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
Why avoiding communication is important (2/2)

- Running time of an algorithm is sum of 3 terms:
  - \( \# \text{ flops} \times \text{time per flop} \)
  - \( \# \text{ words moved} / \text{ bandwidth} \)
  - \( \# \text{ messages} \times \text{latency} \)

- Time per flop \(< 1/\text{bandwidth} < \text{latency} \)
- Gaps growing exponentially with time

- 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 \( \leq 2x \))
  - Arbitrary speedups possible

<table>
<thead>
<tr>
<th>Annual improvements</th>
<th>Time per flop</th>
<th>Bandwidth</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>59%</td>
<td>Network</td>
<td>26%</td>
<td>15%</td>
</tr>
<tr>
<td></td>
<td>DRAM</td>
<td>23%</td>
<td>5%</td>
</tr>
</tbody>
</table>

Review: Naïve Sequential MatMul: \( C = C + A \cdot B \)

for \( i = 1 \) to \( n \)
  for \( j = 1 \) to \( n \)
    \( \text{read row } i \text{ of } A \text{ into fast memory, } n^2 \text{ reads} \)
    \( \text{read } C(i,j) \text{ into fast memory, } n^2 \text{ reads} \)
    \( \text{read column } j \text{ of } B \text{ into fast memory, } n^3 \text{ reads} \)
  for \( k = 1 \) to \( n \)
    \( C(i,j) = C(i,j) + A(i,k) \cdot B(k,j) \)
    \( \text{write } C(i,j) \text{ back to slow memory, } n^2 \text{ writes} \)

Less Communication with Blocked Matrix Multiply

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

  \( \text{... Break } A^{nxn}, B^{nxn}, C^{nxn} \text{ into } bxb \text{ blocks labeled } A(i,j), \text{ etc} \)
  \( \text{... } b \text{ chosen so } 3 bxb \text{ blocks fit in cache} \)
  \( \text{for } i = 1 \text{ to } nb, \text{ for } j = 1 \text{ to } nb, \text{ for } k = 1 \text{ to } nb \)
  \( C(i,j) = C(i,j) + A(i,k) \cdot B(k,j) \)  \( \text{... } b \times b \text{ matmul, } 4b^2 \text{ reads/writes} \)

- \( (nb)^2 \cdot 4b^2 = 4n^3/b \text{ reads/writes altogether} \)
- Minimized when \( 3b^2 = \text{cache size} = M \), yielding \( O(n^3/M^{1/2}) \text{ reads/writes} \)
- What if we had more levels of memory? (L1, L2, cache etc)?
  - Would need 3 more nested loops per level

Blocked vs Cache-Oblivious Algorithms

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

  \( \text{... Break } A^{nxn}, B^{nxn}, C^{nxn} \text{ into } bxb \text{ blocks labeled } A(i,j), \text{ etc} \)
  \( \text{... } b \text{ chosen so } 3 bxb \text{ blocks fit in cache} \)
  \( \text{for } i = 1 \text{ to } nb, \text{ for } j = 1 \text{ to } nb, \text{ for } k = 1 \text{ to } nb \)
  \( C(i,j) = C(i,j) + A(i,k) \cdot B(k,j) \)  \( \text{... } b \times b \text{ matmul} \)
  \( \text{... another level of memory would need 3 more loops} \)

- Cache-oblivious Matmul \( C = A \cdot B \) is independent of cache

Function \( C = \text{RMM}(A,B) \)  \( \text{... R for recursive} \)
  - If \( A \) and \( B \) are 1x1
    \( C = A \cdot B \)
  - else \( \text{... Break } A^{nxn}, B^{nxn}, C^{nxn} \text{ into } (n/2)(n/2) \text{ blocks labeled } A(i,j), \text{ etc} \)
    \( \text{for } i = 1 \text{ to } 2, \text{ for } j = 1 \text{ to } 2, \text{ for } k = 1 \text{ to } 2 \)
    \( C(i,j) = C(i,j) + \text{RMM}(A(i,k), B(k,j)) \)  \( \text{... } n^2 \times n^2 \text{ matmul} \)
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, e.g. computing $A^k$)
  - Dense and sparse matrices (where #flops << $n^3$)
  - Sequential and parallel algorithms
  - Some graph-theoretic algorithms (e.g. Floyd-Warshall)
- Proof later

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 (e.g. 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:**

---

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++)
```

```
for (j=0; j<6; j++)
  for (i=0; i<3; i++)
```

new traversal order!

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

```
for (i= 0; i<3; i++)
  for (j=0; j<6; j++)
```
Tiling Example

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

Strip mine

```c
for (i=1; i<N; i++)
    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

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

How does Unroll-and-Jam benefit locality?

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

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

Other advantages of Unroll-and-Jam

- 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?

```c
for (i=0; i<3; i++)
for (j=0; j<6; 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 & loop:
for (i=0; i<4; i++)
for (j=0; j<8; j++)
for (i=ii; i<ii+2; i++)
A[i][j] = B[j+1][i];

Unroll & tile:
for (i=0; i<4; i++)
for (j=0; j<8; j++)
A[i][j] = B[j+1][i];
A[i+1][j] = B[j+1][i+1];