1. Optimization Considerations

- Two ways to optimize:
  1. Change algorithm
  2. Performance tuning (this class)
- 90/10 rule
  - Use profiling to find kernels
- Can kernels be recast to use already optimized kernel like GEMM?
- “Optimized for performance” & “portable/maintainable” are antonyms
- Purely a priori optimization is pipe dream
  - Hardware component interaction unpredictable
  - Compiler, OS, ISA get in way of hardware
- The more you hand-optimize, the less the compiler can do for you
- Will rewrite kernel for each new arch
  - Use assembly sparingly
  - Try to find other users to reuse kernel

2. Two Types of Optimization

- Memory/cache optimization:
  - Theoretical peak: (bus width) * (bus speed)
    - PII: (32 bits) * (66 Mhz) = 264 MB/s = 33 MW/s
    - Athlon: (64 bits) * (200 Mhz) = 1600 MB/s = 200 MW/s
    - Power3: (128 bits) * (100 Mhz) = 1600 MB/s = 200 MW/s
  - Control data compete for these bits
- Computational optimization:
  - Theoretical peak: (# fpus) * (flops/cycle) * Mhz
    - PII: (1 fpu) * (1 flop/cycle) * (450 Mhz) = 450 MFLOP
    - Athlon: (2 fpu) * (1 flop/cycle) * (600 Mhz) = 1200 MFLOP
    - Power3: (2 fpu) * (2 flops/cycle) * (375 Mhz) = 1500 MFLOP
- Memory at least order of magnitude slower
  - Optimize for memory first and/or
  - Optimize computation in-cache

3. Transformations for Performance Optimization

Computational:

1. Loop unrolling
2. C source hints
3. Strength reduction
4. Scalar expansion
5. Superscalar scheduling
6. Software pipelining (loop skewing)
7. Loop peeling
8. Branch optimization/Frequent path coalescing
9. Instruction selection/alignment
10. Unroll & Jam wt. reg blocking

Memory:

1. Register blocking
2. Fetch scheduling
3. Software Prefetch
4. Block Fetch
5. Efficient array indexing
6. Blocking
7. Data copy

4. Loop Unrolling

- Duplicate loop body N_u times, ensure cleanup
- Reduces loop overhead, enables many other optimizations
- If possible, avoid repetitive index & ptr updates
- Do-while most efficient loop

```
for (i=0; i < N; i++)
   dot += X[i] * Y[i];
              ⇓
for (i=0; i < N; i += 4) {
   dot += X[i] * Y[i];
   dot += X[i+1] * Y[i+1];
   dot += X[i+2] * Y[i+2];
   dot += X[i+3] * Y[i+3];
}
for (i -= 4; i < N; i++)
   dot += X[i] * Y[i];
```

```
double *stX = X + (N/4)*4,
       *stX2 = X + N;
do {
   dot += *X * *Y;
   dot += X[1] * Y[1];
   dot += X[2] * Y[2];
   dot += X[3] * Y[3];
   X += 4; Y += 4;
} while (X != stX);
while (X != stX2)
   dot += *X++ * *Y++;
```

5. C Source Hints

- Use "const" if variable not changing
- Use "register" for critical vars
- Use `*X` over `X[0]`
- Perform manual strength reduction
- Take tmp arrays from stack to guarantee alignment & avoid stack overflow
- Eliminate unused loop vars
  - ptr controlled loops
- Unroll loop for reduced loop overhead
- Increment ptrs only at end of loop

```
double precision ddot(const int N,
   const double *X, const double *Y)
{
   const int N4 = (N>>2)<<2;
   const double * const stX = X + N4
   const double * const stX2 = X + N
   register double dot=0.0;
   double *stX = X + (N/4)*4;
   do {
      dot += *X * *Y;
      dot += X[1] * Y[1];
      dot += X[2] * Y[2];
      dot += X[3] * Y[3];
      X += 4; Y += 4;
   } while (X != stX);
   while (X != stX2) dot += *X++ * *
```

6. Strength Reduction

- Replace costly calculation with one or more cheap calculation(s)

- Replace positive % by div+mult
  - `i%3` ⇒ `(i-(i/3)*3)`
- Replace mult/div by pow-of-2 by shifts
  - `i%8` ⇒ `(i-((i>>3)<<3))`
- Replace mult by several adds and/or shifts:
  - `i = 9*N` ⇒ `i = (N<<3)+N`
- Replace $>$ or $<$ with `==` or `!=` when legal
  - `if (i++<N)` ⇒ `if (i++!=N)`

- If safe in fp, multiply by inverse rather than divide:
  - `t0/1000000.0` ⇒ `t0*1.0e-6`
- Combine SR with common subexpression elimination:

```
c += A[3*i] * B[3*i+2];
```

⇓

```
k = i + i + i;
c += A[k] * B[k+2];
```

7. Scalar expansion

- Replace single scalar with multiple variables to avoid dependency problems
  - For successive fp calcs, make length of FPU pipe

```
sum = 0;
do
{
   sum += *X;
   sum += X[1];
   sum += X[2];
   sum += X[3];
   X += 4;
}
while (X != stX);
```

```
sum1 = sum2 = sum3 = sum = 0.0;
do
{
   sum += *X;
   sum1 += X[1];
   sum2 += X[2];
   sum3 += X[3];
   X += 4;
}
while (X != stX);
sum += sum1 + sum2 + sum3;
```

8. Superscalar scheduling

- Schedule needed computations so that instructions that use independent PEs are close together
- Therefore, if you can avoid it, don't bunch up all FP inst, followed by all int instructions (eg. loop inst), but spread them out.
- For instance, if you have two integer units, an FP adder and an FP multiplier, you could do a pointer update, add the integer loop counter, do a FP add, and an FP multiply in the same clock cycle
- If you've got multiple units, unroll loop until that many independent inst exposed, and reorder as necessary
- Make sure all instructions are independent; this won't work:

```
m0 = a0 * b0;
c0 += m0;
```

$\Rightarrow$ Need *software pipelining* to make independent

9. Software Pipelining

- Intermix loop iterations so that iteration gets dependent information from previous iteration
  1. Unroll loop pipelen iterations
  2. Issue pipelen of starting dep inst before start
  3. Stop iter early to drain pipe with pipelen of 2nd of dep inst

```
for (i=0; i < N; i += 4)
{
   dot += X[0] * Y[0];
   dot += X[1] * Y[1];
   dot += X[2] * Y[2];
   dot += X[3] * Y[3];
   X += 4; Y += 4;
}
```

```
m0 = *X * *Y;      m1 = X[1] * Y[1];
m2 = X[2] * Y[2]; m3 = X[3] * Y[3];
X += 4; Y += 4;
for (i=4; i < N; i += 4)
{
   dot  += m0;  m0 = X[0] * Y[0];
   dot1 += m1;  m1 = X[1] * Y[1];
   dot2 += m2;  m2 = X[2] * Y[2];
   dot3 += m0   m3 = X[3] * Y[3];
   X += 4; Y += 4;
}
dot  += m0; dot1 += m1; dot2 += m2; dot3 += m3
```

10. Loop Peeling

- (Conditionally) perform X iterations of loop before entry
- Often done to enforce needed condition in loop
  - Guarantee loop multiple of factor
  - Guarantee particular alignment
- Here's a DP loop peeled to guarantee 16 byte alignment:

```
for (i=0; i < N; i++)
   sum += X[i];
```

```
int i=0;
if (((unsigned long)X) & 0x7)
{
   sum = *X;
   i = 1
}
else sum = 0.0;
for (; i < N; i++)
   sum += X[i];
```

11. Branch optimization/Frequent path coalescing

- Most archs faster when branch is not taken, so put most common case as fall through
- Before execution, most branch predictors predict:
  1. "taken" for backward jumps (optimizes loops, makes do-while best)
  2. "not taken" for forward jumps (opt for loop exit)

$\Rightarrow$ Organize unlikely jump forward in code, likely fall through or back

- For best performance, make most frequently executed path all fall through (jumps only for infrequent cases)

- Normal vector max:

```
   for (max=0.0,i=0; i < N; i++)
      if (max < X[i]) max = X[i]
```

- Fall-thru most common case:

```
   for (max=0.0,i=0; i < N; i++)
      if (max >= X[i]) continue;
      else max = X[i];
   }
```

- With frequent path coalescing:

```
   for (max=0.0,i=0; i < N; i++)
      if (max < X[i]) goto NEWMA
EOL:;
   }
NEWMAX:
   max = X[i];
   goto EOL;
```

12. Instruction optimizations

- Most instruction optimizations must be done at assembly level, two important ones are:

Instruction Selection

- Most times, there are many different ways of accomplishing same operation
- Find the best one for current context:
  1. Is one method cheaper (strength red)?
  2. Can we do two computations at once?
  3. Does one require less inst memory?

Instruction Alignment

- Inst kept in caches like mem, so frequently jumped to locations may benefit from explicit aligning to inst cache-line
- Some machines have alignment for inst fetch, or limited size windows, so can gain by doing SSS in blocks and/or explicitly sceduling nops between instructions
  - Extreme ex. Athlon: non-aligned inst get 70% peak, but aligned gets 90%

13. Unroll & Jam wt. Register Blocking

- Reg blk: scalar replacement + register asg

14. Memory Optimization Basics

Do:

- Start on cache line boundary
- Use entire line
- Make array stride (lda) a mult of cache line size
- Issue as many nonblocking fetches as cache supports
- If you have reuse:
  - Block for cache size
  - Copy to contiguous storage

Don't

- Make array stride power of 2
- Access strided memory
  - If you must, can you still use full cache line?
- Access more than TLB separate memory locations

15. Fetch Scheduling – Using the PEs

- Schedule to initiate as many nonblocking fetches as possible
- Schedule with enough unique fetch streams to drive all prefetch units
- DDOT has no vector reuse, so no blocking, but can still be opt as above:

4 fetches (32 byte CL):

```
for (i=0; i < N; i += 8) {
   dot0 += x[0] * y[0];
   dot1 += x[4] * y[4];
   dot0 += x[1] * y[1];
   dot1 += x[5] * y[5];
   dot0 += x[2] * y[2];
   dot1 += x[6] * y[6];
   dot0 += x[3] * y[3];
   dot1 += x[7] * y[7];
   x += 8; y += 8;
}
```

4 prefetch units

```
n2 = N>>1;
xx = X + n2; yy = Y + n2;
for (i=0; i < n2; i++) {
   dot0 += x[i] * y[i];
   dot1 += xx[i] * yy[i];
}
```

16. Fetch Scheduling – Software Pipelining/SSS

- Fetches may be software pipelined against use
- Results in superscalar scheduling for concurrent FPU & fetch unit usage

```
x0 = x[0]; y0 = y[0]; x1 = x[1]; y1 = y[1]; x2 = x[2]; y2 = y[2];
x3 = x[3]; y3 = y[3]; x +=4; y += 4;
for (i=4; i < N; i += 4)
{
   dot0 += x0 * y0;   x0 = x[0]; y0 = y[0];
   dot1 += x1 * y1;   x1 = x[1]; y1 = y[1];
   dot2 += x2 * y2;   x2 = x[2]; y2 = y[2];
   dot3 += y3 * y3;   x3 = x[3]; y3 = y[3];
}
dot0 += x0 * y0; dot1 += x1 * y1;
dot2 += x2 * y2; dot3 += y3 * y3;
```

17. Fetch Pipe=4, Mul/add Pipe=2

```
x0 = *X;   y0 = *Y0;  x1 = X[1]; y1 = Y[1];
x2 = X[2]; y2 = Y[2]; x3 = X[3]; y3 = Y[3];
m0 = x0 * y0; x0 = X[4]; y0 = Y[4];
m1 = x1 * y1; x1 = X[5]; y1 = Y[5];

for (i=8; i < N; i += 4)
{
   dot0 += m0;  m0 = x2 * y2;  x2 = X[6]; y2 = Y[6];
   dot1 += m1;  m1 = x3 * y3;  x3 = X[7]; y3 = Y[7];
   dot0 += m0;  m0 = x0 * y0;  x0 = X[8]; y0 = Y[8];
   dot1 += m1;  m1 = x1 * y1;  x1 = X[9]; y1 = Y[9];
      X += 4;  Y += 4;
}
dot0 += m0;  m0 = x2 * y2;  x2 = X[6]; y2 = Y[6];
dot1 += m1;  m1 = x3 * y3;  x3 = X[7]; y3 = Y[7];
dot0 += m0;  m0 = x0 * y0;
dot1 += m1;  m1 = x1 * y1;

dot0 += m0;  m0 = x2 * y2;
dot1 += m1;  m1 = x3 * y3;
dot0 += m0;
dot1 += m1;
```

18. Software Prefetch

- Fetch items to be used later (latency hiding, no bandwidth change)
- No standard C API, so use inline assembly or compiler intrinsic
- Software can beat hardware because you can tune:
  - Which operand(s) to prefetch / prefetch pattern
  - Distance ahead to prefetch for each operand
- Unroll loop to multiple of CL to avoid redundant PF
- Different archs demand different scheduling

```
#ifdef OPTERON
   #define XDIST 256
#elif defined(P4E)
   #define XDIST 512
#else
   #define XDIST 768
#endif
#define YDIST XDIST
```

```
for (i=0; i < N; i += 4) {
   MY_PREF(X+XDIST);
   MY_PREF(Y+YDIST);
   dot0 += X[0] * Y[0];
   dot1 += X[1] * Y[1];
   dot2 += X[2] * Y[2];
   dot3 += X[3] * Y[3];
}
```

19. Block Fetch

- After peeling for CL alignment, loop over blocks and perform operation in two (three) phases:
  1. Load data to cache, driving bus at maximal rate
     - Access only one elt per CL
     - Loop in reverse order, but access in order
  2. Perform computation (optd for in-L1)
  3. Write data out in block
- `http://cdrom.amd.com/devconn/events/AMD_block_prefetch_paper.pdf`

```
int nb = 256, *ia;
for (b=0; b < N; b += nb) {
   /* burst load using ints */
   for (ia=X,i=nb>>1;i>=0;i-=16){
      j += ia[i]; k += ia[i+8];
   }
   for (ia=Y,i=nb>>1;i>=0; i-=16)
      j += ia[i]; k += ia[i+8];
   }

   /* computation on in-L1 data *
   for (i=0; i < nb; i++)
      ddot += X[i] * Y[i];
}
```

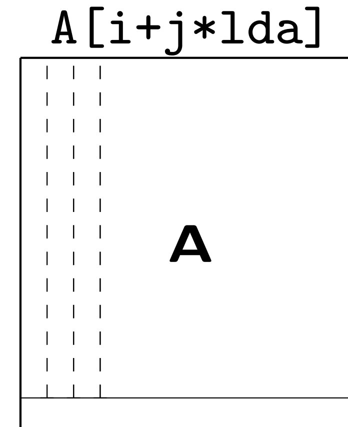
20. Efficient array indexing

- C 2-D arrays are useless
- `**p` can double cost of strided access, and make submatrixing horribly expensive

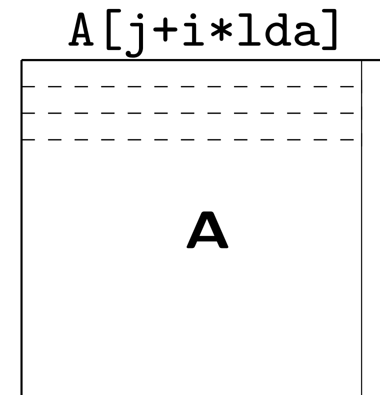
⇒ Utilize `*p` and index yourself

- Can use row or column-major ordering
- Can use ptr or int arithmetic
- Pass in leading dim (`lda`)
  1. Gives stride to next non-contiguous row/col item
  2. Allows for free submatrixing
  3. Allows use of cache-friendly strides

Column-major storage:



Row-major storage:



21. Efficient array indexing

```
for (j=0; j < N; j++)
{
   for (i=0; i < M; i++)
      A[i+j*lda] = my_rand();
}
```

⇓

```
for (j=N; j; j--, A += lda)
{
   for (i=0; i < M; i++)
      A[i] = my_rand();
}
```

```
lda -= M;
for (j=N; j; j--, A += lda)
{
   for (i=M>>2; i; i--)
   {
      *A = my_rand();
      A[1] = my_rand();
      A[2] = my_rand();
      A[3] = my_rand();
      A += 4;
   }
}
```

22. Blocking

AKA: *tiling*

Basic idea

When a data set is too large to fit into a given level of cache, and there is data reuse in the algorithm, the data set is subdivided into blocks of data which *will* fit into the given cache level, and then these cache-resident subsections are reused across multiple computations.

Requirements for blocking

1. Data set of blocked operation must exceed cache
   - Otherwise, op is cache-contained without blocking
2. The number of accesses on a given operand required by algorithm must exceed the number of elements in that operand
   - Otherwise, no data reuse possible
3. It must be legal to reorder computations
   - Blocking is a computational reordering

23. Unblocked Matrix Multiply

- $C \leftarrow AB + C$
- Loops may be arbitrarily re-ordered
  - *Requirement 3*
- $2N^3$ FLOPS
- $3N^3$ reads
- N^3 writes

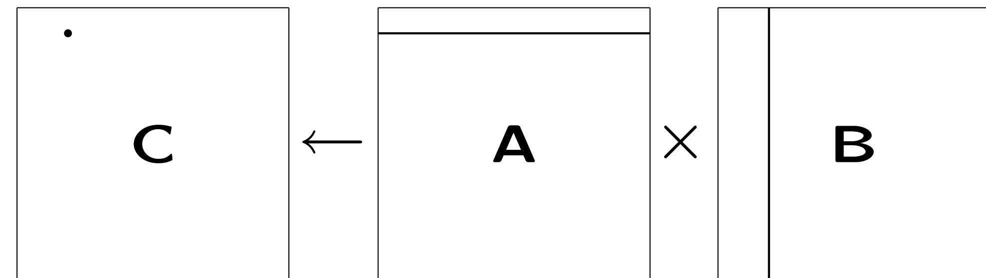
- $3N^2$ minimum reads
- N^2 minimum writes

$\Rightarrow$ Have opportunity for reuse
  - *Requirement 2*
  - $4N^3$ to $4N^2$ blocking win
- If $N^2 + N > C_S$ (C_S = cache size), working set exceeds cache
  - *Requirement 1*
  - $2N^2$ of C, N^3 of A, N^2 or N^3 of B

Loop Nest

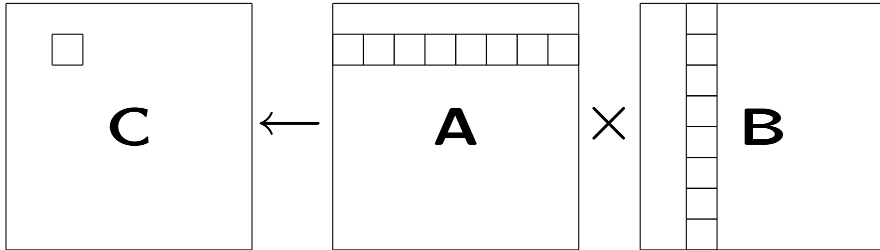
```
DO I = 1, N
   DO J = 1, N
      DO K = 1, N
         C(I,J) = C(I,J) +
                  A(I,K) * B(K,J)
      END DO
   END DO
END DO
```

Operand access in calc. one elt of C



24. L1-Blocked Matrix Multiply

Operand access in calc. one blk of C



Loop nest

```
DO J = 1, N, NB
   DO I = 1, N, NB
      DO K = 1, N, NB
         DO IB = 0, NB-1
            DO JB = 0, NB-1
               DO KB = 0, NB-1
                  C(I+IB,J+JB) = C(I+IB,J+JB) +
                                 A(I+IB,K+KB) *
                                 B(K+KB,J+JB)
               END DO
            END DO
         END DO
      END DO
   END DO
END DO
```

- If cache is *not* write through
  - $N_B \approx \sqrt{\frac{L_1}{3}}$
- Otherwise, above, or
  - $N_B \approx \sqrt{L_1}$
- FLOPS still:
  - $\sum_{j=1}^{N/N_B} \sum_{i=1}^{N/N_B} \sum_{k=1}^{N/N_B} 2{N_B}^3 = (\frac{N}{N_B})^3(2N_B^3) = 2N^3$
- Total A and B reads now:
  - $\sum_{j=1}^{N/N_B} \sum_{i=1}^{N/N_B} \sum_{k=1}^{N/N_B} {N_B}^2 = (\frac{N}{N_B})^3(2{N_B}^2) = 2\frac{N^3}{N_B}$
- Write of C is:
  - If all three blocks retained in cache:
    - $2N^2$ (optimal)
  - If only one:
    - $2\frac{N^3}{N_B}$

25. Recursive Blocking

- AKA: cache-oblivious blocking, divide & conquer
- Divide largest dimension at each level of recursion

strengths:

- Requires no knowledge of machine
- Automatically blocks for all levels of cache
- Easy to implement
- Beautiful

weaknesses:

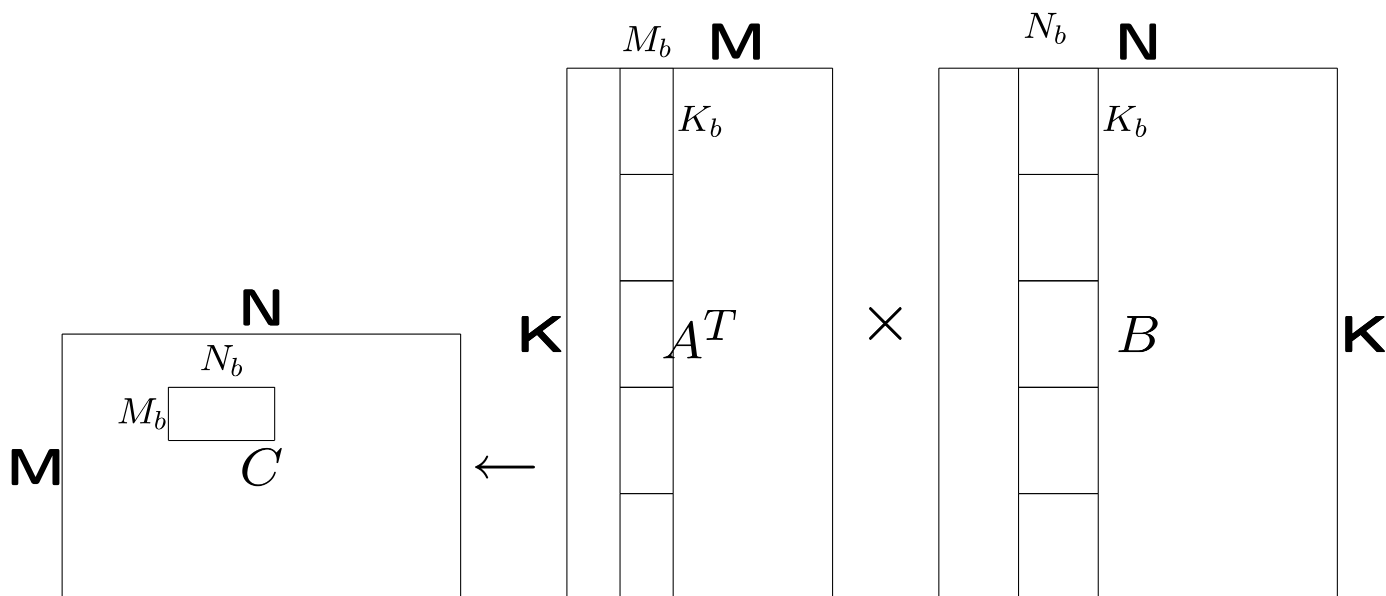
- Will rarely fully utilize cache
- Could miss a level (CS X $\approx$ CS X+1)
- Recurring to 1 will generate too much overhead
  - $\rightarrow$ Will want to stop early enough to use optimized kernel

$\Rightarrow$ Often best to recur to N_B (blocking for Level 1)

- Keep one part of recursive division multiple of N_B

26. Non-square Static Blocking

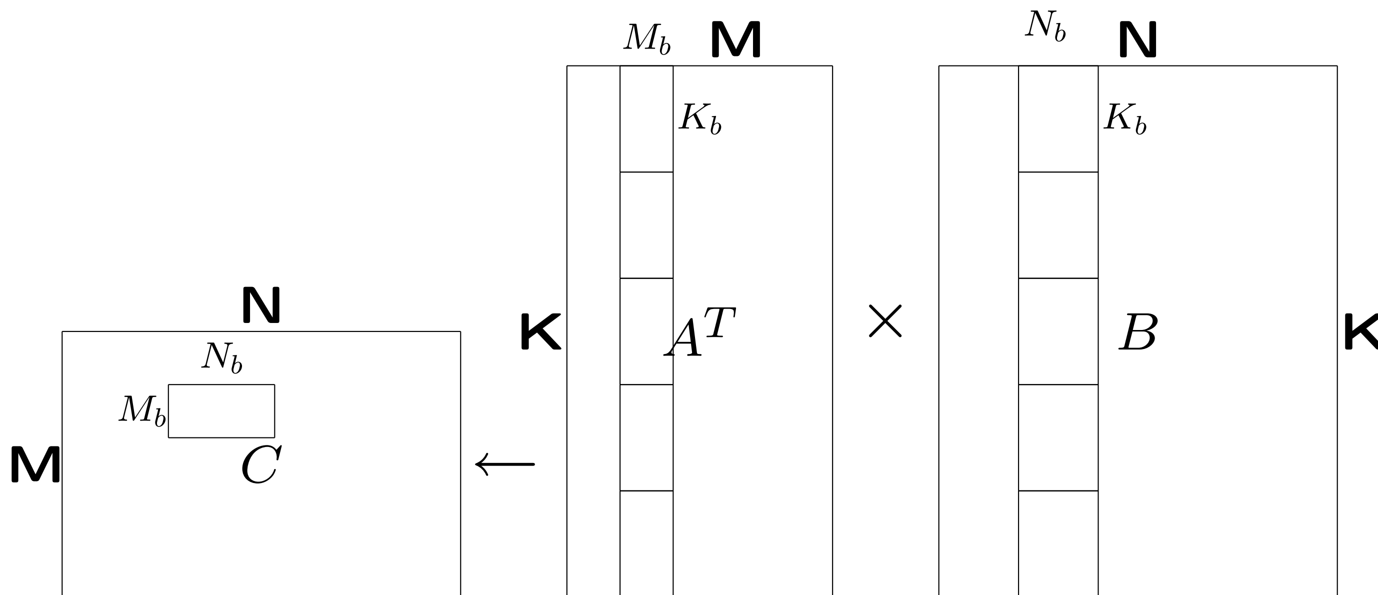
- If all 3 blks in-cache, square best shape for reuse
- If only one blk contained, can make N_b (M_b) as large as you like, assuming NMK (MNK) loop order
  - increasing N_b (M_b) decreases L2 reuse!



- Blocking affects mem access, **and** kernel shape:
  - Large K_b allows kernel better amort. of N^2 C and β costs (import SSE!)
- Often want to make :
  - $K_b > M_b$
  - $N_b > M_b$

27. Two-level Static Blocking

- Partitioning of all dimensions not equal:
  - Cutting K (non-L1) results in mult access C, other dims in mult access of A or B:
    - C is both read and written
    - Writes cause more bus contention
- So, simple blocking maximizes K L2-part by reusing panel of B across all panels of A



- Block for L2 by cutting K:
  - $2M_bK + N_bK + 2M_bN_b \leq L2$
  - $K_p \leq \frac{L2 - 2M_bN_b}{2M_b + N_b}$
  - Don't use all of L2
  - Make K_p mult. of K_b

28. Array-specific Spatial Optimizations

- Access contiguously stored dimension
  - Column for Fortran, row for ANSI C
- Traverse memory in smallest-to-largest order
  - Cache line fill
  - Hardware prefetch
- Operate on multiple columns (rows) to support multiple prefetch units
- Choose row (column) stride that is not power of two to help reduce line conflicts
- To better use TLBs in blocked algorithms:
  - Set $N_B * N_M \leq N_{TLB}$
    - N_B: blocking factor
    - N_M: number of matrices
    - N_{TLB}: number of TLB entries
  - Copy blocks to block-major storage
- If operation not cache blocked, apply TLB blocking
  - I.e., operate on page-contained data first