][j] = B[j+1][i+1] + B[j+1][i+2];
  A[i][j+1] = B[j+2][i] + B[j+2][i+1];
  A[i+1][j+1] B[j+2][i+1] + B[j+2][i+2];
}
```

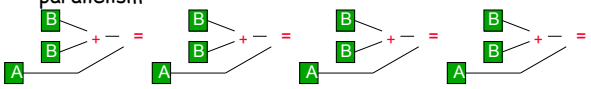
- Temporal reuse of B in registers
- More if I loop is unrolled further

Other advantages of Unroll-and-Jam

Original:
```
for (i=0; i<4; i++)
 for (j=0; j<8; j++)
  A[i][j] = B[j+1][i] + B[j+1][i+1];
```

Unroll-and-jam i and j loops
```
for (i=0; i<4; i+=2)
 for (j=0; j<8; j+=2) {
  A[i][j]   = B[j+1][i] + B[j+1][i+1];
  A[i+1][j] = B[j+1][i+1] + B[j+1][i+2];
  A[i][j+1] = B[j+2][i] + B[j+2][i+1];
  A[i+1][j+1] B[j+2][i+1] + B[j+2][i+2];
}
```

- Less loop control
- Independent computations for instruction-level parallelism



How to determine safety of reordering transformations

- Informally
  - Must preserve relative order of dependence source and sink
  - So, cannot reverse order
- Formally
  - Tracking dependences

Safety of Permutation

- Cannot reverse any dependences
- Ok to permute?

```
for (i= 0; i<3; i++)
  for (j=0; j<6; j++)
    A[i][j+1]=A[i][j]+B[j];
```

```
for (i= 0; i<3; i++)
  for (j=1; j<6; j++)
    A[i+1][j-1]=A[i][j]+B[j];
```


Safety of Tiling

- Tiling = strip-mine and permutation
  - Strip-mine does not reorder iterations
  - Permutation must be legal

  OR
  - strip size less than dependence distance

Safety of Unroll-and-Jam

- Unroll-and-jam = tile + unroll
  - Permutation must be legal

  OR
  - unroll less than dependence distance

Unroll-and-jam = tile + unroll?

Original:

```
for (i=0; i<4; i++)
  for (j=0; j<8; j++)
    A[i][j] = B[j+1][i];
```

Tile i loop:

```
for (ii=0; ii<4; ii+=2)
  for (j=0; j<8; j++)
    for (i=ii; i<ii+2; i++)
      A[i][j] = B[j+1][i];
```

Unroll i tile:

```
for (ii= 0; ii<4; ii+=2)
  for (j=0; j<8; j++)
    A[i][j] = B[j+1][i];
    A[i+1][j] = B[j+1][i+1];
```