Argonne NATIONAL LABORATORY

Autotuning and Specialization: Speeding up Matrix Multiply for Small Matrices with Compiler Technology

Jaewook Shin[1], Mary W. Hall[2], Jacqueline Chame[3],
Chun Chen[2], Paul D. Hovland[1]
[1]ANL/MCS
[2]University of Utah
[3]USC/ISI

iWAPT 2009, October 2, 2009

THE UNIVERSITY OF CHICAGO
Office of Science U.S. DEPARTMENT OF ENERGY
A U.S. Department of Energy laboratory managed by The University of Chicago

PROBLEM STATEMENT

Gap between H/W and S/W Performance

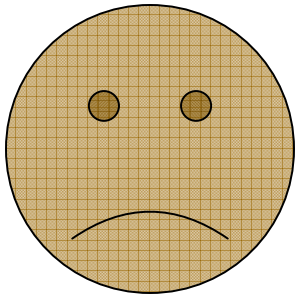
- Moore's Law
  - What do we do with the exponentially increasing number of transistors?
- H/W performance increases through ...
  - More parallelism
    - *Longer vector length for SIMD*
    - *More cores*
  - Heterogeneous architectures
    - *STI CELL processors*
    - *NVIDIA Graphics processors*
  - More hard-to-use instructions
    - *prefetch, cache manipulations, SIMD, ...*

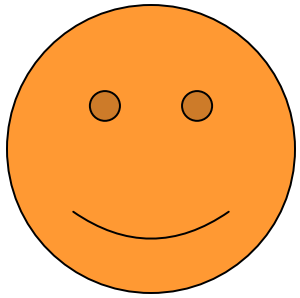
➔ H/W is becoming too complex to utilize all features.

➔ The fraction of H/W performance achieved by S/W decreases.
(Increasing performance gap between H/W and S/W)

COMPILER-BASED EMPIRICAL PERFORMANCE TUNING

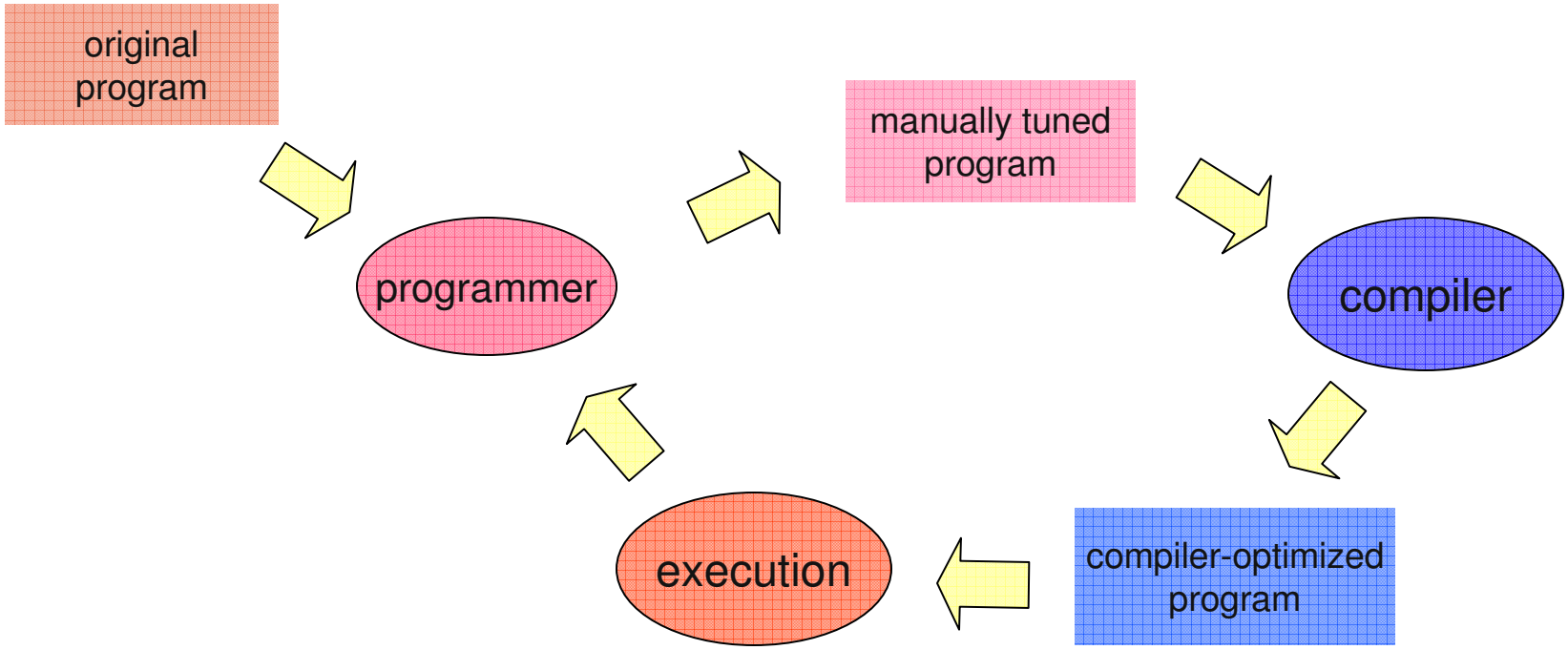
Performance Tuning





Manual Performance Tuning

- Performance is split between compiler and programmer



Application Developers

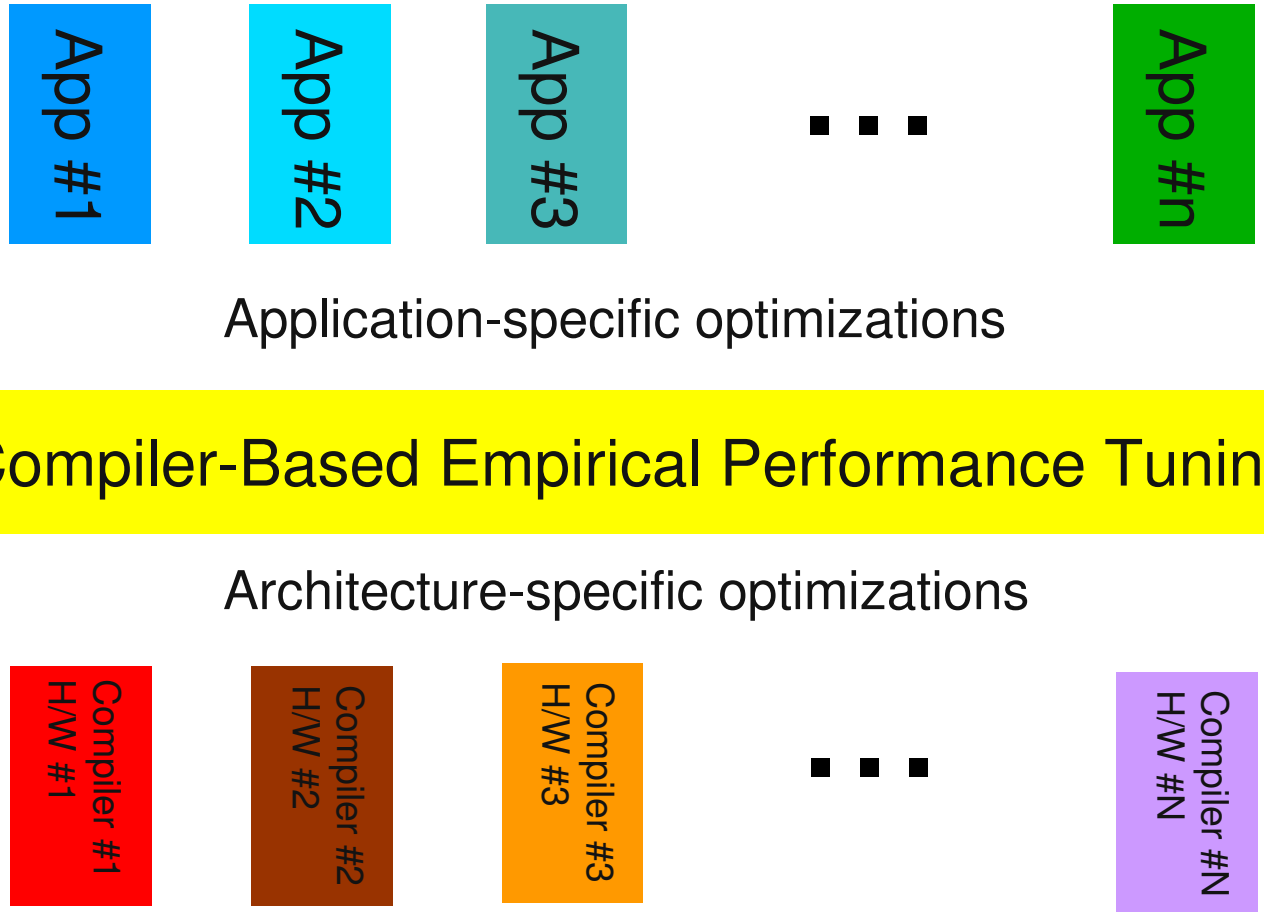
- Significant human efforts ➔ expensive
- Human can explore ➔ slow
  - Small set of code transformations
  - Small set of code variants
  - For one machine-application pair at a time
- Relies on human experiences ➔ error prone

- Mechanical and repetitive ➔ Should be performed by tools automatically

Compiler Optimizations

- Tied to a H/W architecture ➔ Not portable
- Conservative assumptions for unknown information
  - Static analyses cannot benefit from dynamic information.
  - Optimizations are good at mostly simple codes.
- Based on a static profitability model
  - Only the optimizations profitable for most applications
- Fixed order of optimizations

Compiler-Based Empirical Performance Tuning (Autotuning)



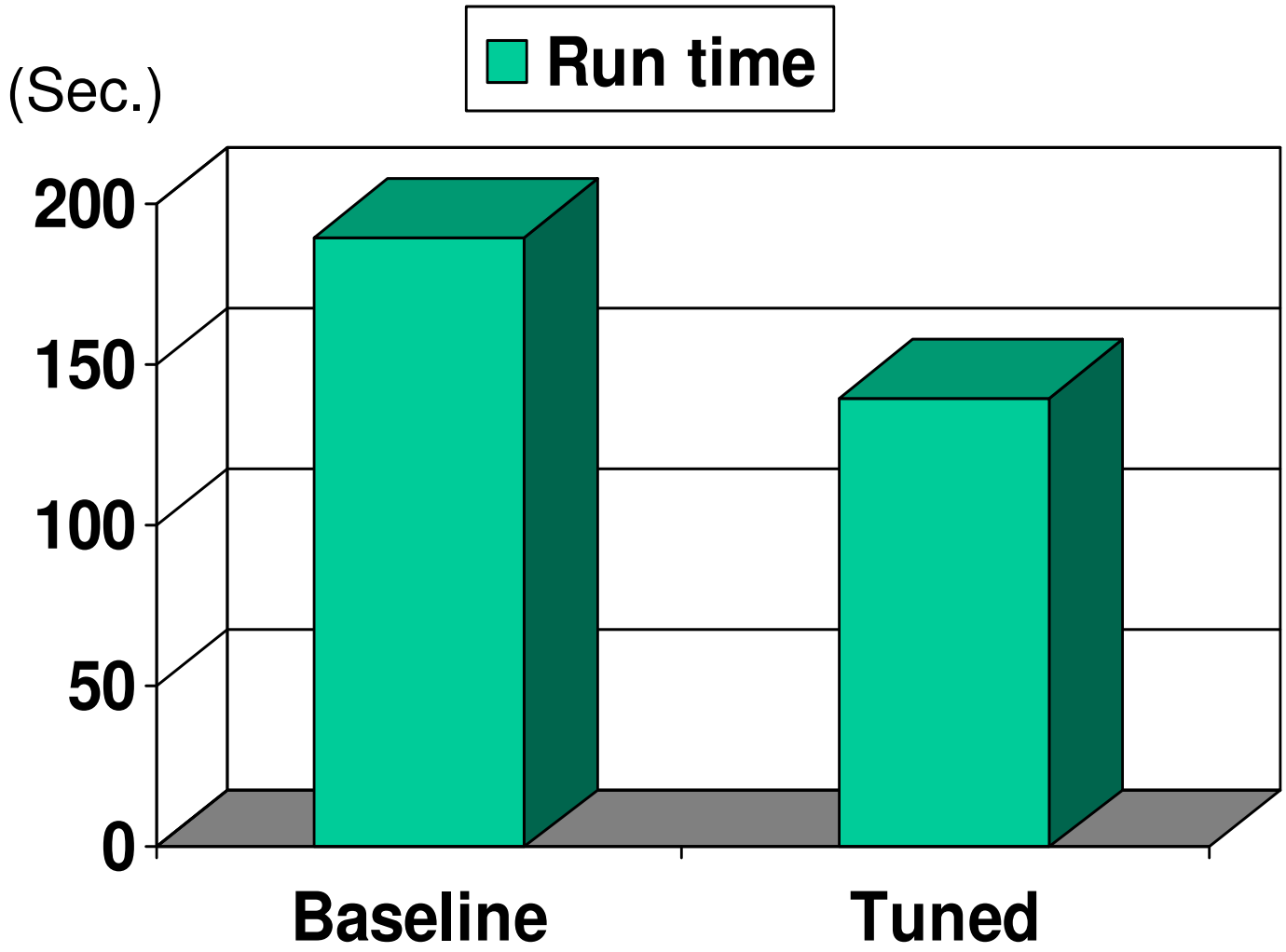
IN THIS PAPER ...

We are…	We are not …
Compilier-based	Manual nor library-based
Collaborative	Fully automatic

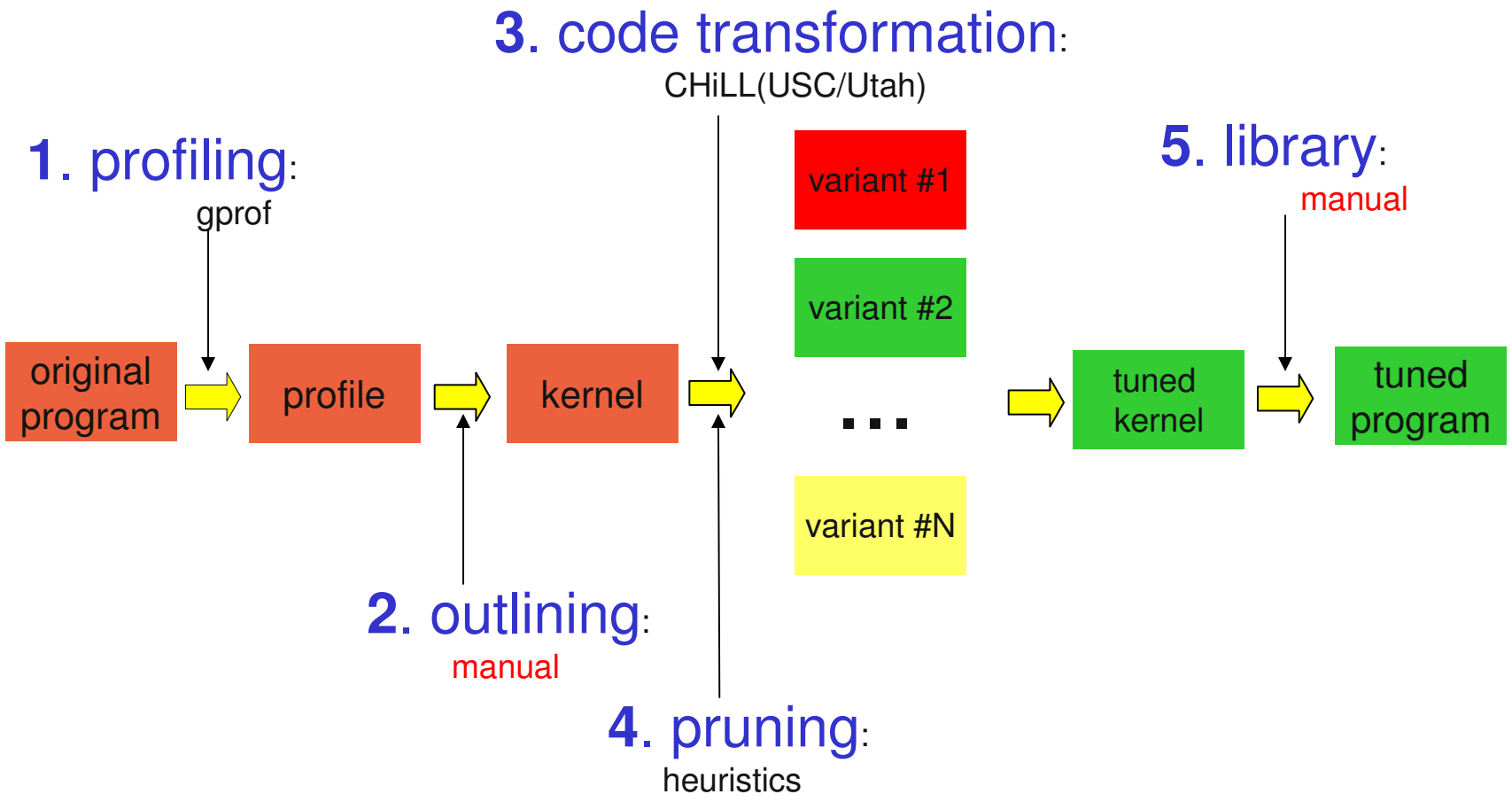
Nek5000

- High-order spectral element CFD code
- http://nek5000.mcs.anl.gov
- Scales to #P > 100,000
- 30,000,000 cpu-hour INCITE allocation (2009)
- Early science application on BG/P
- Applications:
  - nuclear reactor modelling, astrophysics, climate modelling, combustion, biofluids, ...

Speedups of Nek5000: ***1.36X***



Compiler-Based Empirical Performance Tuning for Nek5000



Tools and Environment

- Tools
  - Profiling: gprof
  - Code transformation: CHiLL
  - Backend compiler: Intel compilers version **10.1**
  - PAPI (Performance Application Programming Interface)

- AMD Phenom
  - 2.5 GHz, Quad core
  - 4 double-precision floating point operations / cycle ➔ 10 GFlops / core
  - Ubuntu Linux 8.04-x86_**64**
    - All 16 SIMD registers are available.
  - Kernel patched with ***perfctr***

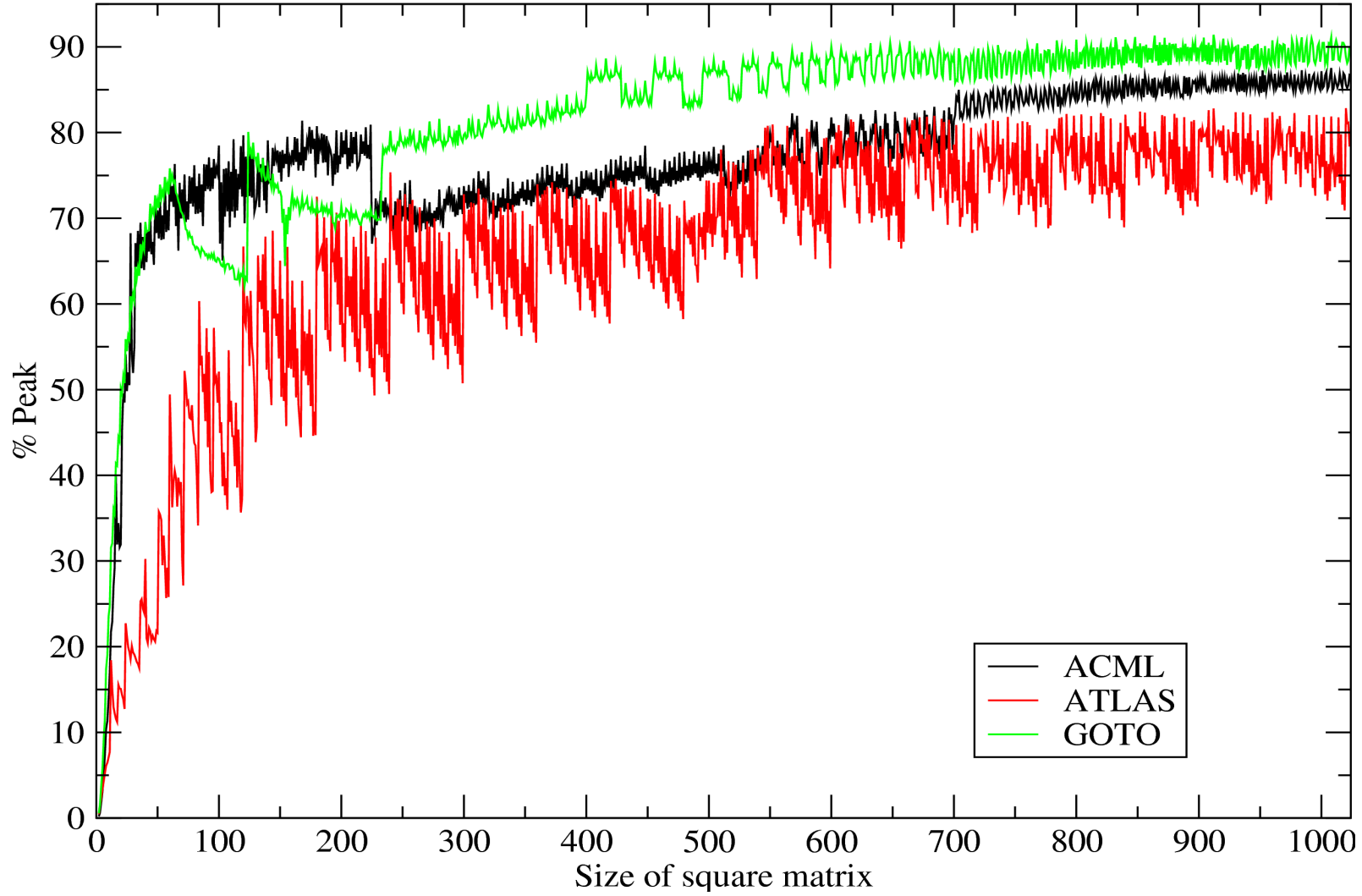
Profiling

- ~ 60% of run time spent in mxm44_0
- mxm44_0 → **Baseline**
  - Dense matrix multiplication
  - Small rectangular matrices
  - Hand-optimized by 4x4 unrolling (434 lines)
  - 8 input sizes comprise 74% of all computation

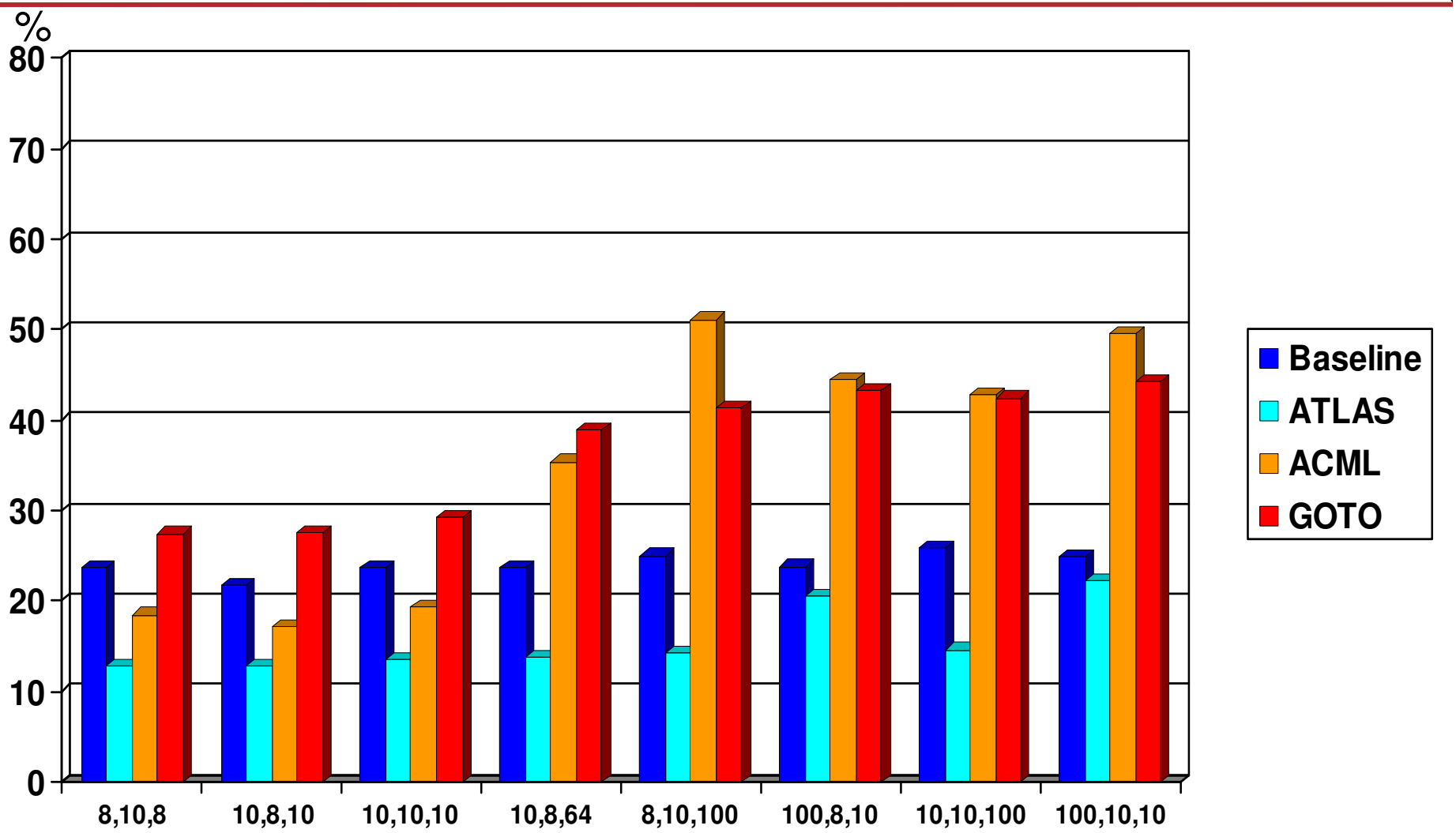
size	m	k	n
1	8	10	8
2	10	8	10
3	10	10	10
4	10	8	64
5	8	10	100
6	100	8	10
7	10	10	100
8	100	10	10

  - Matrix sizes depend only on the degree of problem

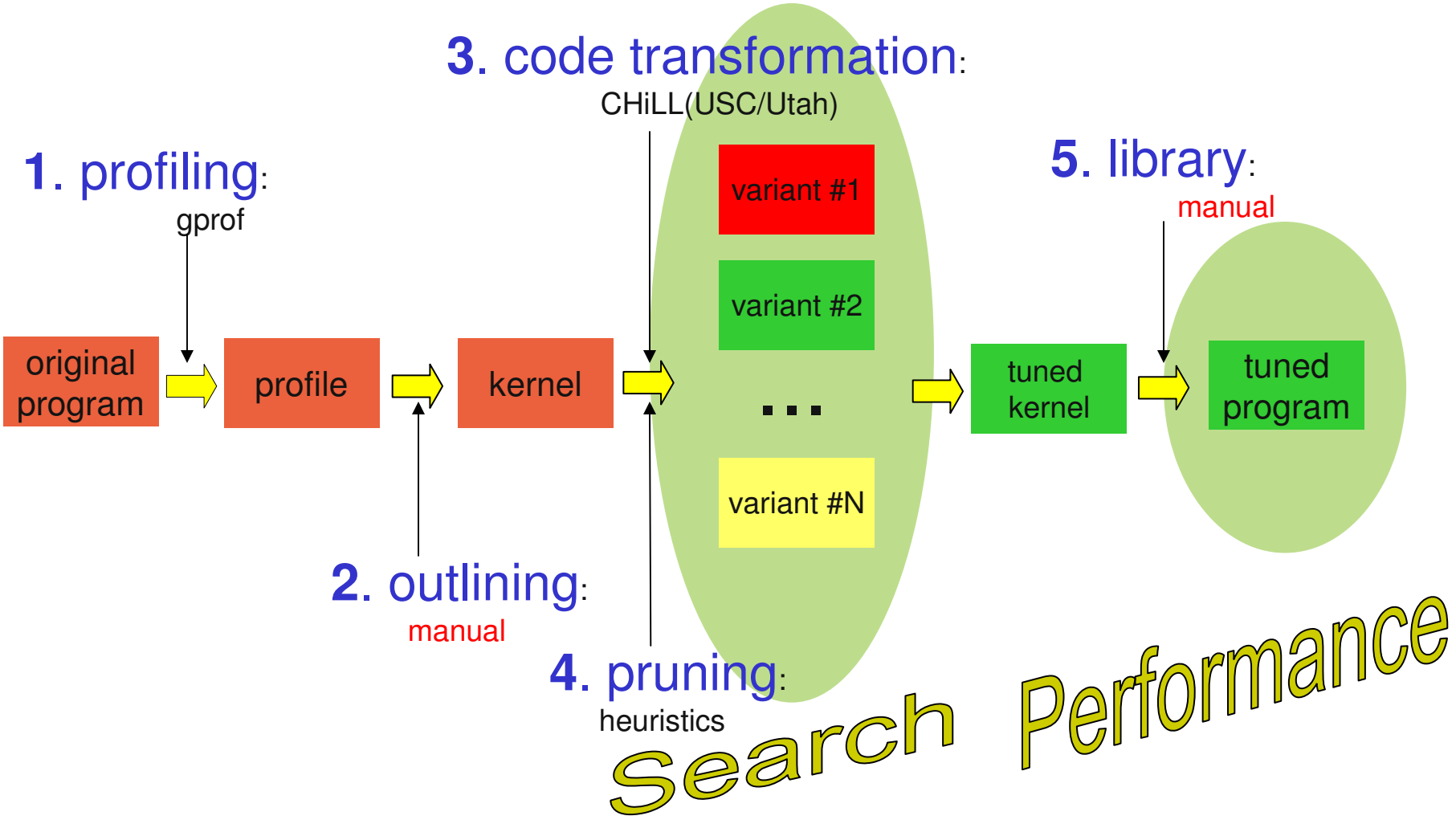
BLAS Library Performance



BLAS Library Performance: Small Rectangular Matrices



Contributions: Fast Search & High Performance



CONTRIBUTION #1:

EMPIRICALLY OBTAINED HEURISTICS FOR PARAMETER-SPACE PRUNING

Dense Matrix Multiply Kernel

```
do i=1,M
   do j=1,N
      C(i,j)=0.0
      do k=1,K
         C(i,j)+=A(i,k)*B(k,j)
```

- Input matrices are represented as (M,K,N).
- The loop order of this loopnest is 123 for (i,j,k).

Two Code Transformations

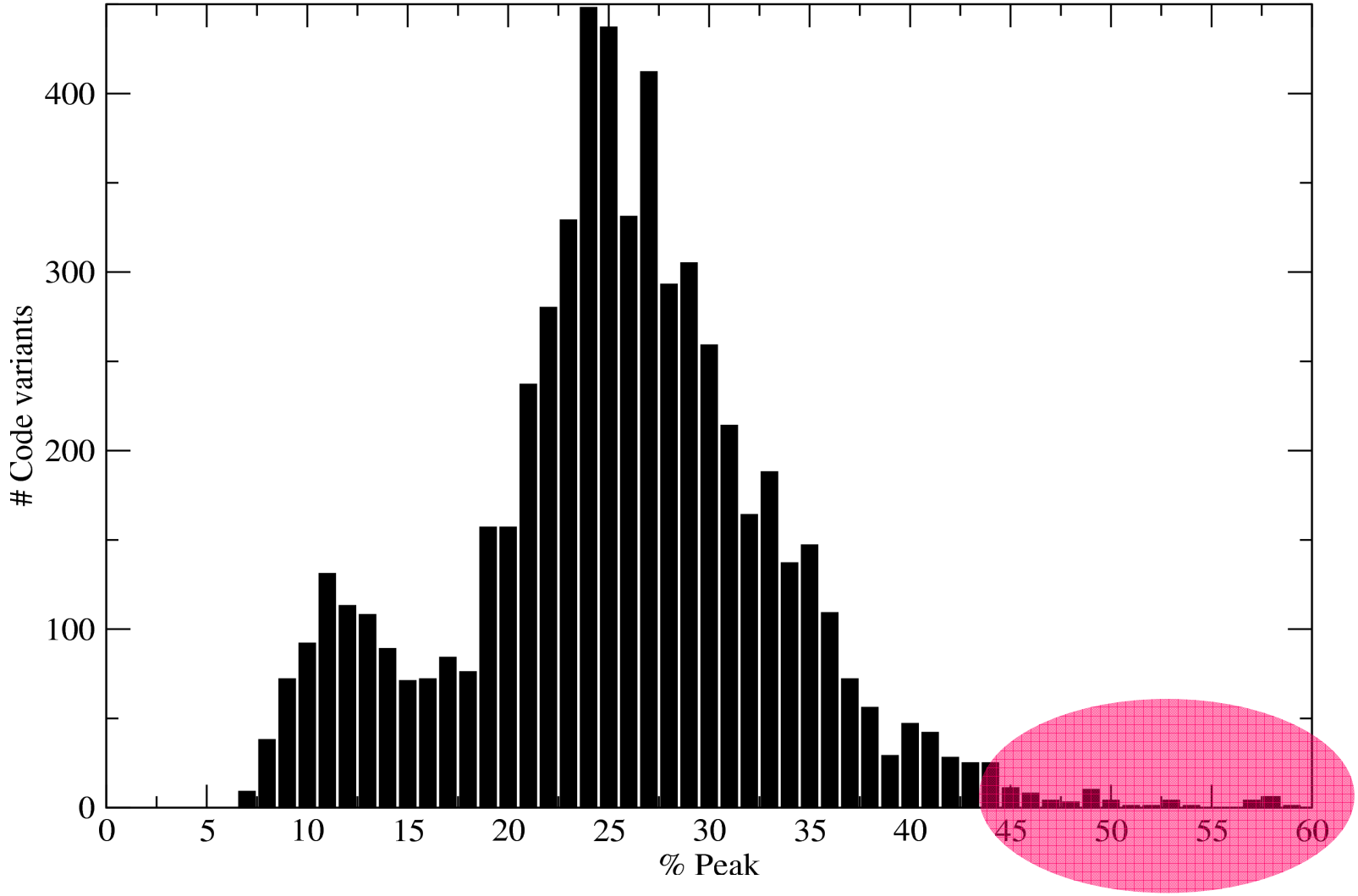
- Unrolling: Increases instruction-level parallelism (ILP), ...
- Loop permutation: Affects the compiler's SIMD code generation, ...
- Example: 10x10x10

Variant #	loop order	Ui	Uk	Uj
1	123(ijk)	1	1	1
2	123(ijk)	1	1	2
...	...	...	...	...
11	123(ijk)	1	2	1
...	...	...	...	...
1000	123(ijk)	10	10	10
1001	132(ikj)	1	1	1
1002	132(ikj)	1	1	2
...	...	...	...	...
6000	321(kji)	10	10	10

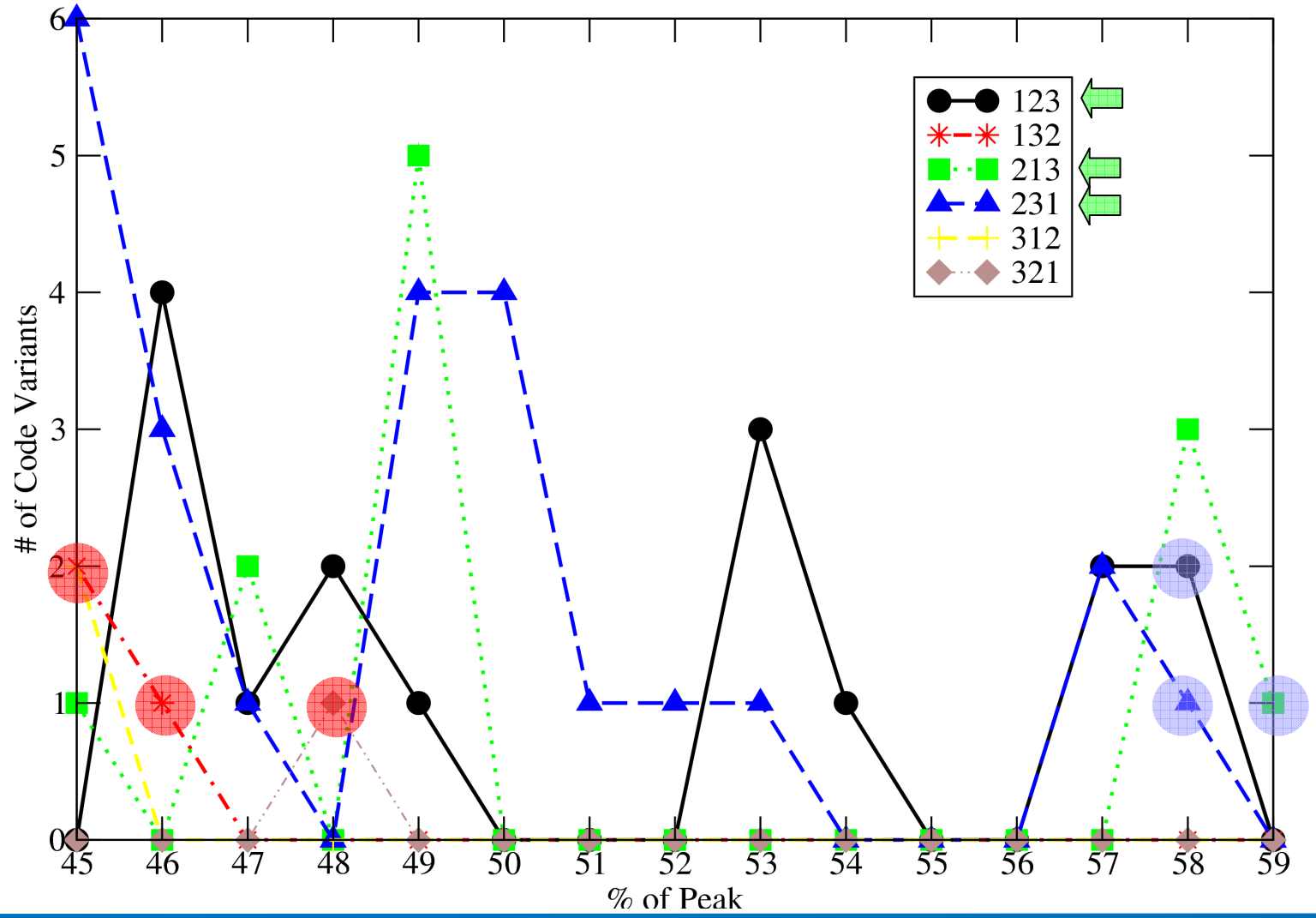
Parameter Space

- Formed by a set of all possible code variants
- Loop permutation
  - Six loop orders for the three loops of *mxm*
- Unrolling
  - N unroll factors for a loop with N iterations ranging from 1 to N
  - M*K*N unrolling for the three loops of M, K and N iterations
- Time budget for tuning: 1 day
- Examples:
  - 10x10x10: 6,000 variants → OK (~ 7 hours)
  - 100x10x10: 60,000 variants → Unacceptable with exhaustive search
    - *10 times larger in size of the space*
    - *Each point in the space has a larger code.*
- Need either
  - Search and/or
  - Pruning the space

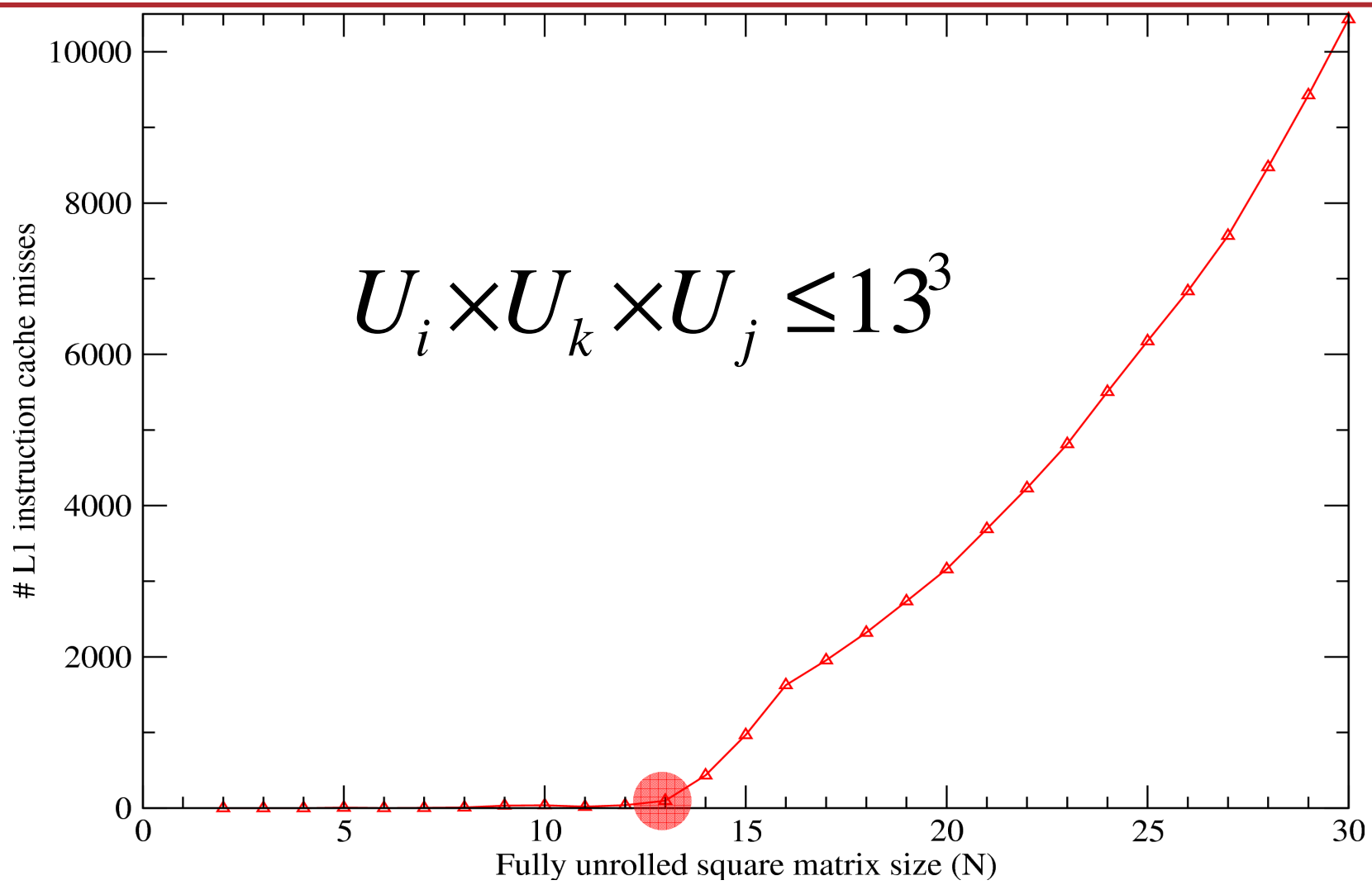
Performance Distribution (10x10x10)



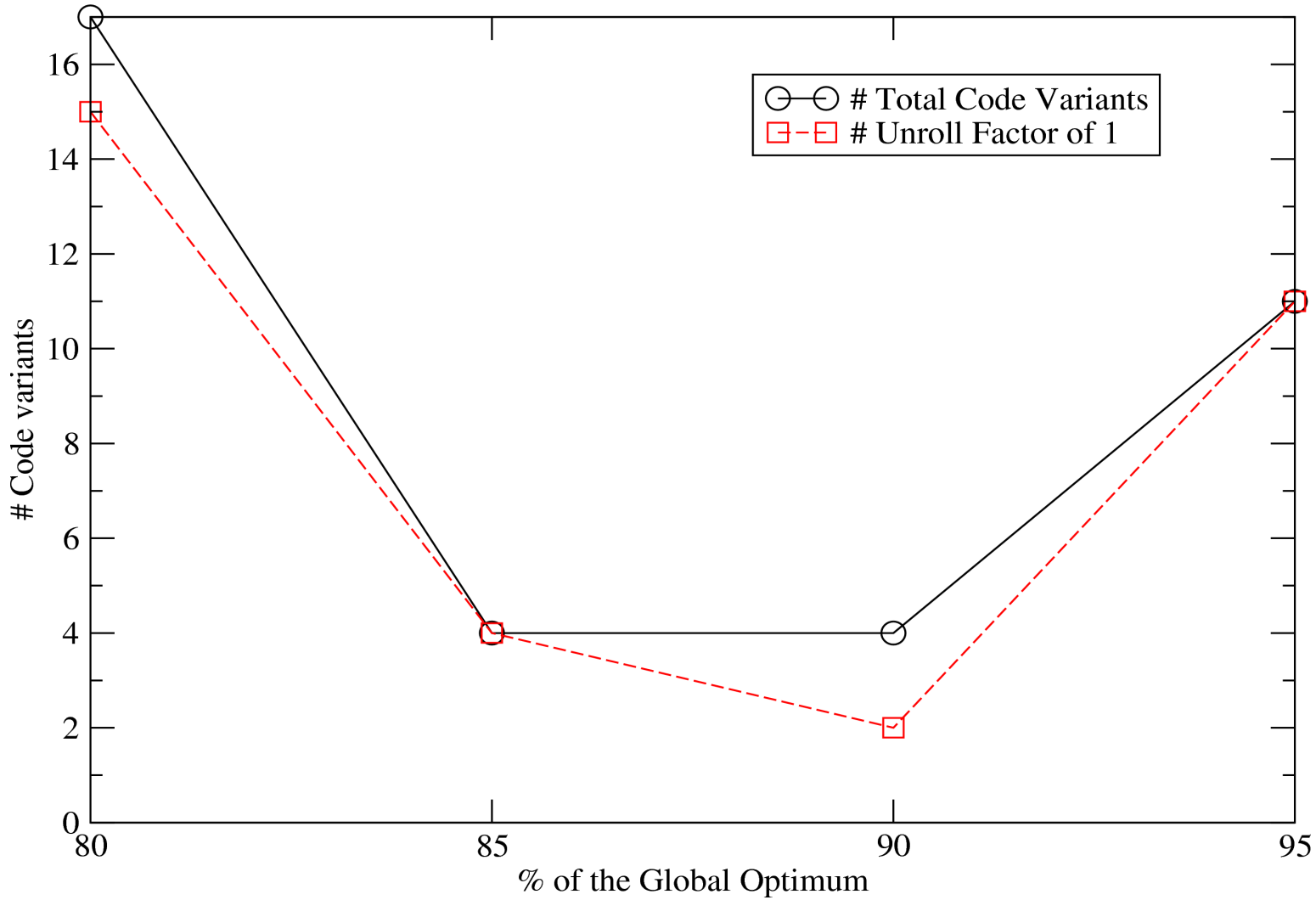
Heuristic #1/4: Loop Order



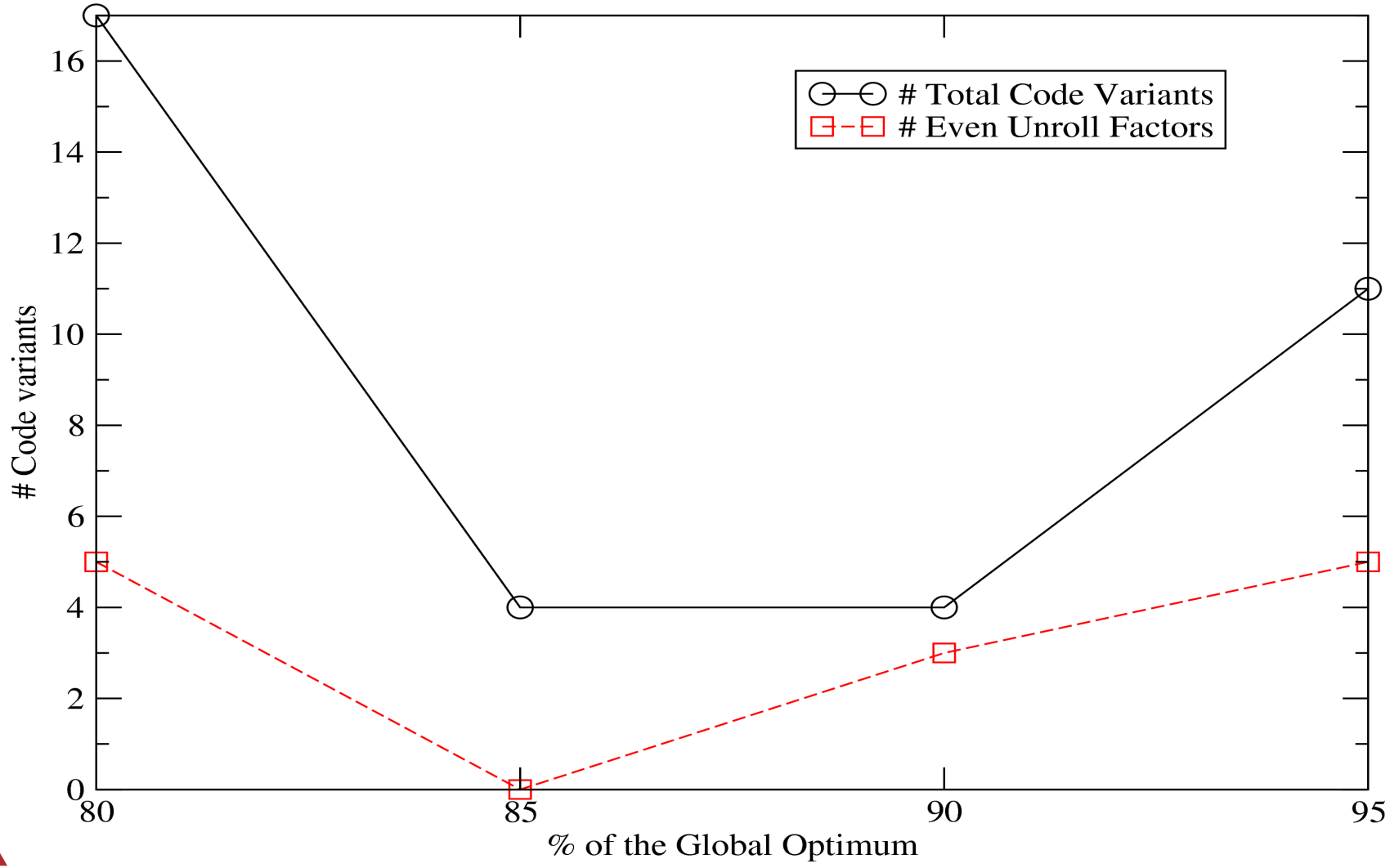
Heuristic #2/4: Instruction Cache



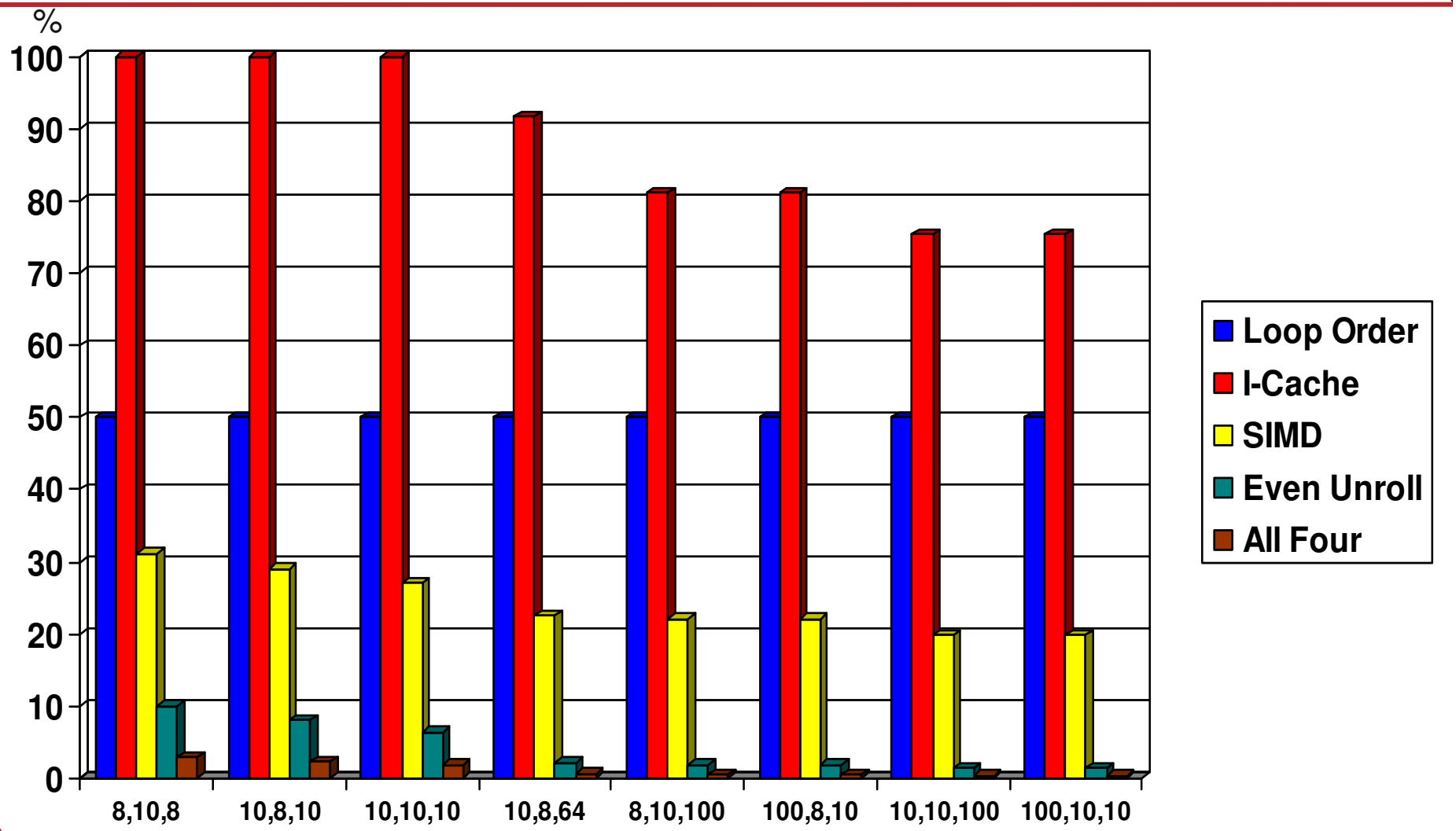
Heuristic #3/4: Unroll Factor of 1 on One Loop (SIMD)



Heuristic #4/4: Unroll Factors Evenly Dividing Iteration Count



Reduction of the Parameter Space by Heuristics

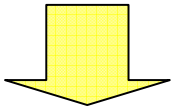


CONTRIBUTION #2:

HIGH PERFORMANCE BY SPECIALIZATION

Specialization

```
mxm(a, M, b, K, c, N){
  for(i=0; i<M; i++)
    for(j=0; j<N; j++)
      for(k=0; k<K; k++)
            c[i][j]+=a[i][k]*b[k][j];
}
```



```
mxm_101010(a, b, c){
  for(i=0; i<10; i++)
    for(j=0; j<10; j++)
      for(k=0; k<10; k++)
        c[i][j]+=a[i][k]*b[k][j];
}
```

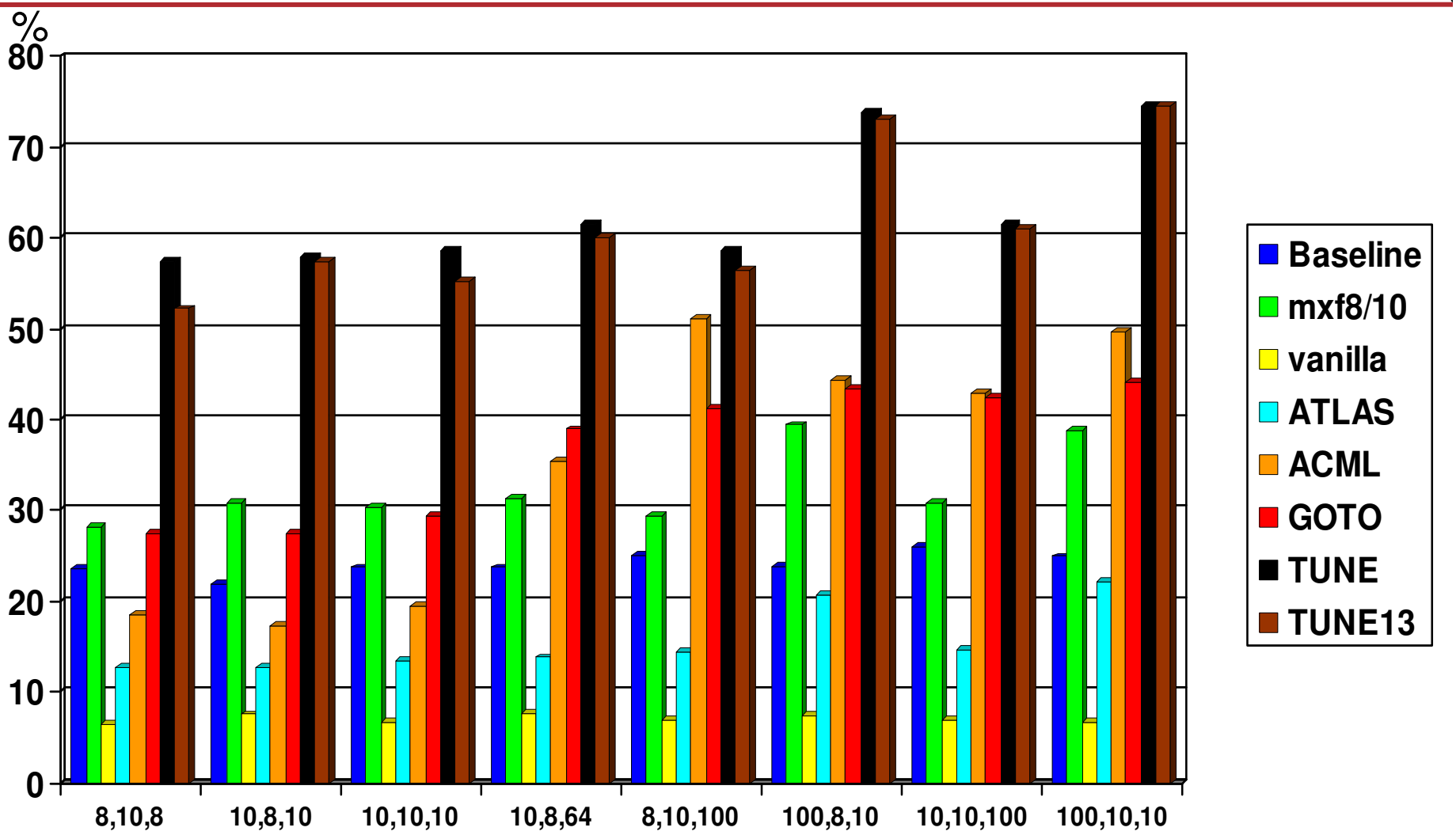
```
mxm(a, M, b, K, c, N){
  if(M==10&&K==10&&N==10)
    mxm_101010(a,b,c);
  else
    mxm_original(a,M,b,K,c,N);
}
```

➔ Fix the input matrix sizes

*High Performance by **Specialization***

- Simpler code for more information (CHiLL, ifort)
  - Makes a simple kernel simpler
  - Concrete information for compilers
  - Ex) Interprocedural analysis:
    - *The arrays are aligned to 16 byte boundaries in memory.*
    - *The arrays are not aliased with each other.*
- Code optimization:
  - SIMD:
    - *No conditionals to check for alignments*
    - *No instructions for aligning data*
  - Custom code-transformations
    - *Optimizations tailored to particular input matrix sizes*
  - More efficient code:
    - *Less checking*

Matrix Multiply Performance for Small Matrices (in cache)



The Code Variants Selected by Applying the Heuristics

No.	m,k,n	Size	Loop Order	Ui	Uk	Uj	%max
1	8,10,8	3840	ijk	8	10	4	98.7
2	10,8,10	4800	ijk	1	8	5	100
3	10,10,10	6000	jik	1	9	5	99.3
4	10,8,64	30720	ijk	1	8	4	
5	8,10,100	48000	ijk	1	10	4	
6	100,8,10	48000	jki	1	8	5	
7	10,10,100	60000	jik	1	10	4	
8	100,10,10	60000	jik	1	10	10	

Custom Code-Transformations

No.	m,k,n	1	2	3	4	5	6	7	8
1	8,10,8	58	27	49	38	58	49	56	54
2	10,8,10	43	61	58	20	20	51	39	58
3	10,10,10	39	37	59	31	20	52	44	58
4	10,8,64	44	20	54	62	61	47	62	50
5	8,10,100	57	38	57	38	59	50	59	54
6	100,8,10	27	73	74	19	19	75	58	67
7	10,10,100	39	37	58	39	61	52	61	57
8	100,10,10	26	41	71	34	19	62	60	75

(% of peak)

What we've learned are ...

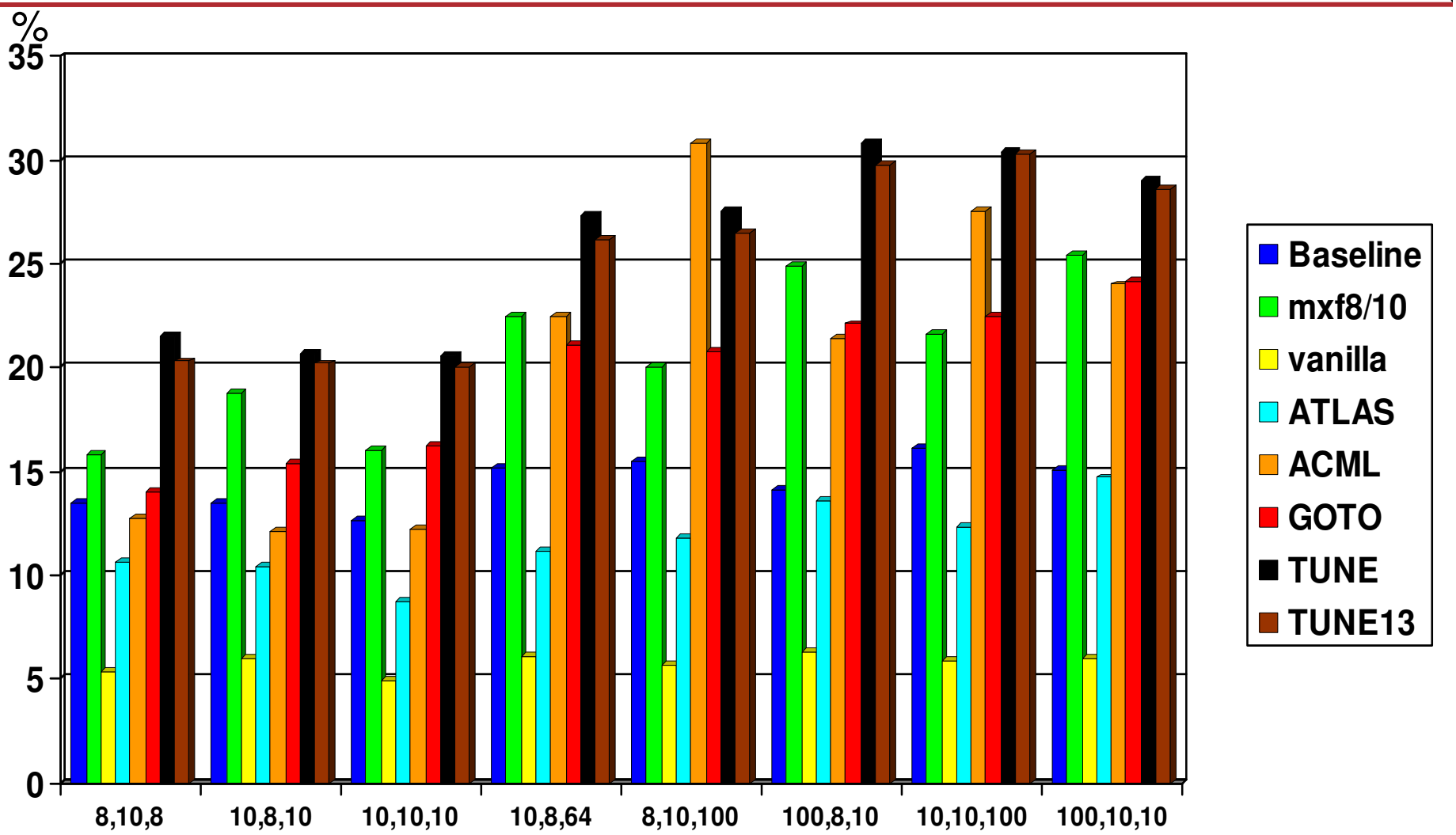
- Job partitioning:
  - Tools: Simple and repetitive work
  - Human: The rest
- Pruning heuristics ⟶ Small parameter space
  - No local searches → embarrassingly parallel
- Specialization
  - Fix the input matrix sizes: ifort, CHiLL
  - Have **ifort** generate aligned SIMD code:
    - *-ipo, __attribute__((aligned (16)))*
  - Simpler input to the tools ⟶ More information ⟶ High performance
  - Custom code transformations
- Success in tuning Nek5000
  - Potential for a (wide) range of machine-application pairs
    - *No dependency or commutative operations*
    - *Small data that fits in the L1 cache*
  - A stride in tuning matrix multiply for small, rectangular matrices

Summary

- The performance gap between H/W and S/W is increasing.
- Compiler-based empirical performance tuning is a viable solution.
- Specialization → custom optimization (~ 74% of peak)
- Pruning heuristics → embarrassingly parallel (Use supercomputers!)
- Future work
  - → Other machines: BG/P,Q
  - → At a higher level
  - → Other applications: UNIC, S3D, MADNESS, ...


Questions?

Matrix Multiply Performance for Small Matrices (memory)



```
do e=1,240
    call mxm (dxm12,8,x(1,e),10,ta1,100)
    do iz=0,9
       call mxm (ta1(1+80*iz),8,iytm12,10,ta2(1+64*iz),8)
    enddo
    call mxm  (ta2,64,iztm12,10,dx(1,e),8)
    do i=1,512
      dx(i,e)=dx(i,e)*rm2(i,e)
    enddo

    call mxm  (ixm12,8,x(1,e),10,ta3,100)
    do iz=0,9
      call mxm (ta3(1+80*iz),8,dytm12,10,ta2(1+64*iz),8)
    enddo
    call mxm     (ta2,64,iztm12,10,ta1,8)
    do i=1,512
      dx(i,e)=dx(i,e)+ta1(i)*sm2(i,e)
    enddo

    do iz=0,9
      call mxm (ta3(1+80*iz),8,iytm12,10,ta2(1+64*iz),8)
    enddo
    call mxm (ta2,64,dztm12,10,ta3,8)
    do i=1,512
      dx(i,e)=dx(i,e)+ta3(i)*tm2(i,e)
    enddo
    do i=1,512
      dx(i,e)=dx(i,e)*w3m2(i,e)
    enddo
enddo
```

Array	Size(byte)
dxm12	640
dytm12	6400
dztm12	640
ixm12	640
iytm12	640
iztm12	640
ta1	6400
ta2	5120
ta3	6400
x(1,e)	8000
dx(1,e)	4096
rm2(1,e)	4096
sm2(1,e)	4096
tm2(1,e)	4096
w3m2(1,e)	4096
Total	56000

Different Optimization Parameters for Different Input Sizes

No.	m,k,n	1	2	3	4	5	6	7	8
1	8,10,8	58	27	49	38	58	49	56	54
2	10,8,10	43	61	58	20	20	51	39	58
3	10,10,10	39	37	59	31	20	52	44	58
4	10,8,64	44	20	54	62	61	47	62	50
5	8,10,100	57	38	57	38	59	50	59	54
6	100,8,10	27	73	74	19	19	75	58	67
7	10,10,100	39	37	58	39	61	52	61	57
8	100,10,10	26	41	71	34	19	62	60	75

Selected parameters

Loop bound is smaller than the applied unroll amount

Pruned parameters