Optimization Techniques in SGI Scientific Libraries

John Baron

Mimi Celis

Performance Engineering and Math Libraries Group

Optimization Techniques in SCSL

Overview

- Ten-Step Tuning Process
- Let the Compiler Do the Work
  - Software Pipelining
  - Loop-level Optimizations
- Tune Cache Performance
  - Cache-blocking Examples in BLAS, LAPACK, FFTs
  - Copying to Reduce TLB Misses
- Summary

Ten-Step Tuning Process

Single-Processor Optimization

- Get the Right Answers
- Use Existing Tuned Code
  - -lm, -lfastm, -lscs[_mp] (SCSL)
- Find Out Where to Tune
  - Perfex, SpeedShop
- Let the Compiler Do the Work
- Tune Cache Performance

Ten-Step Tuning Process

Multi-processor Optimization

- Tune Single-Processor Performance
- Parallelize Code
- Identify and Solve Bottlenecks
- Fix False Sharing
- Tune for Data Placement

Reference: ***Origin 2000 and Onyx Performance and Optimization Tuning Guide***

http://techpubs.sgi.com/

Let the Compiler Do the Work

Recommended Compiler Flags

- -n32|-64 –mips4 –O3
- -OPT:IEEE_arithmetic=3
- -OPT:roundoff=3
- -OPT:alias=restrict

```
for (i = 0; i < n; i++) {
    y[i] += a * x[i];
}
```

- cdir$ ivdep or #pragma ivdep

```
#pragma ivdep
for (i = 0; i < n; i++) {
    y[index[i]] += a * x[index[i]];
}
```

- -TENV:X=[234] (use with caution)

Let the Compiler Do the Work

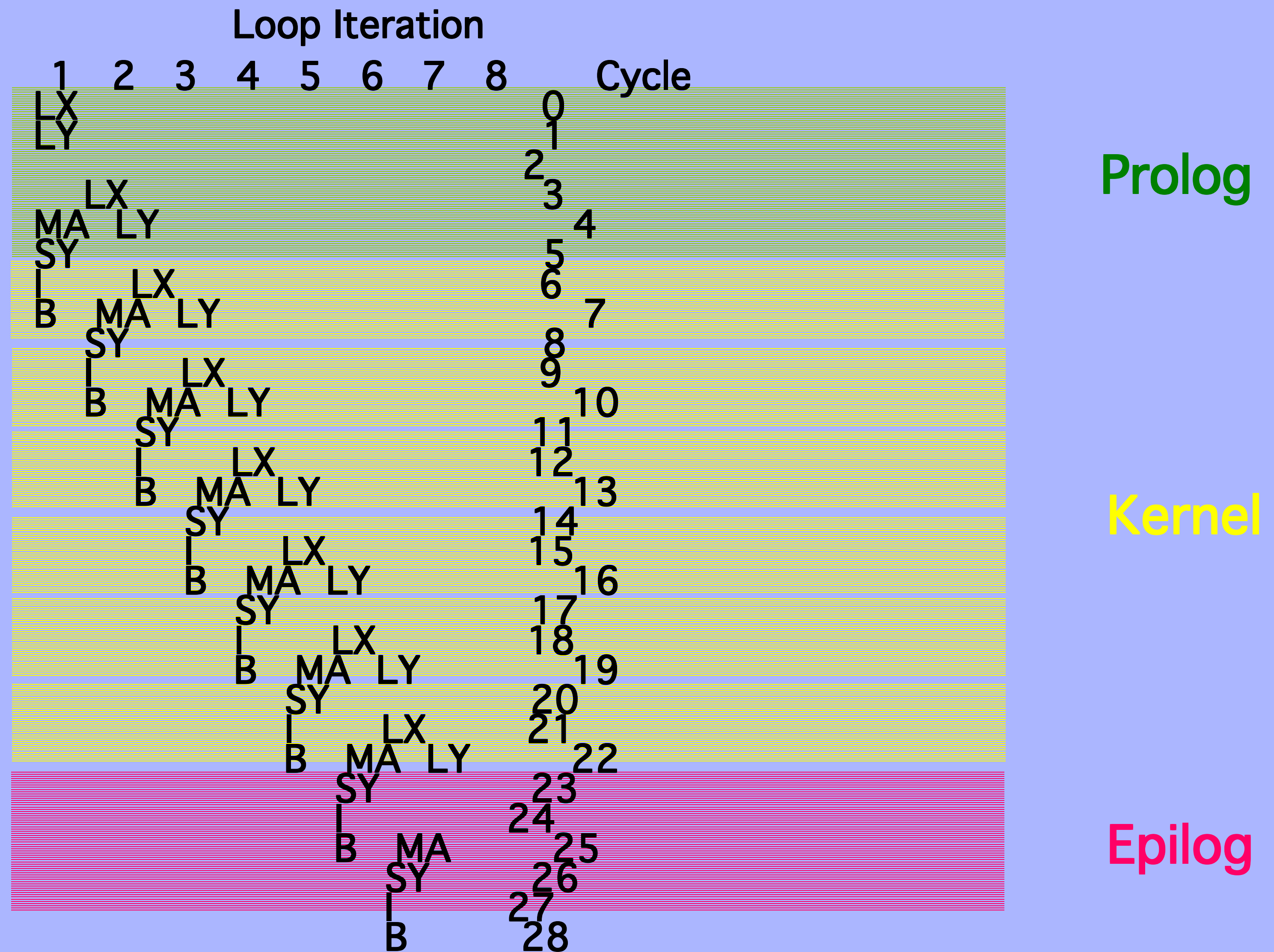
– **Loops have parallelism, e.g., BLAS DAXPY:**

```
do i = 1, n
     y(i) = y(i) + a * x(i)
  enddo
```

Loop parallelism	*4 superscalar slots/cycle*
2 loads, 1 store	*1 load or store, and/or*
1 multiply	*1 ALU1 instruction, and/or*
1 add	*1 ALU2 instruction, and/or*
2 address increments	*1 floating point add, and/or*
1 loop-end test	*1 floating point multiply*
1 branch	

– ***Software Pipelining* fills up the slots**

Let the Compiler Do the Work



Let the Compiler Do the Work

Software Pipelining

- No default SWP: use –O3
- Vectorizable loops well-suited to SWP
- SWP may be slower for small trip counts
- SWP cannot be done if:
  - Function calls in loop: use –IPA or –INLINE
  - Complicated conditions or branching
- SWP impeded by:
  - Loop body too large, register overflow: try splitting loop
  - Recurrences between iterations: use IVDEP directive

Let the Compiler Do the Work

Reading the "Love Notes"

f77 –O3 –LNO:prefetch=0 –S daxpy.f

```
#<loop> Loop body line 1, nesting depth: 1, estimated iterations: 17
#<loop> Unrolled 2 times
#<swps>
#<swps> Pipelined loop line 1 steady state
#<swps>
#<swps>    50 estimated iterations before pipelining
#<swps>     2 unrollings before pipelining
#<swps>     6 cycles per 2 iterations
#<swps>     4 flops         ( 33% of peak) (madds count as 2)
#<swps>     2 flops         ( 16% of peak) (madds count as 1)
#<swps>     2 madds          ( 33% of peak)
#<swps>     6 mem refs       (100% of peak)
#<swps>     3 integer ops   ( 25% of peak)
#<swps>    11 instructions ( 45% of peak)
#<swps>     1 short trip threshold
#<swps>     5 integer registers used.
#<swps>     9 float registers used.
#<swps>
```

Let the Compiler Do the Work

Loop-level Optimizations

- Inner Loop Unrolling
  - More useful work per iteration
  - Enabled automatically at –O2 and higher (e.g., DAXPY unrolled twice)
  - Controlled by –OPT:… flags (man opt)
- Outer Loop Unrolling
  - Exposes more computations
  - Can increase ratio of floating point operations to memory operations
  - Controlled by –LNO:… flags (man lno)
- Examine compiler intermediate representation with –flist or –clist flag

Let the Compiler Do the Work

Matrix-matrix Multiplication

– C = A B   A: M x K, B: K x N, C: M x N

```
do j = 1, N
   do i = 1, M
     tmp = 0.0
     do k = 1, P
       tmp = tmp + a(i,k) * b(k,j)
     end do
     c(i,j) = c(i,j) + tmp
   end do
 end do
```

– Inner loop: 2 loads, 1 multiply-add (assuming tmp in register)

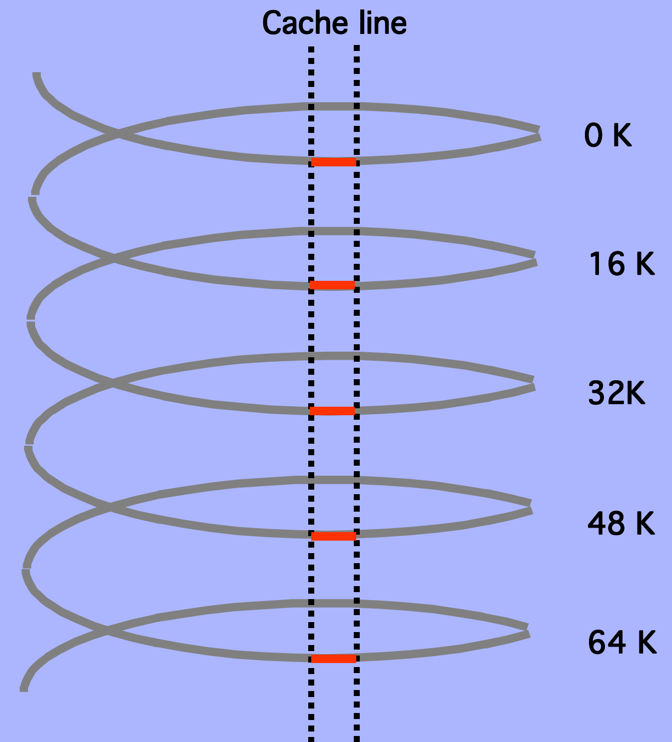
Let the Compiler Do the Work

```
do j = 1, N, 2
   do i = 1, M, 2
     tmp11 = tmp12 = tmp21 = tmp22 = 0.0
     do k = 1, P
       tmp11 = tmp11 + a(i,k)   * b(k,j)
       tmp12 = tmp12 + a(i+1,k) * b(k,j)
       tmp21 = tmp21 + a(i,k)   * b(k,j+1)
       tmp22 = tmp22 + a(i+1,k) * b(k,j+1)
     end do
     c(i,j)     = c(i,j)     + tmp11
     c(i+1,j)   = c(i+1,j)   + tmp12
     c(i,j+1)   = c(i,j+1)   + tmp21
     c(i+1,j+1) = c(i+1,j+1) + tmp22
   end do
 end do
```

- **Inner loop: 4 loads, 4 multiply-adds**
- **Limited by number of available floating point registers**

Tune Cache Performance

– R10000‰/R12000‰/R14000‰ cache organization
  - 2 level
    – 32 KB L1
    – Up to 8 MB L2
  - 2-way set associative
  - Accessed by “lines”
    – 32 B L1 cache lines
    – 128 B L2 cache lines
  - Lowest level fastest
    – L1 access 2 or 3 cycles (integer or f.p.)
    – L2 access ~12 cycles
    – Memory 90+ cycles (220+ ns)



Tune Cache Performance

General Cache Principles

- Stride-1 accesses for cache line reuse

```
do i = 1, n
      do j = 1, m
         a(i,j) = b(i,j)
      enddo
enddo
```

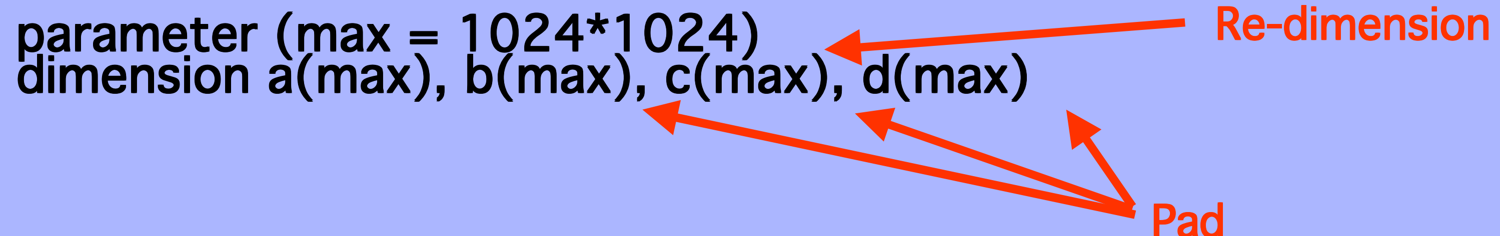
```
do j = 1, m
      do i = 1, n
         a(i,j) = b(i,j)
      enddo
enddo
```

- Group together data used at the same time

```
x(n),y(n),z(n),t(n)   →   q(4,n)
```

- Avoid power-of-2 sizes arrays, or pad

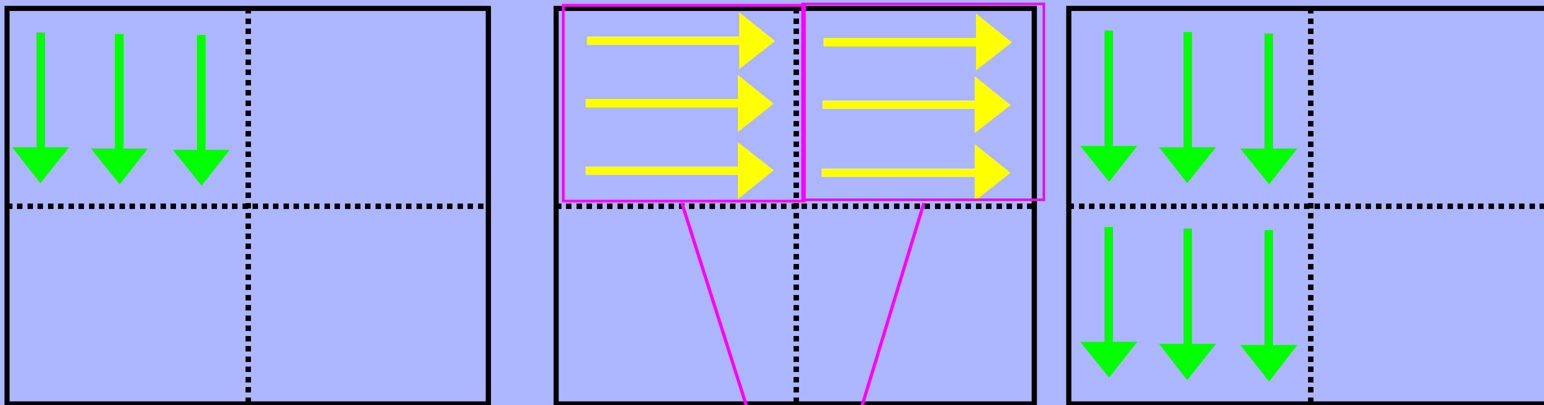
```
parameter (max = 1024*1024)
dimension a(max), b(max), c(max), d(max)
```

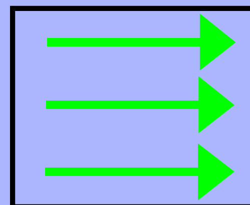


Tune Cache Performance

Matrix-matrix Multiplication Example

C = A x B





Copy to scratch space
of contiguous memory

Tune Cache Performance

Performance Comparison: 2000 x 2000 DGEMM

	Event Counter Name	Typical Counter Value	Time (sec)
No Blocking	0 Cycles..............................	79271934272	158.543869
	26 Secondary data cache misses..........	438427696	78.916985
	25 Primary data cache misses............	2025750544	44.566512
	23 TLB misses...........................	2098400	0.222430
Blocking, No Copy	0 Cycles..............................	27354302592	54.708605
	23 TLB misses...........................	244545472	25.921820
	25 Primary data cache misses............	1095659040	24.10449
	26 Secondary data cache misses..........	4140592	0.745307
Blocking, With Copy	0 Cycles..............................	9125033760	18.250068
	25 Primary data cache misses............	944214416	20.77271
	26 Secondary data cache misses..........	3788656	0.681958
	23 TLB misses...........................	196768	0.020857

Tune Cache Performance

LAPACK Example

- Solution of Dense System of Linear Equations:

$$Ax = LUx = Ly = b \Rightarrow y = L^{-1}b, x = U^{-1}y$$

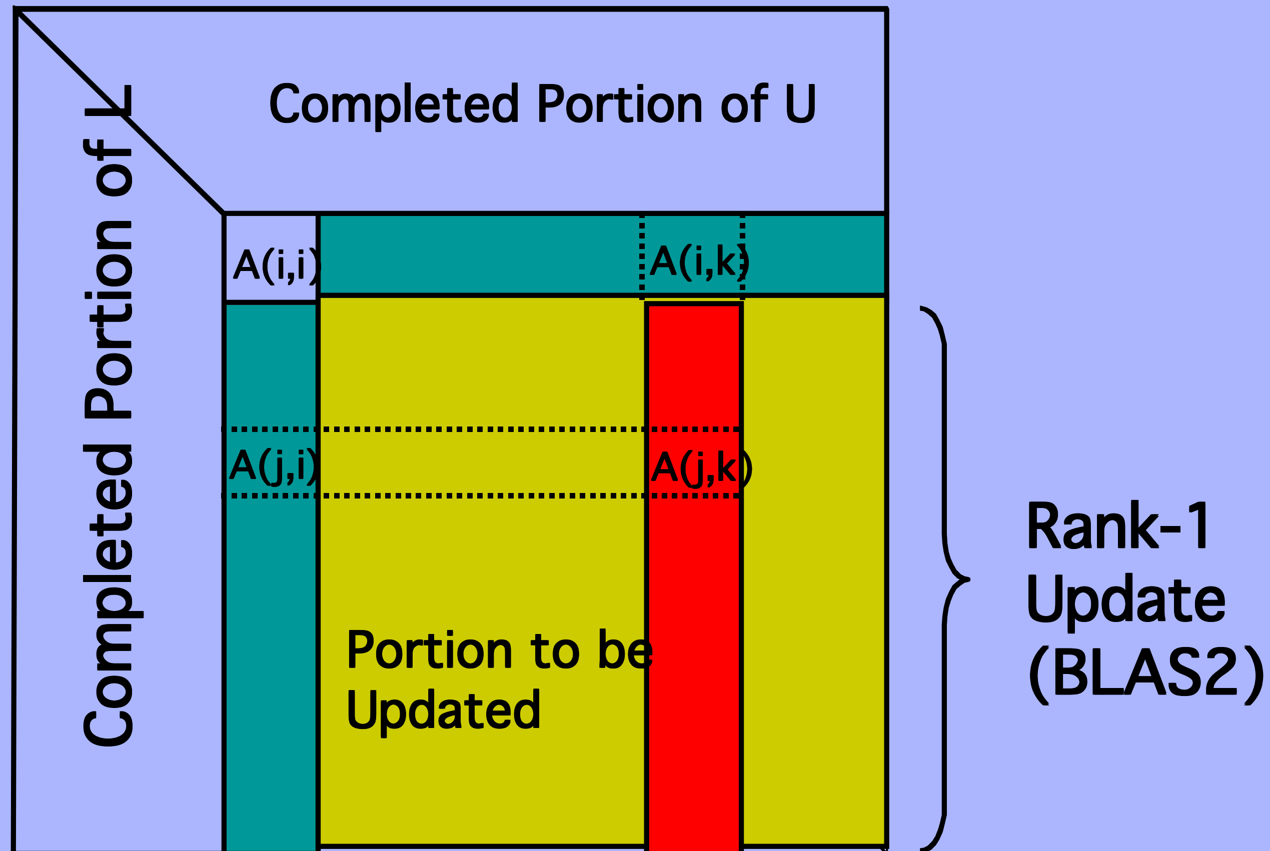
- Gaussian Elimination (no pivoting)

```
do i = 1, N-1
   do j = i+1, N
     A(j,i) = A(j,i) / A(i,i)
   end do
   do k = i+1, N
     do j = i+1, N
       A(j,k) = A(j,k) - A(j,i) * A(i,k)
     end do
   end do
 end do
```

BLAS1 (DSCAL)

BLAS1 (DAXPY)

Tune Cache Performance

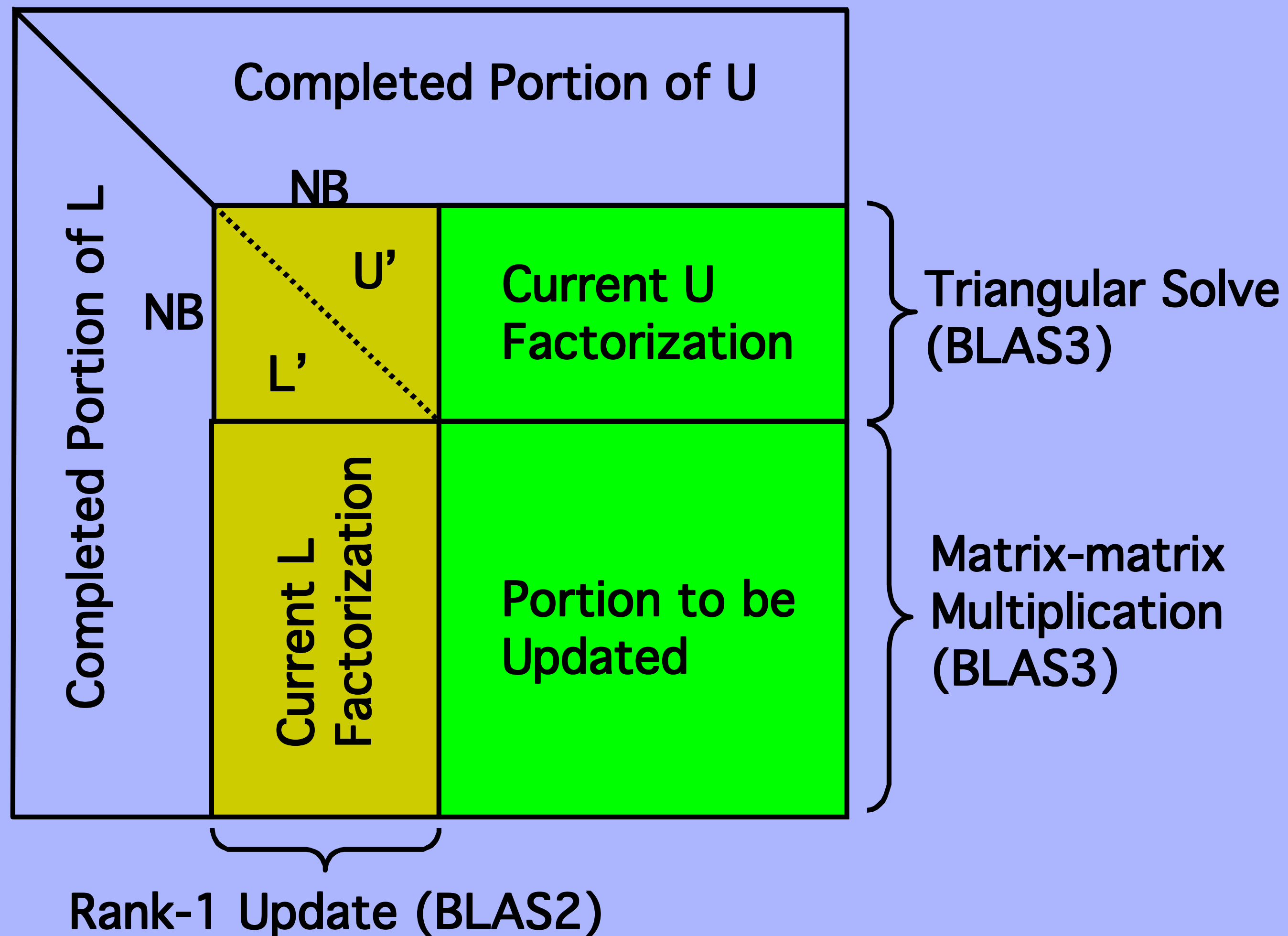


Tune Cache Performance

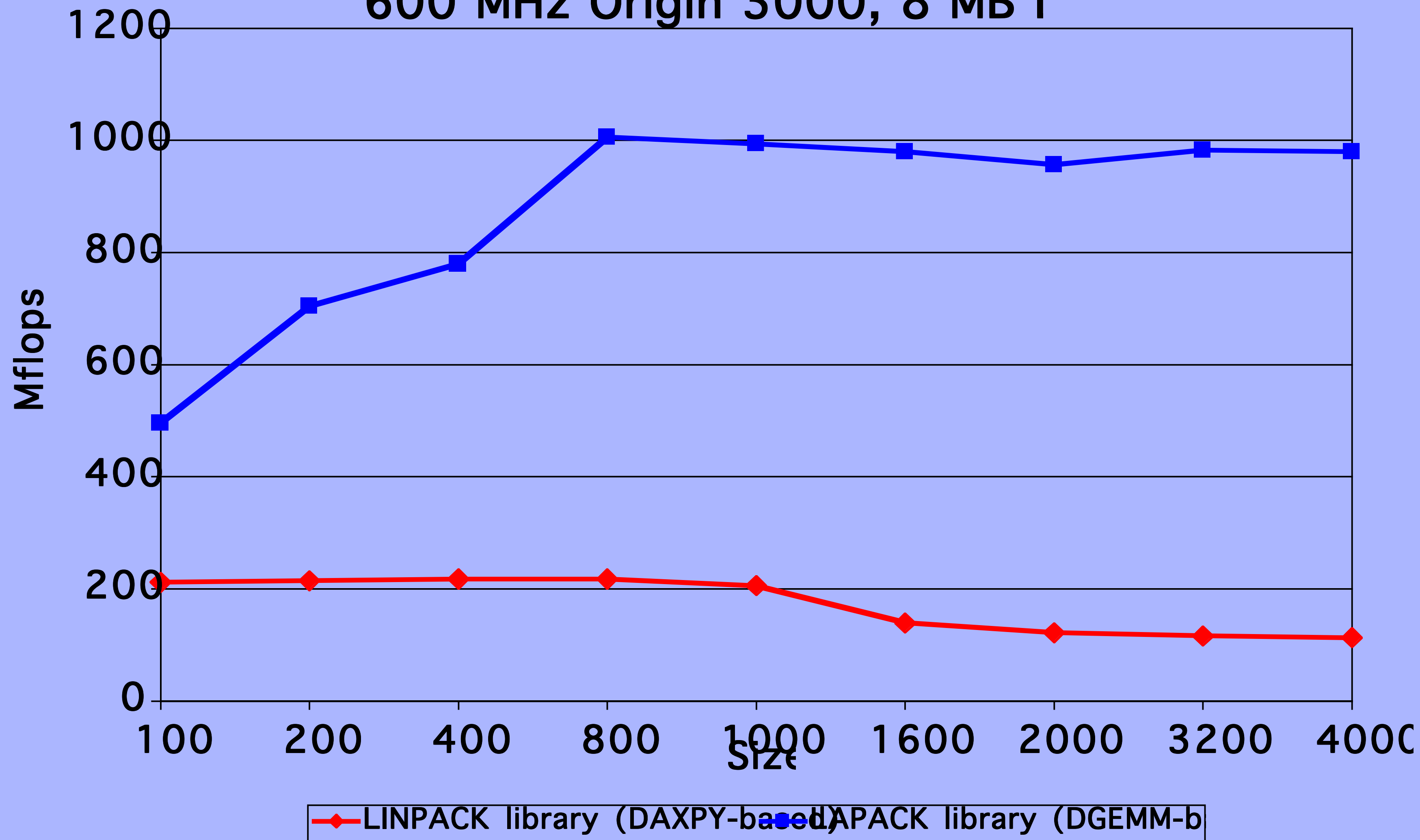
De-vectorize the Code

- Process matrix in blocks of size NB
  - "Delayed updating", a.k.a. blocking
- Rank-1 update becomes matrix-matrix multiplication
- Block size constraints:
  - Small enough for NB columns fit in cache
  - Large enough for fast matrix-matrix multiplication performance

Tune Cache Performance



Tune Cache Performance



Tune Cache Performance

Signal Processing Example

- **Discrete Fourier Transform:**

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{i2\pi nk/N} \Leftrightarrow \mathbf{X} = \mathbf{W}_N \cdot \mathbf{x}$$

- **Matrix-vector multiplication has $O(N^2)$ operations**
- **Can exploit symmetry and periodicity of $W_N^{kn} = e^{i2\pi nk/N}$ to obtain dramatic performance improvements**

Tune Cache Performance

Fast Fourier Transform

- Can exploit symmetry and periodicity of $W_N^{kn} = e^{i2pnk/N}$ to obtain dramatic performance improvements
- O(N log2(N)) operations
- Log2(N) passes through the data

Tune Cache Performance

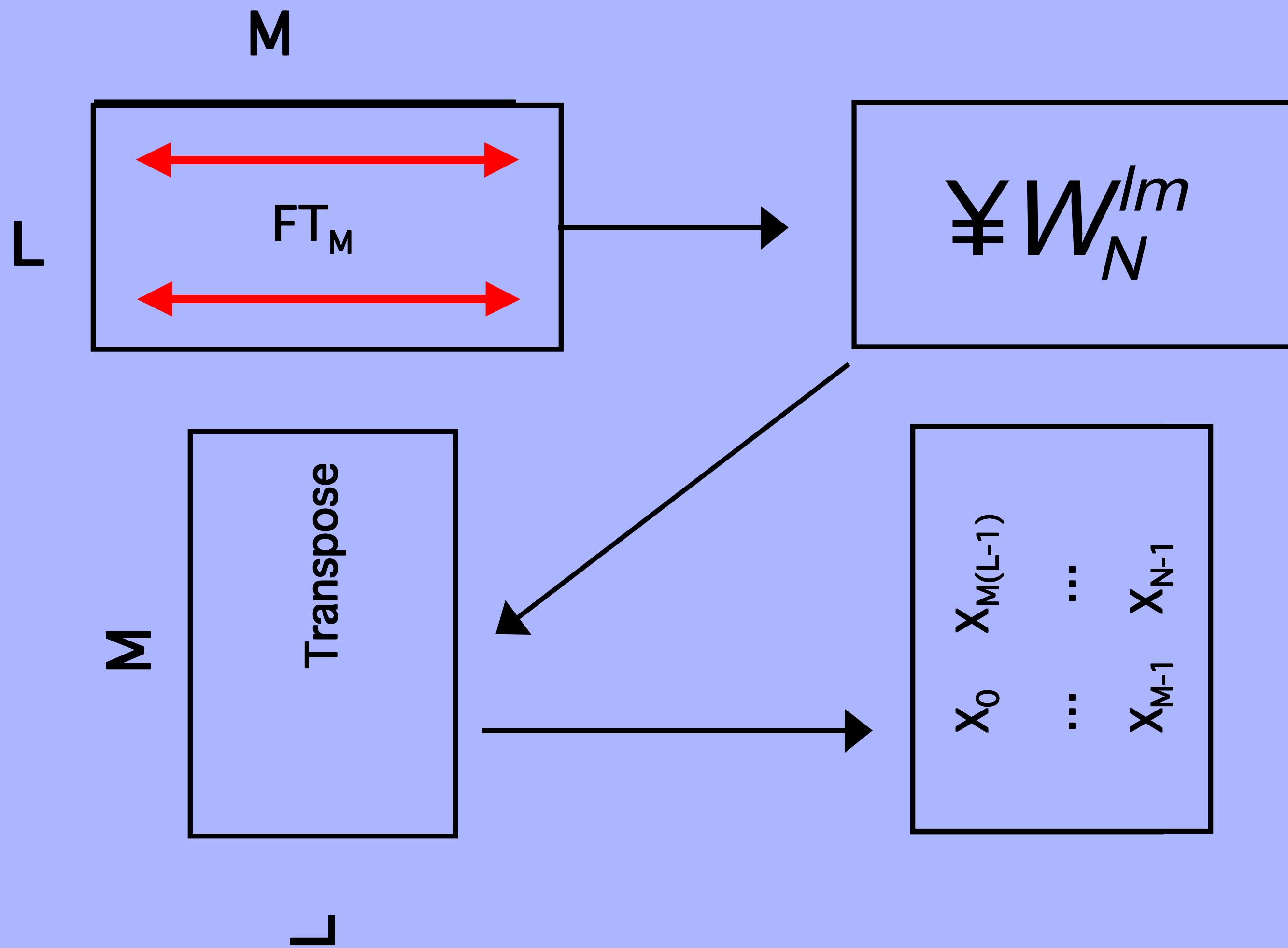
“Fast” Fourier Transform

– If N = L M, can reshape x as a 2-D matrix of dimensions (L,M):

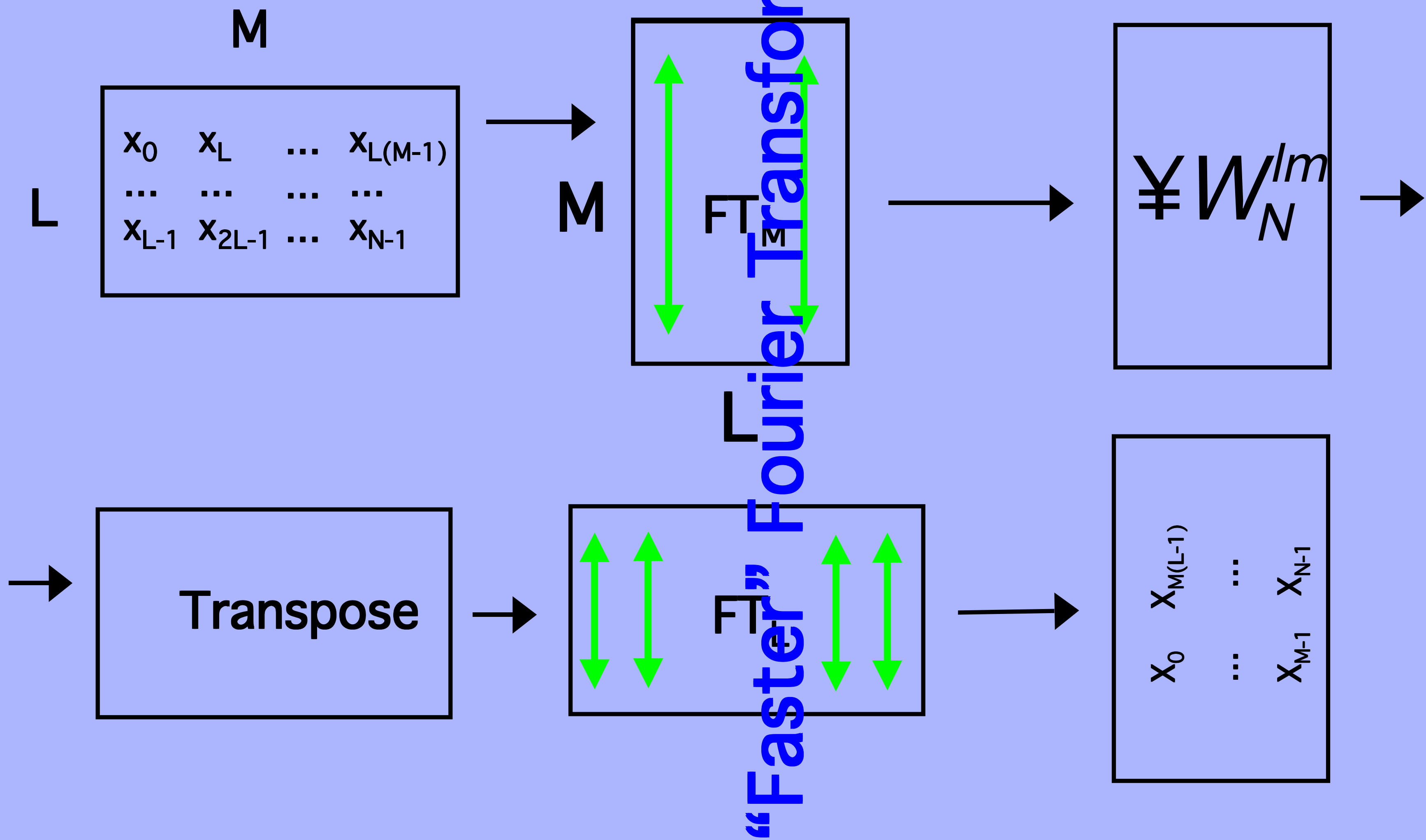
$$\mathbf{X} = FT_N(\mathbf{x}_N) = FT_L\left(\mathbf{W}_N \circ FT_M\left(\mathbf{x}_{L \times M}\right)\right)$$

- Perform L transforms of length M
- Weight with “twiddle” matrix W_N
- Transpose matrix
- Perform M transforms of length L
- More cache-efficient approach

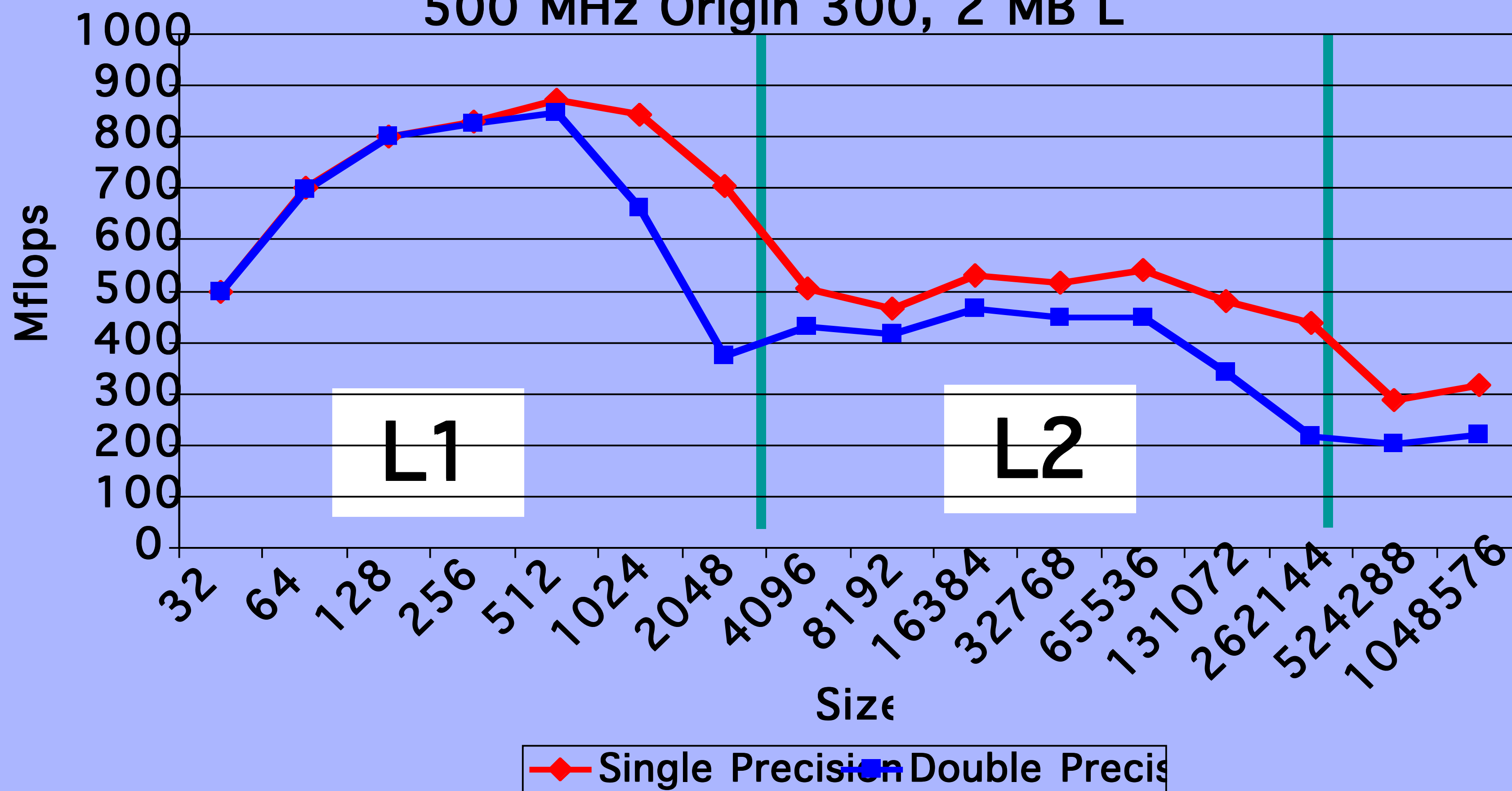
Tune Cache Performance



Tune Cache Performance



Tune Cache Performance



Optimization Techniques in SCSL

Summary

- Rely on the Compiler
  - Use –O3 whenever possible
  - Check quality of software pipelining in love notes
  - File bugs on performance divots
- Cache is King
  - LNO is your friend
  - Make use of BLAS3 routines (de-vectorize code)
  - Restructure algorithms, if necessary