CS 140 : Matrix multiplication

- Warmup: Matrix times vector: communication volume
- Matrix multiplication I: parallel issues
- Matrix multiplication II: cache issues

Thanks to Jim Demmel and Kathy Yelick (UCB) for some of these slides
Matrix-Matrix Multiplication (“DGEMM”)

{implements $C = C + A*B$}
for $i = 1$ to $n$
  for $j = 1$ to $n$
    for $k = 1$ to $n$
      $C(i,j) = C(i,j) + A(i,k) * B(k,j)$

Work: $2*n^3$ flops
Memory: $3*n^2$ words
Parallel matrix multiply, \[ C = C + A \times B \]

- Basic sequential algorithm:
  - \[ C(i,j) += A(i,1) \times B(1,j) + A(i,2) \times B(1,j) + \ldots + A(i,n) \times B(n,j) \]
  - work = \( t_1 = 2n^3 \) floating point ops
- Highly parallel: \( t_p = \frac{2n^3}{p} \) is easy, for \( p \) up to at least \( n^2 \)
- The issue is communication cost, as affected by:
  - Data layout
  - Structure & schedule of communication
- Where’s the data?
Communication volume model

- Network of $p$ processors
  - Each with local memory
  - Message-passing

- Communication volume ($v$)
  - Total size (words) of all messages passed during computation
  - Broadcasting one word costs volume $p$ (actually, $p-1$)

- No explicit accounting for communication time
  - Thus, we can’t model parallel efficiency or speedup; for that, we’d use the *latency-bandwidth model* (see extra slides)
Parallel Matrix Multiply with 1D Column Layout

• Assume matrices are n x n and n is divisible by p

Let A(k) be the n-by-n/p block column that processor k owns
  • similarly B(k) and C(k)

\[
C(k) += A * B(k)
\]

• Now let B(i,k) be a subblock of B(k) with n/p rows

\[
C(k) += A(0) * B(0,k) + A(1) * B(1,k) + \ldots + A(p-1) * B(p-1,k)
\]

(A reasonable assumption for analysis, not for code)
Matmul for 1D layout on a Processor Ring

- Proc \( k \) communicates only with procs \( k-1 \) and \( k+1 \)
- Different pairs of processors can communicate simultaneously
- Round-Robin “Merry-Go-Round” algorithm

Copy \( A(\text{myproc}) \) into MGR \((\text{MGR} = \text{“Merry-Go-Round”})\)
\[
C(\text{myproc}) = C(\text{myproc}) + \text{MGR} \cdot B(\text{myproc}, \text{myproc})
\]
for \( j = 1 \) to \( p-1 \)
- send \( \text{MGR} \) to processor \( \text{myproc+1 mod p} \) \((\text{but see deadlock below})\)
- receive \( \text{MGR} \) from processor \( \text{myproc-1 mod p} \) \((\text{but see below})\)
\[
C(\text{myproc}) = C(\text{myproc}) + \text{MGR} \cdot B(\text{myproc-j mod p}, \text{myproc})
\]

- Avoiding deadlock:
  - even procs send then recv, odd procs recv then send
  - or, use nonblocking sends and be careful with buffers

- Comm volume of one inner loop iteration = \( n^2 \)
Matmul for 1D layout on a Processor Ring

- One iteration: \( v = n^2 \)

- All \( p-1 \) iterations: \( v = (p-1) \times n^2 \sim pn^2 \)

- Optimal for 1D data layout:
  - Perfect speedup for arithmetic
  - \( A(\text{myproc}) \) must meet each \( C(\text{myproc}) \)

- “Nice” communication pattern – can probably overlap independent communications in the ring.

- In latency/bandwidth model (see extra slides), parallel efficiency \( e = 1 - O(p/n) \)
**MatMul with 2D Layout**

- Consider processors in 2D grid (physical or logical)
- Processors can communicate with 4 nearest neighbors
  - Alternative pattern: broadcast along rows and columns

Assume p is square s x s grid
Cannon’s Algorithm: 2-D merry-go-round

... $C(i,j) = C(i,j) + \sum_{k} A(i,k) \cdot B(k,j)$
... assume $s = \sqrt{p}$ is an integer
  forall $i=0$ to $s-1$ ... “skew” A
    left-circular-shift row $i$ of $A$ by $i$
  ... so that $A(i,j)$ overwritten by $A(i,(j+i)\mod s)$
  forall $i=0$ to $s-1$ ... “skew” B
    up-circular-shift $B$ column $i$ of $B$ by $i$
  ... so that $B(i,j)$ overwritten by $B((i+j)\mod s), j)$
  for $k=0$ to $s-1$ ... sequential
    forall $i=0$ to $s-1$ and $j=0$ to $s-1$ ... all processors in parallel
      $C(i,j) = C(i,j) + A(i,j) \cdot B(i,j)$
    left-circular-shift each row of $A$ by 1
    up-circular-shift each row of $B$ by 1
Cannon’s Matrix Multiplication

Cannon’s Matrix Multiplication Algorithm

Initial A, B

A, B after skewing

A, B after shift k=1

A, B after shift k=2

\[ C(1,2) = A(1,0) \times B(0,2) + A(1,1) \times B(1,2) + A(1,2) \times B(2,2) \]
Initial Step to Skew Matrices in Cannon

- Initial blocked input

- After skewing before initial block multiplies
Skewing Steps in Cannon

- First step
  - A(0,0)  A(0,1)  A(0,2)
  - A(1,1)  A(1,2)  A(1,0)
  - A(2,2)  A(2,0)  A(2,1)
  - B(0,0)  B(1,1)  B(2,2)
  - B(1,0)  B(2,1)  B(0,2)
  - B(2,0)  B(0,1)  B(1,2)

- Second
  - A(0,1)  A(0,2)  A(0,0)
  - A(1,2)  A(1,0)  A(1,1)
  - A(2,0)  A(2,1)  A(2,2)
  - B(1,0)  B(2,1)  B(0,2)
  - B(2,0)  B(0,1)  B(1,2)
  - B(0,0)  B(1,1)  B(2,2)

- Third
  - A(0,2)  A(0,0)  A(0,1)
  - A(1,0)  A(1,1)  A(1,2)
  - A(2,1)  A(2,2)  A(2,0)
  - B(2,0)  B(0,1)  B(1,2)
  - B(0,0)  B(1,1)  B(2,2)
  - B(1,0)  B(2,1)  B(0,2)
Communication Volume of Cannon’s Algorithm

forall i=0 to s-1
  left-circular-shift row i of A by i
forall i=0 to s-1
  up-circular-shift B column i of B by i
for k=0 to s-1
  forall i=0 to s-1 and j=0 to s-1
    C(i,j) = C(i,j) + A(i,j)*B(i,j)
  left-circular-shift each row of A by 1
  up-circular-shift each row of B by 1

° Total comm \( v = 2*n^2 + 2*s*n^2 \sim 2*\sqrt{p}*n^2 \)
° Computational intensity \( q = t_1 / v \sim n / \sqrt{p} \)
° In latency/bandwidth model (see extra slides), parallel efficiency \( e = 1 - O(\sqrt{p} / n) \)
Cannon is beautiful, but maybe too beautiful ...

- Drawbacks to Cannon:
  - Hard to generalize for $p$ not a perfect square, $A$ & $B$ not square, dimensions not divisible by $s = \sqrt{p}$, different memory layouts, etc.
  - Memory hog – needs extra copies of local matrices.

- Algorithm used instead in practice is **SUMMA**:  
  - uses row and column broadcasts, not merry-go-round
  - see extra slides below for details

- Comparing *computational intensity* = work / comm volume:
  - 1-D MGR has computational intensity $q = O(n / p)$
  - Cannon has computational intensity $q = O(n / \sqrt{p})$
  - SUMMA has computational intensity $q = O(n / \sqrt{p} \log p)$
Sequential Matrix Multiplication

Simple mathematics, but getting good performance is complicated by memory hierarchy --- cache issues.
Naïve 3-loop matrix multiply

\{\text{implements } C = C + A \times B\}

for \( i = 1 \) to \( n \)
  
  for \( j = 1 \) to \( n \)
    
    for \( k = 1 \) to \( n \)
      
      \[ C(i,j) = C(i,j) + A(i,k) \times B(k,j) \]

\begin{align*}
\text{Work: } & 2n^3 \text{ flops} \\
\text{Memory: } & 3n^2 \text{ words}
\end{align*}
O(N³) performance would have constant cycles/flop
Performance looks much closer to O(N⁵)
3-Loop Matrix Multiply [Alpern et al., 1992]

- Page miss every iteration
- TLB miss every iteration
- Cache miss every 16 iterations
- Page miss every 512 iterations

The graph shows the relationship between log cycles/flop and log problem size. The x-axis represents the log of the problem size, while the y-axis represents the log of cycles/flop.
Avoiding data movement: Reuse and locality

- Large memories are slow, fast memories are small
- Parallel processors, collectively, have large, fast cache
  - the slow accesses to “remote” data we call “communication”
- Algorithm should do most work on local data
Simplified model of hierarchical memory

- Assume just 2 levels in the hierarchy, fast and slow
- All data initially in slow memory
  - \( m \) = number of memory elements (words) moved between fast and slow memory
  - \( t_m \) = time per slow memory operation
  - \( f \) = number of arithmetic operations
  - \( t_f \) = time per arithmetic operation \(< t_m\)
  - \( q = f / m \) (computational intensity) flops per slow element access
- Minimum possible time = \( f * t_f \) when all data in fast memory
- Actual time
  - \( f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/q) \)
- Larger \( q \) means time closer to minimum \( f * t_f \)
“Naïve” Matrix Multiply

{implements $C = C + A*B$}

for $i = 1$ to $n$

{read row $i$ of $A$ into fast memory}

for $j = 1$ to $n$

{read $C(i,j)$ into fast memory}

{read column $j$ of $B$ into fast memory}

for $k = 1$ to $n$

\[ C(i,j) = C(i,j) + A(i,k) \times B(k,j) \]

{write $C(i,j)$ back to slow memory}
“Naïve” Matrix Multiply

How many references to slow memory?

\[
m = n^3 \text{ read each column of } B \text{ } n \text{ times } \\
+ \ n^2 \text{ read each row of } A \text{ once } \\
+ \ 2n^2 \text{ read and write each element of } C \text{ once } \\
= n^3 + 3n^2
\]

So \( q = f / m = \frac{2n^3}{(n^3 + 3n^2)} \)
\(~= 2 \text{ for large } n~\)
**Blocked Matrix Multiply**

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}
  for k = 1 to N
    {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) * B(k,j) \{do a matrix multiply on blocks\}
    {write block C(i,j) back to slow memory}
**Blocked Matrix Multiply**

$m$ is amount memory traffic between slow and fast memory matrix has $nxn$ elements, and $NxN$ blocks each of size $bxb$

$f$ is number of floating point operations, $2n^3$ for this problem

$q = f / m$ measures data reuse, or computational intensity

\[
m = N^2 n^2 \quad \text{read every block of B N times}
+ N^2 n^2 \quad \text{read every block of A N times}
+ 2n^2 \quad \text{read and write every block of C once}
= (2N + 2) * n^2
\]

Computational intensity $q = f / m = 2n^3 / ((2N + 2) * n^2)$

$\sim n / N = b$ for large $n$

We can improve performance by increasing the blocksize $b$

(but only until $3b^2$ gets as big as the fast memory size)

Can be much faster than matrix-vector multiply ($q = 2$)
Multi-Level Blocked Matrix Multiply

- More levels of memory hierarchy => more levels of blocking!

- Version 1: One level of blocking for each level of memory (L1 cache, L2 cache, L3 cache, DRAM, disk, ...)

- Version 2: Recursive blocking, $O(\log n)$ levels deep

In the “Uniform Memory Hierarchy” cost model, the 3-loop algorithm is $O(N^5)$ time, but the blocked algorithms are $O(N^3)$
**BLAS: Basic Linear Algebra Subroutines**

- Industry standard interface
- Vendors, others supply optimized implementations
- **History**
  - BLAS1 (1970s):
    - vector operations: dot product, saxpy (\(y=\alpha \cdot x+y\)), etc
    - \(m=2\cdot n\), \(f=2\cdot n\), \(q \sim 1\) or less
  - BLAS2 (mid 1980s)
    - matrix-vector operations: matrix vector multiply, etc
    - \(m=n^2\), \(f=2\cdot n^2\), \(q\sim 2\), less overhead
    - somewhat faster than BLAS1
  - BLAS3 (late 1980s)
    - matrix-matrix operations: matrix matrix multiply, etc
    - \(m \geq n^2\), \(f=O(n^3)\), so \(q\) can possibly be as large as \(n\)
    - BLAS3 is potentially much faster than BLAS2
- **Good algorithms use BLAS3 when possible (LAPACK)**
  - See [www.netlib.org/blas](http://www.netlib.org/blas), [www.netlib.org/lapack](http://www.netlib.org/lapack)
BLAS speeds on an IBM RS6000/590

Peak speed = 266 Mflops

BLAS 3 (n-by-n matrix matrix multiply) vs
BLAS 2 (n-by-n matrix vector multiply) vs
BLAS 1 (saxpy of n vectors)
ScaLAPACK Parallel Library

ScaLAPACK SOFTWARE HIERARCHY

ScaLAPACK

PBLAS

LAPACK

BLACS

BLAS

Message Passing Primitives
(MPI, PVM, etc.)

Global

Local
Extra Slides:

Parallel matrix multiplication in the latency-bandwidth cost model
Latency Bandwidth Model

• Network of $p$ processors, each with local memory
  • Message-passing

• Latency ($\alpha$)
  • Cost of communication per message

• Inverse bandwidth ($\beta$)
  • Cost of communication per unit of data

• Parallel time ($t_p$)
  • Computation time plus communication time

• Parallel efficiency:
  • $e(p) = \frac{t_1}{p \cdot t_p}$
  • perfect speedup $\rightarrow e(p) = 1$
Matrix Multiply with 1D Column Layout

- Assume matrices are n x n and n is divisible by p

- A(k) is the n-by-n/p block column that processor k owns (similarly B(i) and C(i))

- B(i,k) is an n/p-by-n/p subblock of B(k)
  - in rows i*n/p through (i+1)*n/p

- Formula:  \( C(k) = C(k) + A \cdot B(k) = C(k) + \sum_{i=0:p} A(i) \cdot B(i,k) \)
Matmul for 1D layout on a Processor Ring

• Proc $k$ communicates only with procs $k-1$ and $k+1$
• Different pairs of processors can communicate simultaneously
• Round-Robin “Merry-Go-Round” algorithm

Copy $A(myproc)$ into MGR

$C(myproc) = C(myproc) + MGR*B(myproc, myproc)$

for $j = 1$ to $p-1$

send MGR to processor $myproc+1 \mod p$ (but see deadlock below)
receive MGR from processor $myproc-1 \mod p$ (but see below)
$C(myproc) = C(myproc) + MGR * B(myproc-j \mod p, myproc)$

• Avoiding deadlock:
  • even procs send then recv, odd procs recv then send
  • or, use nonblocking sends

• Time of inner loop $= 2*(\alpha + \beta*n^2/p) + 2*n^*(n/p)^2$
Matmul for 1D layout on a Processor Ring

- Time of inner loop = \(2*(\alpha + \beta*n^2/p) + 2*n*(n/p)^2\)

- Total Time = \(2*n* (n/p)^2 + (p-1) * \text{Time of inner loop}\)
  \[\sim 2*n^3/p + 2*p* \alpha + 2* \beta*n^2\]

- Optimal for 1D layout on Ring or Bus, even with broadcast:
  - Perfect speedup for arithmetic
  - \(A(\text{myproc})\) must move to each other processor, costs at least \((p-1)\)\text{cost of sending } n*(n/p) \text{ words}

- Parallel Efficiency = \(2*n^3 / (p * \text{Total Time})\)
  \[= 1/(1 + \alpha * p^2/(2*n^3) + \beta * p/(2*n) )\]
  \[= 1/ (1 + O(p/n))\]
  \[= 1 - O(p/n)\]

- Grows to 1 as n/p increases (or \(\alpha\) and \(\beta\) shrink)
MatMul with 2D Layout

• Consider processors in 2D grid (physical or logical)
• Processors can communicate with 4 nearest neighbors
  • Alternative pattern: broadcast along rows and columns

\[
\begin{array}{ccc}
  p(0,0) & p(0,1) & p(0,2) \\
  p(1,0) & p(1,1) & p(1,2) \\
  p(2,0) & p(2,1) & p(2,2) \\
\end{array}
\quad = \quad
\begin{array}{ccc}
  p(0,0) & p(0,1) & p(0,2) \\
  p(1,0) & p(1,1) & p(1,2) \\
  p(2,0) & p(2,1) & p(2,2) \\
\end{array}
\quad \ast
\begin{array}{ccc}
  p(0,0) & p(0,1) & p(0,2) \\
  p(1,0) & p(1,1) & p(1,2) \\
  p(2,0) & p(2,1) & p(2,2) \\
\end{array}
\]

• Assume p is square s x s grid
Cannon’s Algorithm: 2-D merry-go-round

... $C(i,j) = C(i,j) + \sum_{k} A(i,k)B(k,j)$

... assume $s = \sqrt{p}$ is an integer

forall $i=0$ to $s-1$ ... “skew” A

left-circular-shift row $i$ of A by $i$

... so that $A(i,j)$ overwritten by $A(i,(j+i)\text{mod } s)$

forall $i=0$ to $s-1$ ... “skew” B

up-circular-shift B column $i$ of B by $i$

... so that $B(i,j)$ overwritten by $B((i+j)\text{mod } s), j)$

for $k=0$ to $s-1$ ... sequential

for all $i=0$ to $s-1$ and $j=0$ to $s-1$ ... all processors in parallel

$C(i,j) = C(i,j) + A(i,j)B(i,j)$

left-circular-shift each row of A by 1

up-circular-shift each row of B by 1
Cannon’s Matrix Multiplication

Cannon’s Matrix Multiplication Algorithm

Initial A, B

A, B after skewing

A, B after shift k=1

A, B after shift k=2

\[ C(1,2) = A(1,0) \times B(0,2) + A(1,1) \times B(1,2) + A(1,2) \times B(2,2) \]
**Initial Step to Skew Matrices in Cannon**

- **Initial blocked input**

  \[
  \begin{array}{ccc}
  A(0,0) & A(0,1) & A(0,2) \\
  A(1,0) & A(1,1) & A(1,2) \\
  A(2,0) & A(2,1) & A(2,2) \\
  \end{array}
  \quad
  \begin{array}{ccc}
  B(0,0) & B(0,1) & B(0,2) \\
  B(1,0) & B(1,1) & B(1,2) \\
  B(2,0) & B(2,1) & B(2,2) \\
  \end{array}
  \]

- **After skewing before initial block multiplies**

  \[
  \begin{array}{ccc}
  A(0,0) & A(0,1) & A(0,2) \\
  A(1,1) & A(1,2) & A(1,0) \\
  A(2,2) & A(2,0) & A(2,1) \\
  \end{array}
  \quad
  \begin{array}{ccc}
  B(0,0) & B(0,1) & B(0,2) \\
  B(1,0) & B(2,1) & B(0,2) \\
  B(2,0) & B(0,1) & B(1,2) \\
  \end{array}
  \]
**Skewing Steps in Cannon**

- **First step**

<table>
<thead>
<tr>
<th>A(0,0)</th>
<th>A(0,1)</th>
<th>A(0,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1,1)</td>
<td>A(1,2)</td>
<td>A(1,0)</td>
</tr>
<tr>
<td>A(2,2)</td>
<td>A(2,0)</td>
<td>A(2,1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B(0,0)</th>
<th>B(1,1)</th>
<th>B(2,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1,0)</td>
<td>B(2,1)</td>
<td>B(0,2)</td>
</tr>
<tr>
<td>B(2,0)</td>
<td>B(0,1)</td>
<td>B(1,2)</td>
</tr>
</tbody>
</table>

- **Second**

<table>
<thead>
<tr>
<th>A(0,1)</th>
<th>A(0,2)</th>
<th>A(0,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1,2)</td>
<td>A(1,0)</td>
<td>A(1,1)</td>
</tr>
<tr>
<td>A(2,0)</td>
<td>A(2,1)</td>
<td>A(2,2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B(0,1)</th>
<th>B(2,1)</th>
<th>B(0,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(2,0)</td>
<td>B(0,1)</td>
<td>B(1,2)</td>
</tr>
<tr>
<td>B(0,0)</td>
<td>B(1,1)</td>
<td>B(2,2)</td>
</tr>
</tbody>
</table>

- **Third**

<table>
<thead>
<tr>
<th>A(0,2)</th>
<th>A(0,0)</th>
<th>A(0,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1,0)</td>
<td>A(1,1)</td>
<td>A(1,2)</td>
</tr>
<tr>
<td>A(2,1)</td>
<td>A(2,2)</td>
<td>A(2,0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B(2,0)</th>
<th>B(0,1)</th>
<th>B(1,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(0,0)</td>
<td>B(1,1)</td>
<td>B(2,2)</td>
</tr>
<tr>
<td>B(1,0)</td>
<td>B(2,1)</td>
<td>B(0,2)</td>
</tr>
</tbody>
</table>
Cost of Cannon’s Algorithm

forall  i=0 to s-1 ... recall s = sqrt(p)
  left-circular-shift row i of A by i ... cost = s*(\(\alpha + \beta \cdot n^2/p\))
forall  i=0 to s-1
  up-circular-shift B column i of B by i ... cost = s*(\(\alpha + \beta \cdot n^2/p\))
for k=0 to s-1
  forall  i=0 to s-1 and j=0 to s-1
    \(C(i,j) = C(i,j) + A(i,j) \cdot B(i,j)\) ... cost = 2*(n/s)^3 = 2*n^3/p^{3/2}
  left-circular-shift each row of A by 1 ... cost = \(\alpha + \beta \cdot n^2/p\)
  up-circular-shift each row of B by 1 ... cost = \(\alpha + \beta \cdot n^2/p\)

° Total Time = \(2 \cdot n^3/p + 4 \cdot s \cdot \alpha + 4 \cdot \beta \cdot n^2/s\)
° Parallel Efficiency = \(2 \cdot n^3 / (p \cdot \text{Total Time})\)
  = \(1 / (1 + \alpha \cdot 2 \cdot (s/n)^3 + \beta \cdot 2 \cdot (s/n))\)
  = \(1 - O(\sqrt{p}/n)\)
° Grows to 1 as \(n/s = n/sqrt(p) = sqrt(\text{data per processor})\) grows
° Better than 1D layout, which had Efficiency = \(1 - O(p/n)\)
Extra Slides:

SUMMA parallel matrix multiplication algorithm
SUMMA Algorithm

• SUMMA = Scalable Universal Matrix Multiply

• Slightly less efficient than Cannon
  … but simpler and easier to generalize

• Presentation from van de Geijn and Watts
  • www.netlib.org/lapack/lawns/lawn96.ps
  • Similar ideas appeared many times

• Used in practice in PBLAS = Parallel BLAS
  • www.netlib.org/lapack/lawns/lawn100.ps
\[ C(I,J) = C(I,J) + \sum_k A(I,k) \cdot B(k,J) \]

- \( I, J \) represent all rows, columns owned by a processor
- \( k \) is a single row or column
  - or a block of \( b \) rows or columns
- Assume a \( pr \) by \( pc \) processor grid (\( pr = pc = 4 \) above)
  - Need not be square
For k=0 to n-1  … or n/b-1 where b is the block size

 … = # cols in A(I,k) and # rows in B(k,J)

for all I = 1 to p_r  … in parallel

owner of A(I,k) broadcasts it to whole processor row

for all J = 1 to p_c  … in parallel

owner of B(k,J) broadcasts it to whole processor column

Receive A(I,k) into Acol

Receive B(k,J) into Brow

C( myproc , myproc ) = C( myproc , myproc ) + Acol * Brow
SUMMA performance

- To simplify analysis only, assume $s = \sqrt{p}$

For $k=0$ to $n/b-1$
  
  for all $l = 1$ to $s$  
    ... $s = \sqrt{p}$
    
    owner of $A(l,k)$ broadcasts it to whole processor row
    
    ... time = $\log s \times (\alpha + \beta \times b/n/s)$, using a tree
  
  for all $J = 1$ to $s$
    
    owner of $B(k,J)$ broadcasts it to whole processor column
    
    ... time = $\log s \times (\alpha + \beta \times b/n/s)$, using a tree

Receive $A(l,k)$ into $A_{col}$
Receive $B(k,J)$ into $B_{row}$

$C(\text{myproc}, \text{myproc}) = C(\text{myproc}, \text{myproc}) + A_{col} \times B_{row}$

... time = $2 \times (n/s)^2 \times b$

- Total time = $2 \times n^3/p + \alpha \times \log p \times n/b + \beta \times \log p \times n^2 /s$
SUMMA performance

- Total time = \( \frac{2n^3}{p} + \alpha \log p \frac{n}{b} + \beta \log p \frac{n^2}{s} \)
- Parallel Efficiency =
  \[
  \frac{1}{1 + \alpha \log p \frac{p}{(2\beta n^2)} + \beta \log p \frac{s}{(2n)}}
  \]
- ~Same \( \beta \) term as Cannon, except for \( \log p \) factor
  
  \( \alpha \log p \) grows slowly so this is ok
- Latency (\( \alpha \)) term can be larger, depending on \( b \)
  
  When \( b = 1 \), get \( \alpha \log p \times n \)
  
  As \( b \) grows to \( n/s \), term shrinks to
  
  \( \alpha \log p \times s \) (\( \log p \) times Cannon)
- Temporary storage grows like \( 2b \times n/s \)
- Can change \( b \) to tradeoff latency cost with memory