Code Generation and Autotuning in Computer Algebra

Jeremy Johnson, Werner Krandick, David Richardson
Department of Computer Science
Drexel University, USA

Anatole Ruslanov
Department of Computer and Information Sciences
SUNY Fredonia, USA

ACA 2009, Montreal, Canada

Motivation

- Automatic analysis of empirical performance data can lead to significant performance gains
- Computer architecture today is
  - Highly efficient and complex
  - Often proprietary/trade secret
  - Evolves quickly
  - Difficult to model
- Objective: automatically generate and test many implementations (live or at installation).

Automatic generation & tuning - how?

- High-performance depends on
  - The algorithm $\longrightarrow$ automatically generate and test several/many/all
  - The platform architecture* $\longrightarrow$ iterate on generating and testing with many parameters
- The optimal code/algorithm/parameters are determined via runtime experiments

*Pipeline organization, number of registers, integer units, cache and memory hierarchy organization, etc.

Tricky questions (can't ~~model~~ well)

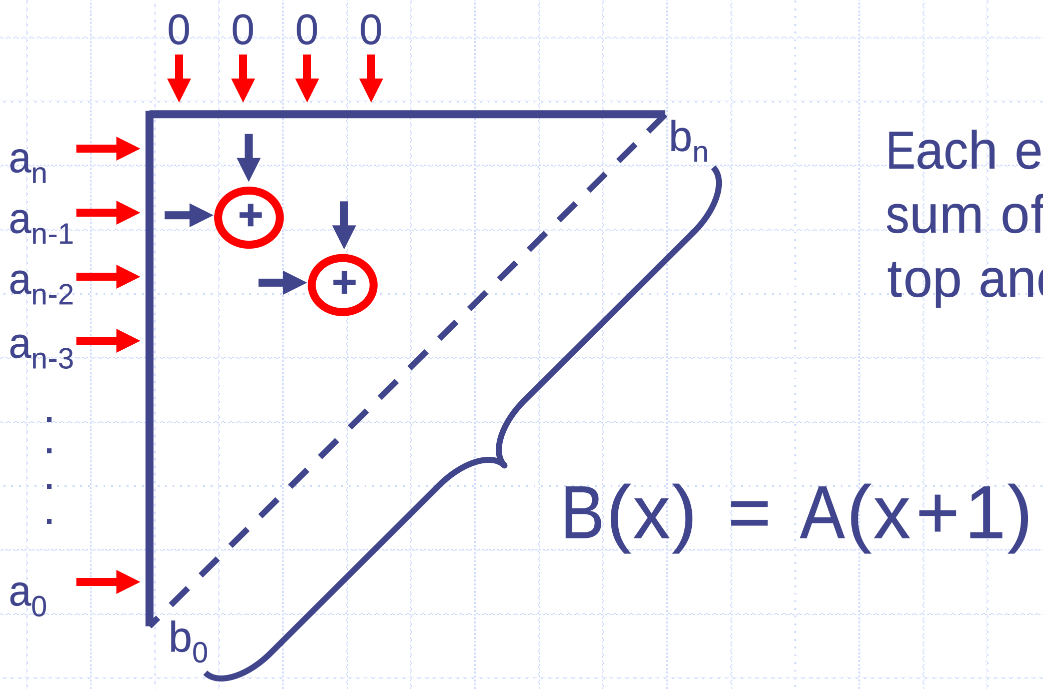
- How is the pipeline organized?
  - Branch misprediction handling
  - Instruction prefetching, issue, reordering
- How is cache organized?
  - How well does it prefetch? How many ports?
- How many integer units are there?
  - How well can they be engaged in parallel?
- How do compilers use the CPU registers?
- What happens when the code is compiled on one machine but run on another?

Apply to computer algebra

- Automatic code generation and tuning techniques may be applied to symbolic computation and computer algebra systems.
- In this talk, we present an example that demonstrate benefits of these techniques.
- We show that the performance of the Taylor shift operation used in real root isolation can be substantially improved through automatic code generation and tuning.

Classical Taylor shift by 1

Let $A(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$
Pascal's triangle with inputs: $a_n, \ldots, a_0$, and $0, \ldots, 0$

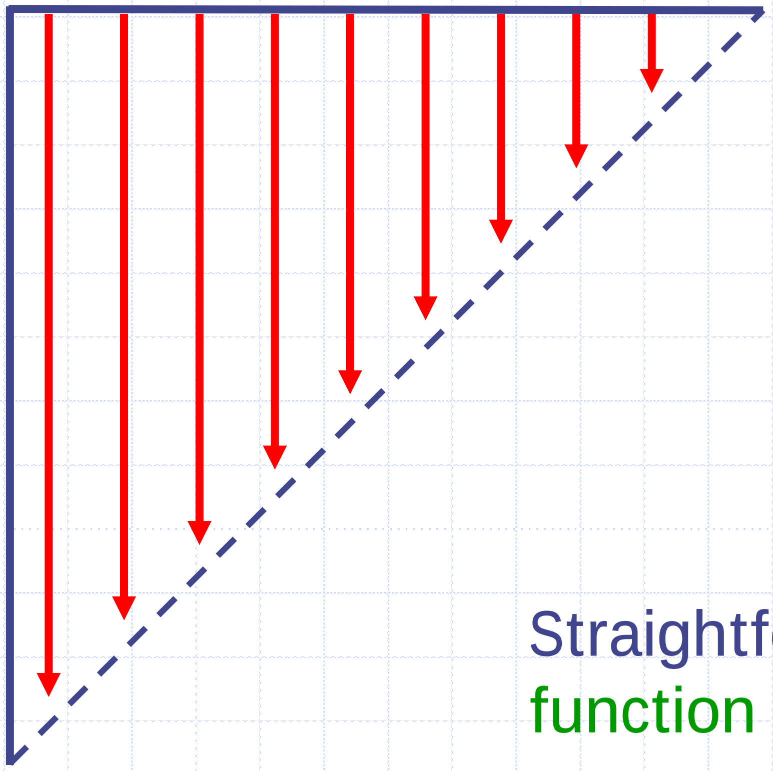


Each element is the sum of its top and left neighbors

$$B(x) = A(x+1)$$

Traditional computation

Sequence of n addition passes



Input: $A(x) = a_n x^n + \ldots + a_0$

```
for  i = 0, … , n-1
    for  k = n-1, … , i
        $a_k \leftarrow a_k + a_{k+1}$
```

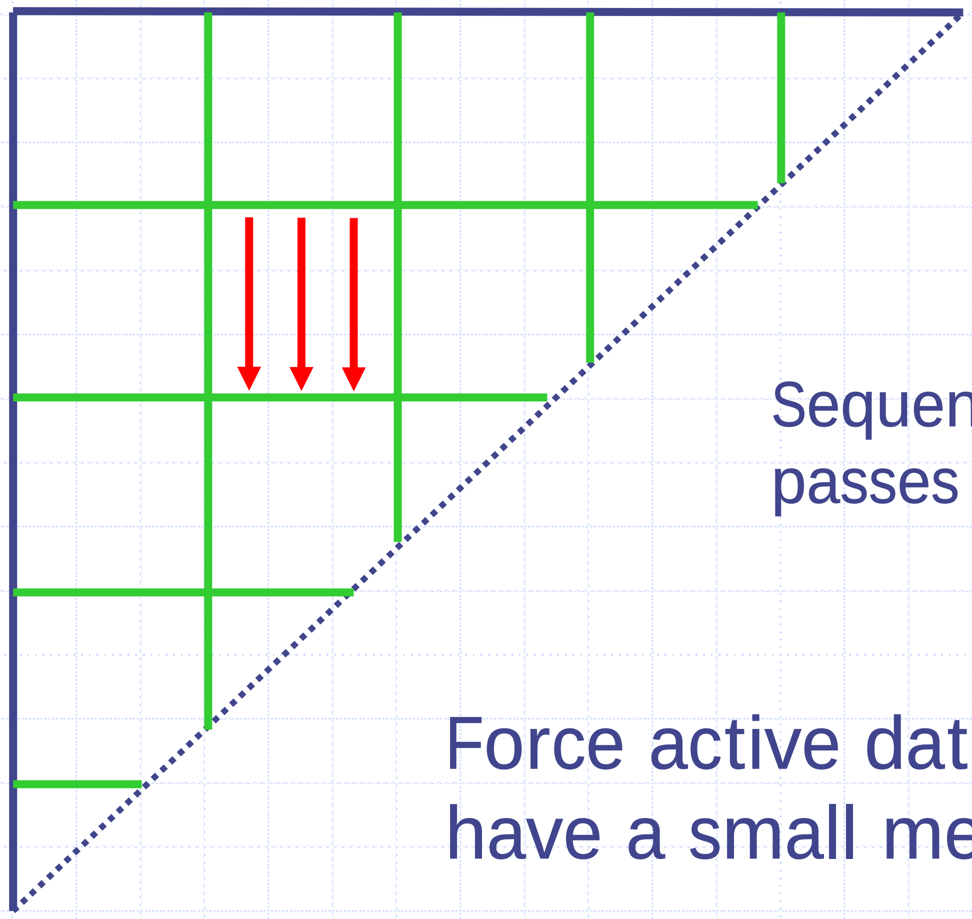
Output: $B(x) = a_n x^n + \ldots + a^0$

Straightforward methods:
function calls to integer addition

Taylor shift by 1 algorithm redesign

- Performance depends on addition
- Minimize cycles per word addition
  - by reducing memory traffic
  - by removing most carry computations
- Arithmetic ideas:
  - signed digits
  - suspended normalization
  - radix reduction
  - delayed carry propagation

Tiling improves data locality

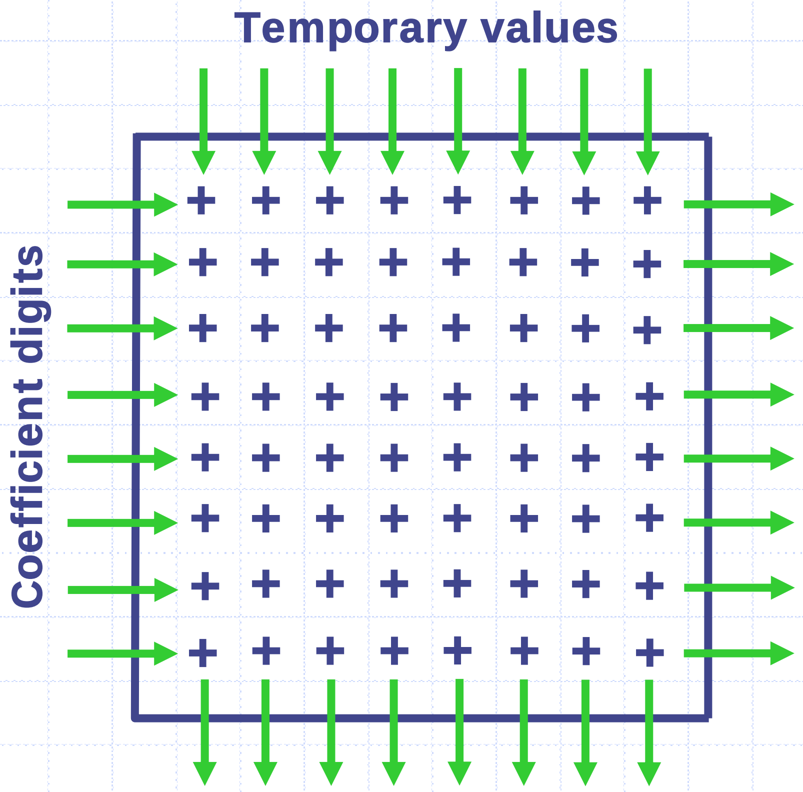


Sequence of addition passes within each tile.

Force active data structures to have a small memory footprint.

Register tile avoids memory traffic

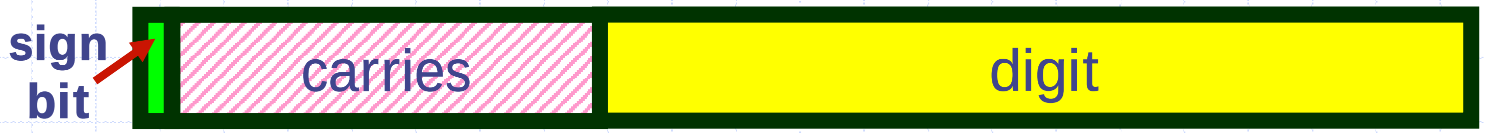
Key idea: avoid reads by keeping all digits in registers.



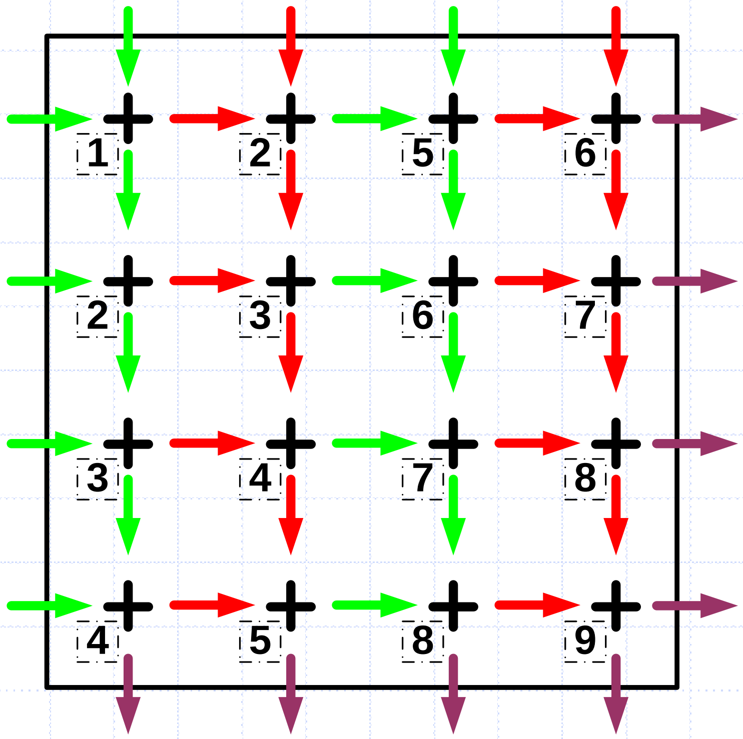
- Do additions for the *i*-th order digits only
  - Read coefficient digits
  - Read temporary values
  - Do additions in registers
  - Store back to L1 cache
- No carry propagation

Delayed carry propagation

- Reduce radix to prevent overflow and absorb carries during register tile computation



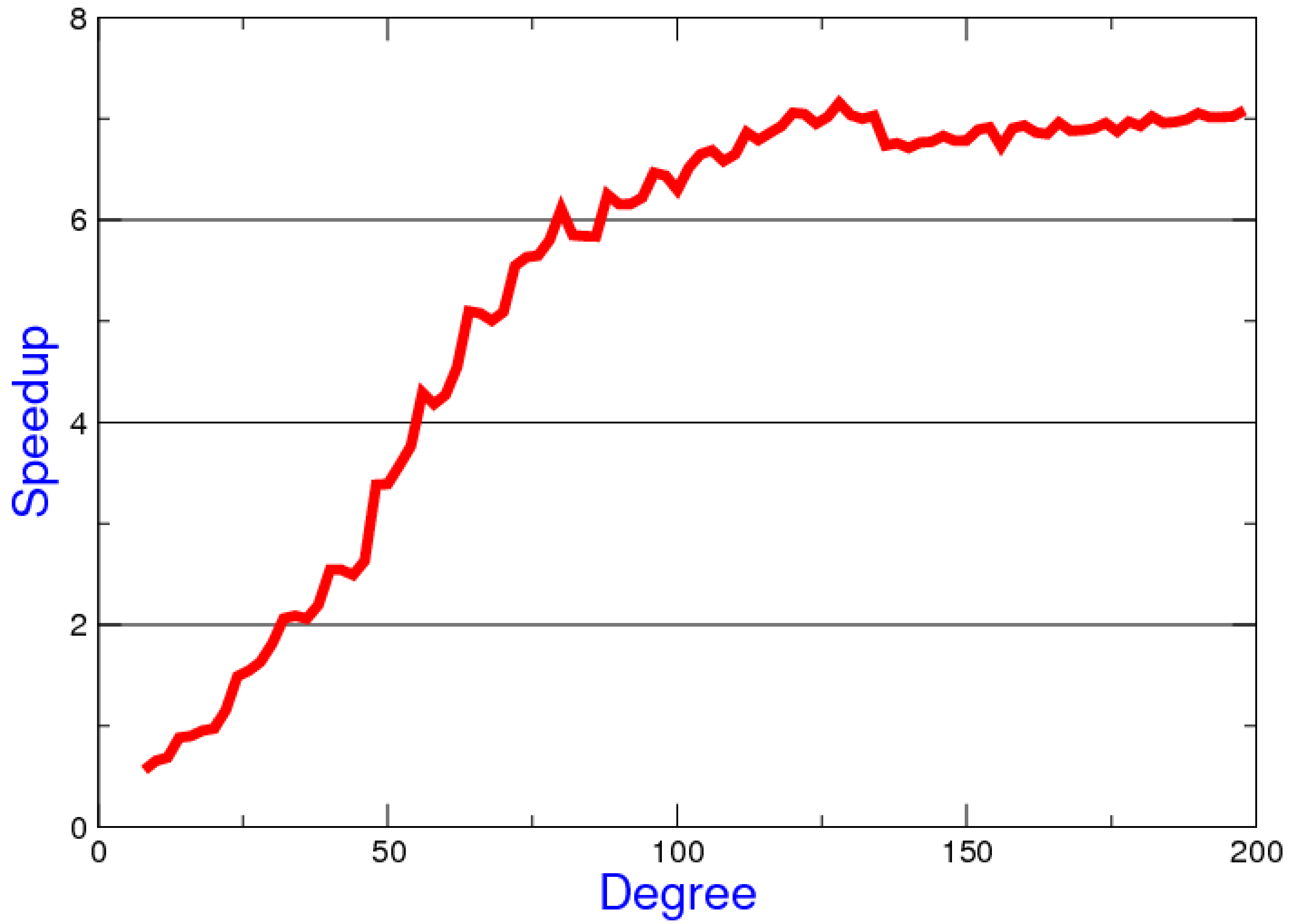
Schedule register tile to improve instruction-level parallelism (ILP)



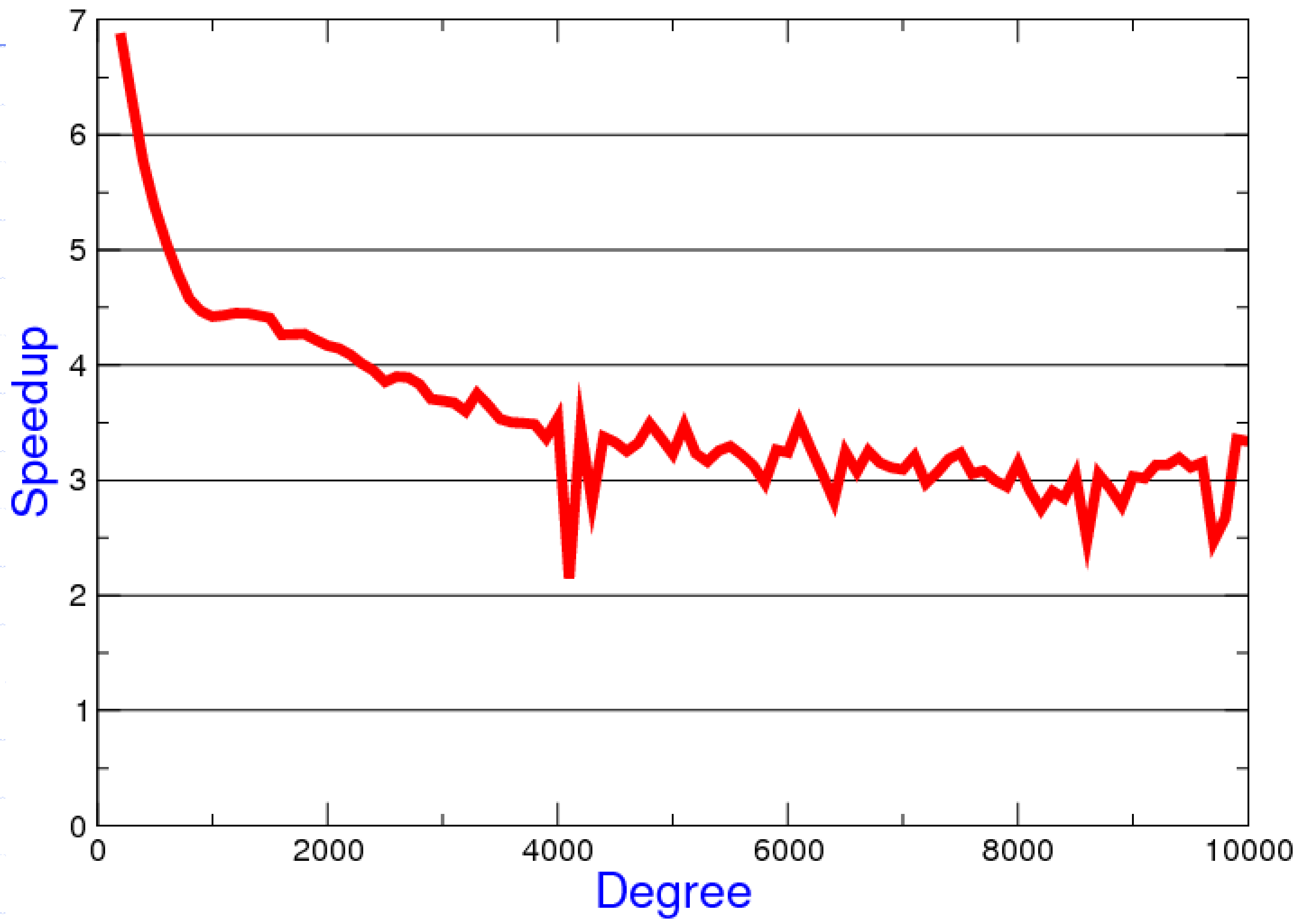
Integer Execution Unit 1
Integer Execution Unit 2
Processor Cycles

- Assist the compiler with scheduling by grouping additions.
- Example pictured is 4x4 register tile.
- The 16 additions consume about 10 cycles on any 2 IEU CPU.
- We did not try scheduling for 3 or more IEU.

Speedup relative to straightforward method



Speedup relative to straightforward method



Automatic code generation and tuning for Taylor Shift computation

- Each register tile computation is defined / influenced by
  - The tile size
  - A number of parallel additions

- Today’s compilers still need to receive fully unrolled code for best performance

Automatic code generation and tuning for Taylor Shift computation

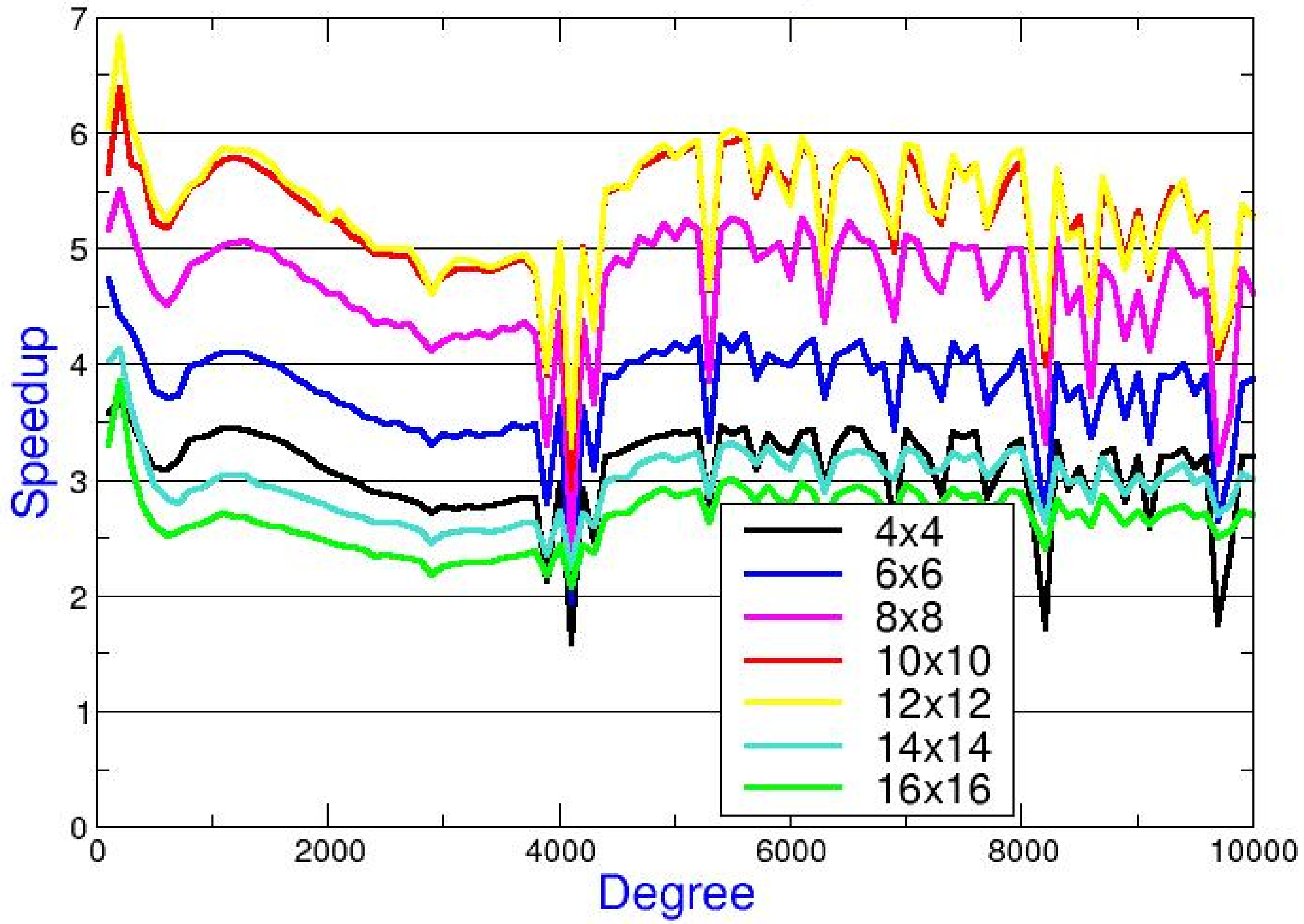
- We wrote Perl-based code generator that
  - Consists of ~ 1000 lines of code
  - Unrolls the loops
  - Uses performance counters for assessment
  - Selects best tile size automatically
- Then we played with the generator!

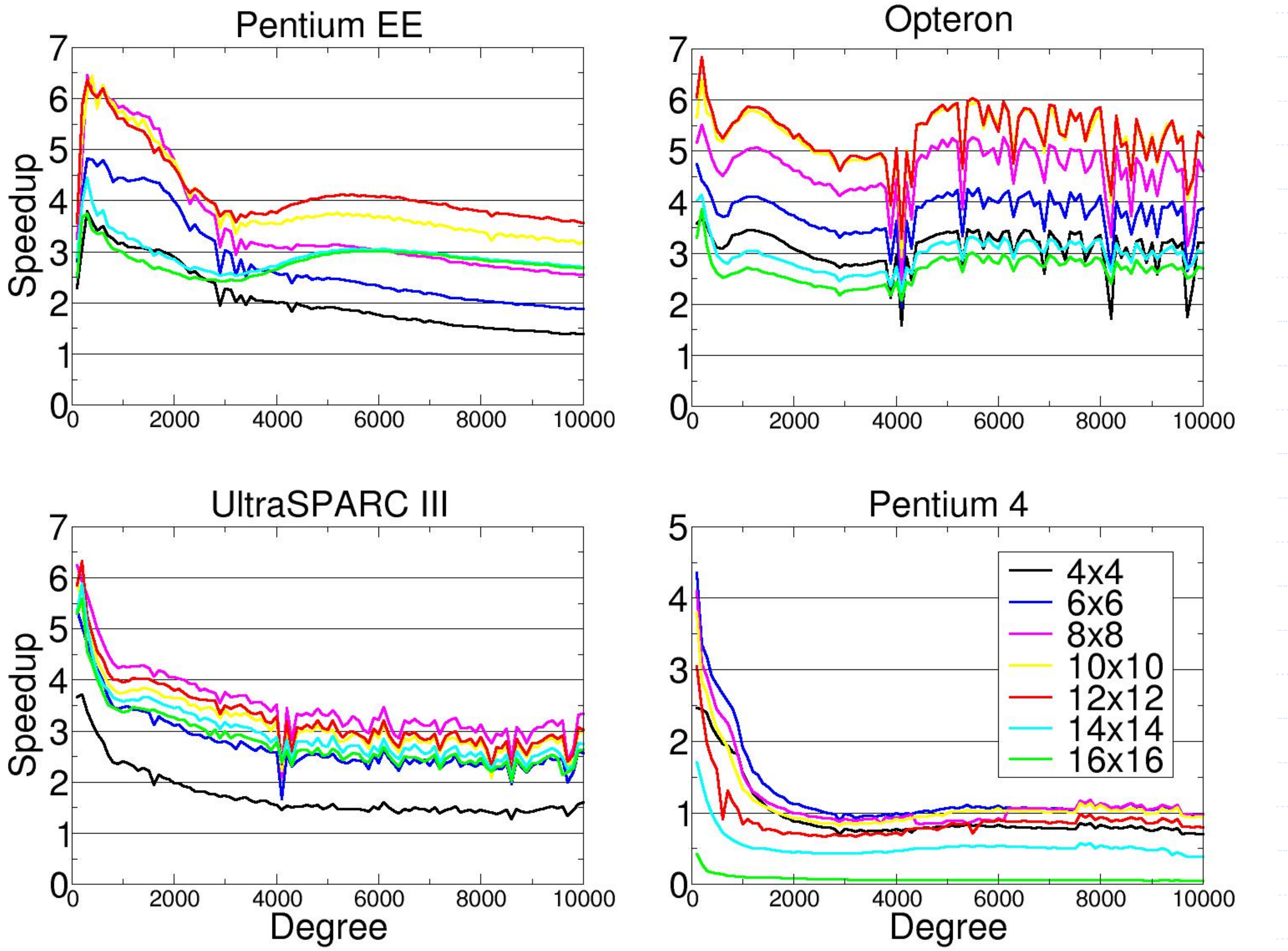
Code generator worked hard!

Square tile size	Lines of code generated
4	1,124
6	1,876
8	3,044
10	4,724
12	7,012
14	10,004
16	13,796
Total	41,580

Impact of register tile size on performance

Architecture: AMD Opteron




Pentium EE
Opteron
UltraSPARC III
Pentium 4
Speedup
Degree
0
1
2
3
4
5
6
7
2000
4000
6000
8000
10000
4x4
6x6
8x8
10x10
12x12
14x14
16x16

Processor architectures

processor	word-length	registers	IEUs	cache assoc.	optimal tile-size
Pentium4	32	8	2x2	8-way	6x6
UltraSPARC III	64	32	2	4-way	8x8
Pentium EE	64	16	2x2	8-way	12x12
Opteron	64	16	3	4-way	12x12

Summary

- Improved performance through automatic code generation and tuning!
  - Modeling is difficult
  - Invent new implementations
  - Spoon-feed the compilers
  - Automatically experiment/test
  - Choose the best!

Thank you! / Merci!

Questions?