Performance Implications of Blocking/Tiling

Murat Manguoglu
CENG577
Spring 2019
An example: Banded matrix – matrix (multiple vectors) Multiplication

- What is a banded matrix? → A matrix where

\[ a_{ij} = 0 \text{ if } j - i > \beta_U \text{ or } i - j > \beta_L \]

\[ \beta = \beta_U + \beta_L + 1 : = \text{bandwidth} \]

\[ A = \]

\[ \begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix} \]
A Naïve approach: use what is available

Objective is to compute $C = A \times B$ where $A$ is $n \times n$ banded $B, C$ are $n \times k$ dense matrices

$$
\begin{pmatrix}
0 & 0 \\
0 & \end{pmatrix}
\begin{pmatrix}
\end{pmatrix}
= \\
\begin{pmatrix}
\end{pmatrix}
$$

Using BLAS:

for $i=1$ to $k$
    call DGBEMV for each $B(:,i)$
    (to obtain $C(:,i)$)
end
Alternative: Implementing the multiplication by blocking

Objective is to compute $C = A \times B$ where $A$ is $n \times n$ banded, $B, C$ are $n \times k$ dense matrices.

For each block row we use DGEMM + 2 DTRMM. These are all BLAS Level 3 operations.

We call this algorithm DSBMM2\(^1\) which is an improved DSBMM\(^2\).

\(^1\) Meftun Cincioglu, Parallel sparse and banded matrix - multiple vectors multiplication, Msc Thesis, 2014 METU
\(^2\) https://github.com/certik/feast/blob/master/src/lbprim.f90
speed improvement for matrix - matrix multiplication (n = 160000)
What **more** can be done with blocking?

(the following slides are obtained from: [https://sites.google.com/lbl.gov/cs267-spr2018/](https://sites.google.com/lbl.gov/cs267-spr2018/))
Memory Hierarchy

- Memory hierarchy exploits **locality** to improve average case performance
  - **spatial locality**: accessing things nearby previous accesses
  - **temporal locality**: reusing an item that was previously accessed

- **Roofline model**: upper limit of bandwidth and compute speed (more on this next week)

- **Tiling**: adding nested loops to improve temporal locality

- **Communication lower bound**: minimum data movement necessary for the algorithm
Consider $A, B, C$ to be $N$-by-$N$ matrices of $b$-by-$b$ subblocks where $b = n / N$ is called the block size.

For $i = 1$ to $N$
   For $j = 1$ to $N$
      {read block $C(i,j)$ into fast memory}
      {read block $A(i,k)$ into fast memory}
      {read block $B(k,j)$ into fast memory}
      $C(i,j) = C(i,j) + A(i,k) \times B(k,j)$ {do a matrix multiply on blocks}
      {write block $C(i,j)$ back to slow memory}

Tiling for registers (managed by you/compiler) or caches (hardware)
Theory: Communication lower bounds for Matmul

Theorem (Hong & Kung, 1981):
Any reorganization of matmul (using only associativity) has computational intensity $q = O\left( (M_{\text{fast}})^{1/2} \right)$, so

$\#\text{words moved between fast/slow memory} = \Omega \left( \frac{n^3}{(M_{\text{fast}})^{1/2}} \right)$

Useful work max:
Matmul does $O(n^3)$ work on $O(n^2)$ data, in cache $n=\sqrt{M}$
so $q = O(\sqrt{M})$

Data transferred min:
All work
Useful work in cache = $\Omega \left( \frac{n^3}{M^{1/2}} \right)$

- Cost also depends on the number of “messages” (e.g., cache lines)
  - $\#\text{messages} = \Omega \left( \frac{n^3}{M_{\text{fast}}^{3/2}} \right)$
- Tiled matrix multiply (with tile size $= \frac{M_{\text{fast}}^{1/2}}{3}$) achieves this lower bound
- Lower bounds extend to similar programs nested loops accessing arrays
Practice: Tile Size Selection for Cache-Aware Tiling

- Maximize $b$, but small enough to fit in cache (or in registers)
  - Avoid interference: depends on $n$, $b$, cache associativity, etc.
  - Not necessarily square block (row / column accesses different)

Hundreds of papers on this; above from [Mehtra, Beeraka, Yew, 2013]
Tuning Code in Practice

• Tuning code can be tedious
  - Many optimizations besides blocking
  - Behavior of machine and compiler hard to predict

• Approach #1: Analytical performance models
  - Use model (of cache, memory costs, etc.) to select best code
  - But model needs to be both simple and accurate 🙄

• Approach #2: “Autotuning”
  - Let computer generate large set of possible code variations, and search for the fastest ones
  - Sometimes all done “off-line”, sometimes at run-time
A 2-D slice of a 3-D register-tile search space. The dark blue region was pruned. (Platform: Sun Ultra-IIi, 333 MHz, 667 Mflop/s peak, Sun cc v5.0 compiler)
• ATLAS is faster than all other portable BLAS implementations and it is comparable with machine-specific libraries provided by the vendor.
What about move levels of memory?

• Need to minimize communication between all levels
  • Between L1 and L2 cache, cache and DRAM, DRAM and disk…

• The tiled algorithm requires finding a good block size
  • Machine dependent (cache aware – block size matched to hardware)
  • Need to “block” b x b matrix multiply in inner most loop
    • 1 level of memory ⇒ 3 nested loops (naïve algorithm)
    • 2 levels of memory ⇒ 6 nested loops
    • 3 levels of memory ⇒ 9 nested loops …

• Cache Oblivious Algorithms offer an alternative
  • Treat nxn matrix multiply as a set of smaller problems
  • Eventually, these will fit in cache
  • Will minimize # words moved between every level of memory hierarchy – at least asymptotically
  • “Oblivious” to number and sizes of levels
Recursive Matrix Multiplication (RMM) (1/2)

\[
C = \begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix} = A \cdot B = \begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix} \cdot \begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
\]

\[
= \begin{pmatrix}
A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\
A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22}
\end{pmatrix}
\]

- True when each block is a 1x1 or n/2 x n/2
- For simplicity: square matrices with n = 2^m
  - Extends to general rectangular case
Recursive Matrix Multiplication (2/2)

```
func C = RMM (A, B, n)
  if n=1, C = A * B, else
    {  C
        11  = RMM (A
                     11 , B
                     11 , n/2) + RMM (A
                     12 , B
                     21 , n/2)
        C
        12  = RMM (A
                     11 , B
                     12 , n/2) + RMM (A
                     12 , B
                     22 , n/2)
        C
        21  = RMM (A
                     21 , B
                     11 , n/2) + RMM (A
                     22 , B
                     21 , n/2)
        C
        22  = RMM (A
                     21 , B
                     12 , n/2) + RMM (A
                     22 , B
                     22 , n/2)  }
  return
```

A(n) = # arithmetic operations in RMM( . , . , n)
  = 8 · A(n/2) + 4(n/2)² if  n > 1,  else 1
  = 2n³ … same operations as usual, in different order

W(n) = # words moved between fast, slow memory by RMM( . , . , n)
  = 8 · W(n/2) + 4· 3(n/2)² if  3n² > M_{fast} , else 3n²
  = O( n³ / (M_{fast})^{1/2} + n² ) … same as blocked matmul

Don’t need to know M_{fast} for this to work!
Experience with Cache-Oblivious Algorithms

- In practice, need to cut off recursion well before 1x1 blocks
  - Call “micro-kernel” on small blocks
  - Careful attention to micro-kernel is needed
- Pingali et al report that they never got more than 2/3 of peak.
  - Fully recursive approach with highly optimized recursive micro-kernel
  - (unpublished, presented at LACSI’06)

- Issues with Cache Oblivious (recursive) approach
Recursive Data Layouts

- A related idea is to use a recursive structure for the matrix
  - Improve locality with machine-independent data structure
  - Can minimize latency with multiple levels of memory hierarchy
- There are several possible recursive decompositions depending on the order of the sub-blocks
- This figure shows Z-Morton Ordering (“space filling curve”)
- See papers on “cache oblivious algorithms” and “recursive layouts”

Advantages:
- the recursive layout works well for any cache size

Disadvantages:
- The index calculations to find $A[i,j]$ are expensive
- Implementations switch to column-major for small sizes
Optimizing in Practice

• Tiling for registers
  • loop unrolling, use of named “register” variables
• Tiling for multiple levels of cache and TLB
• Exploiting fine-grained parallelism in processor
  • superscalar; pipelining
• Complicated compiler interactions (flags)
• Hard to do by hand (but you’ll try)
• Automatic optimization an active research area
  • ASPIRE: aspire.eecs.berkeley.edu
  • BeBOP: bebop.cs.berkeley.edu
    • Weekly group meeting Mondays 1pm
  • PHiPAC: www.icsi.berkeley.edu/~bilmes/phipac
    in particular tr-98-035.ps.gz
  • ATLAS: www.netlib.org/atlas
Removing False Dependencies

• Using local variables, reorder operations to remove false dependencies

```plaintext
a[i] = b[i] + c;  \text{false read-after-write hazard}
\downarrow

float f1 = b[i];  \text{between } a[i] \text{ and } b[i+1]
float f2 = b[i+1];

a[i] = f1 + c;
a[i+1] = f2 \times d;
```

With some compilers, you can declare a and b unaliased.
• Done via “restrict pointers,” compiler flag, or pragma
• In Fortran, can use function calls (arguments assumed unaliased, maybe).
Exploit Multiple Registers

- Reduce demands on memory bandwidth by pre-loading into local variables

```c
while( ... ) {
    *res++ = filter[0]*signal[0]
    + filter[1]*signal[1]
    + filter[2]*signal[2];
    signal++;
}
```

also:
```
register float f0 = ...;
```

Example is a convolution
Loop Unrolling

- Expose instruction-level parallelism

```c
float f0 = filter[0], f1 = filter[1], f2 = filter[2];
float s0 = signal[0], s1 = signal[1], s2 = signal[2];
*res++ = f0*s0 + f1*s1 + f2*s2;
do {
    signal += 3;
    s0 = signal[0];
    res[0] = f0*s1 + f1*s2 + f2*s0;
    s1 = signal[1];
    res[1] = f0*s2 + f1*s0 + f2*s1;
    s2 = signal[2];
    res[2] = f0*s0 + f1*s1 + f2*s2;
    res += 3;
} while( ... );
```
Expose Independent Operations

• Hide instruction latency
  • Use local variables to expose independent operations that can execute in parallel or in a pipelined fashion
  • Balance the instruction mix (what functional units are available?)

```plaintext
f1 = f5 * f9;
f2 = f6 + f10;
f3 = f7 * f11;
f4 = f8 + f12;
```
Minimize Pointer Updates

• Replace pointer updates for strided memory addressing with constant array offsets

\[
\begin{align*}
f_0 &= *r8; \quad r8 += 4; \\
f_1 &= *r8; \quad r8 += 4; \\
f_2 &= *r8; \quad r8 += 4; \\
\end{align*}
\]

\[
\downarrow
\]

\[
\begin{align*}
f_0 &= r8[0]; \\
f_1 &= r8[4]; \\
f_2 &= r8[8]; \\
r8 &= r8[12];
\end{align*}
\]

Pointer vs. array expression costs may differ.

• Some compilers do a better job at analyzing one than the other
# Copy optimization

- Copy input operands or blocks
  - Reduce cache conflicts
  - Constant array offsets for fixed size blocks
  - Expose page-level locality
  - Alternative: use different data structures from start (if users willing)
    - Recall recursive data layouts

## Original matrix (numbers are addresses)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

## Reorganized into 2x2 blocks

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>2</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>9</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>12</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>
Strassen’s Matrix Multiply

• The traditional algorithm (with or without tiling) has $O(n^3)$ flops
• Strassen discovered an algorithm with asymptotically lower flops
  • $O(n^{2.81})$
• Consider a 2x2 matrix multiply, normally takes 8 multiplies, 4 adds
  • Strassen does it with 7 multiplies and 18 adds

Let $M = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$

Let $p_1 = (a_{12} - a_{22}) \cdot (b_{21} + b_{22})$  \hspace{1cm} p_5 = a_{11} \cdot (b_{12} - b_{22})$
$p_2 = (a_{11} + a_{22}) \cdot (b_{11} + b_{22})$  \hspace{1cm} p_6 = a_{22} \cdot (b_{21} - b_{11})$
$p_3 = (a_{11} - a_{21}) \cdot (b_{11} + b_{12})$  \hspace{1cm} p_7 = (a_{21} + a_{22}) \cdot b_{11}$
$p_4 = (a_{11} + a_{12}) \cdot b_{22} \hspace{1cm}$

Then $m_{11} = p_1 + p_2 - p_4 + p_6$
$m_{12} = p_4 + p_5$
$m_{21} = p_6 + p_7$
$m_{22} = p_2 - p_3 + p_5 - p_7$

Extends to nxn by divide&conquer
Strassen (continued)

\[
T(n) = \text{Cost of multiplying nxn matrices} \\
= 7\cdot T(\frac{n}{2}) + 18\cdot (\frac{n}{2})^2 \\
= O(n \log_2 7) \\
= O(n^{2.81})
\]

- Asymptotically faster
  - Several times faster for large \( n \) in practice
  - Cross-over depends on machine
  - “Tuning Strassen's Matrix Multiplication for Memory Efficiency”, M. S. Thottethodi, S. Chatterjee, and A. Lebeck, in Proceedings of Supercomputing '98

- Possible to extend communication lower bound to Strassen
  - \#words moved between fast and slow memory
    \[
    \Omega(n^{\log_2 7} / M^{(\log_2 7)/2 - 1}) \sim \Omega(n^{2.81} / M^{0.4})
    \]
    (Ballard, D., Holtz, Schwartz, 2011, SPAA Best Paper Prize)
  - Attainable too, more on parallel version later
Other Fast Matrix Multiplication Algorithms

• World’s record was $O(n^{2.37548...})$
  • Coppersmith & Winograd, 1987

• New Record! $2.37548$ reduced to $2.37293$
  • Virginia Vassilevska Williams, 2011

• Newer Record! $2.37293$ reduced to $2.37286$
  • Francois Le Gall, 2014

• Lower bound on #words moved can be extended to (some) of these algorithms (2015 thesis of Jacob Scott)

• Possibility of $O(n^{2+\varepsilon})$ algorithm!
  • Cohn, Umans, Kleinberg, 2003

• Can show they all can be made numerically stable
  • Demmel, Dumitriu, Holtz, Kleinberg, 2007

• Can do rest of linear algebra (solve $Ax=b$, $Ax=\lambda x$, etc) as fast, and numerically stably
  • Demmel, Dumitriu, Holtz, 2008

• Fast methods (besides Strassen) may need unrealistically large $n$