Introduction to Optimization

Why study optimizations?

Moore's Law

- Chip density doubles every 18 months
- Often reflected CPU power doubling every 18 months

Proebsting's Law

- Compiler technology doubles CPU power every 18 years
- 4% improvement per year because of optimizations

Corollary

- 1 year of code optimization research = 1 month of hardware improvements
- No need for compiler research… Just wait a few months!

Free Lunch is over

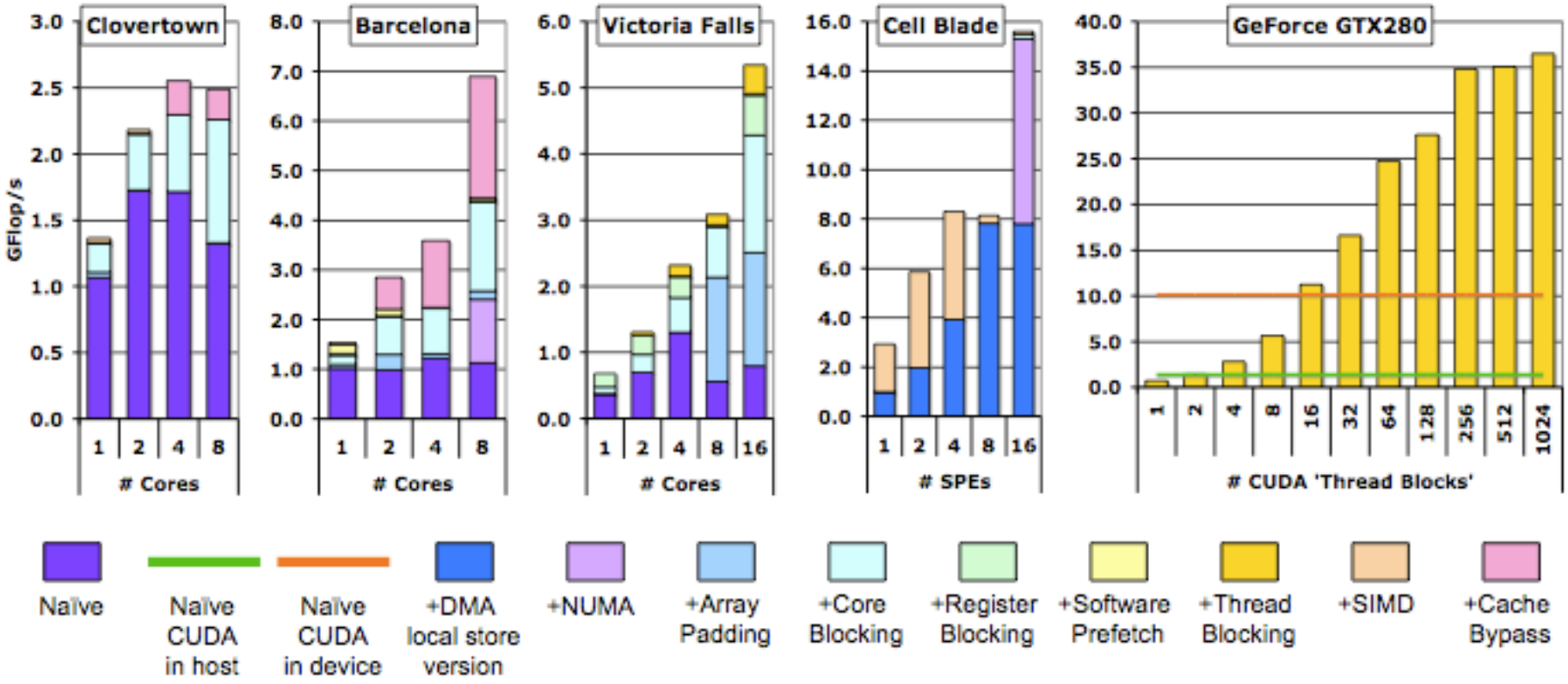
Moore's Law

- Chip density doubles every 18 months
- **PAST :** ~~**Often reflected CPU power doubling every 18 months**~~
- **CURRENT: Density doubling reflected in more cores on chip!**

Corollary

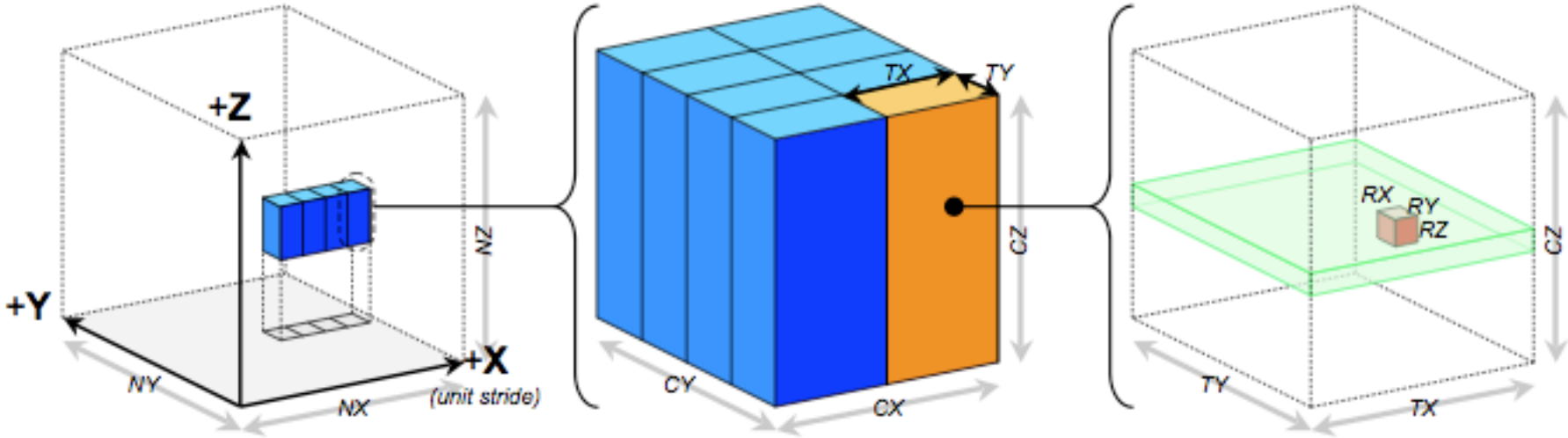
- Cores will become simpler
- Just wait a few months… Your code gets slower!
- Many optimizations now being down by hand!
  - → "autotuning"

Recent Autotuning Results



Slide Source: Datta et al., "Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures" Supercomputing

Recent Autotuning Results



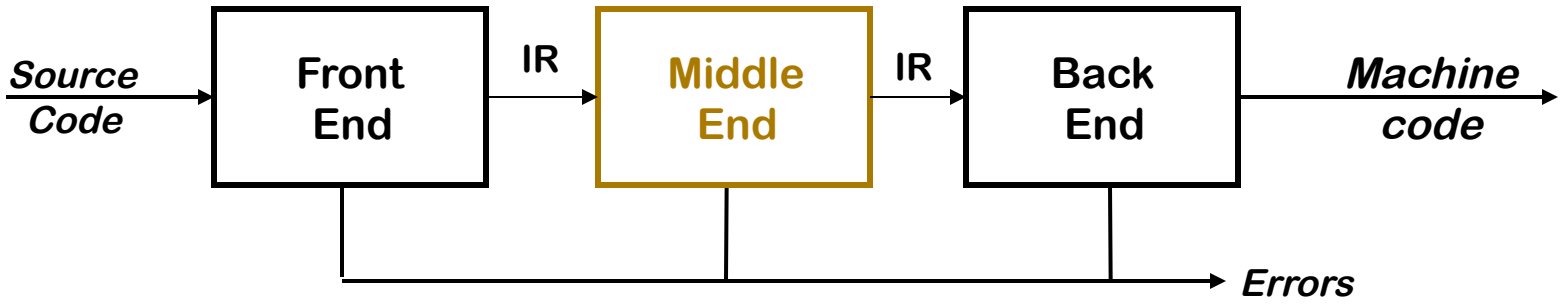
(a)
Decomposition of a Node Block into a Chunk of Core Blocks

(b)
Decomposition into Thread Blocks

(c)
Decomposition into Register Blocks

Slide Source: Datta et al., "Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures" Supercomputing

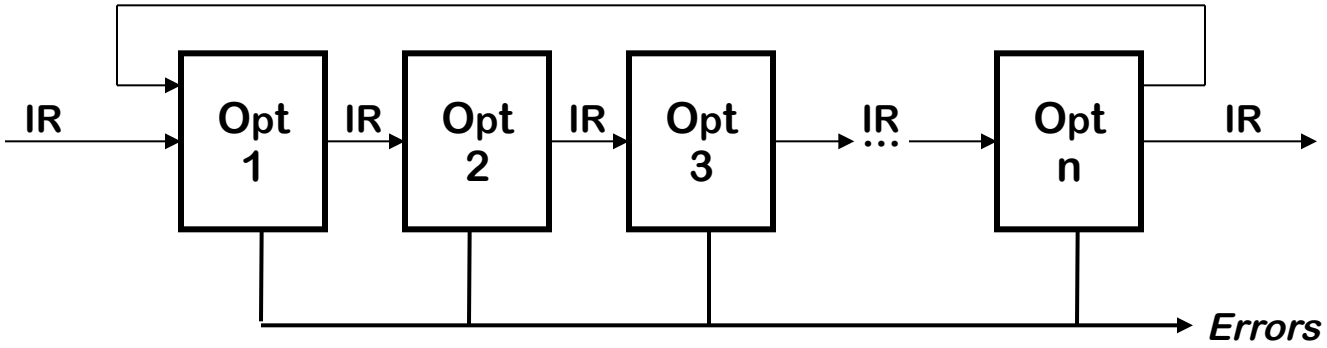
Traditional Three-pass Compiler



Code Improvement (or Optimization)

- Analyzes IR and rewrites (or transforms) IR
- Primary goal is to reduce running time of the compiled code
  - → May also improve space, power consumption, …
- Must preserve "meaning" of the code
  - → Measured by values of named variables
  - → A course (or two) unto itself

The Optimizer (or Middle End)



Modern optimizers are structured as a series of passes

Typical Transformations

- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialize some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form

The Role of the Optimizer

- The compiler can implement a procedure in many ways
- The optimizer tries to find an implementation that is "better"
  - → Speed, code size, data space, …

To accomplish this, it

- Analyzes the code to derive knowledge about run-time behavior
  - → Data-flow analysis, pointer disambiguation, …
  - → General term is "static analysis"
- Uses that knowledge in an attempt to improve the code
  - → Literally hundreds of transformations have been proposed
  - → Large amount of overlap between them

Nothing "optimal" about optimization

- Proofs of optimality assume restrictive & unrealistic conditions

Redundancy Elimination as an Example

An expression x+y is redundant if and only if, along every path from the procedure's entry, it has been evaluated, and its constituent subexpressions (x & y) have not been re-defined.

If the compiler can prove that an expression is redundant

- It can preserve the results of earlier evaluations
- It can replace the current evaluation with a reference

Two pieces to the problem

- Proving that x+y is redundant
- Rewriting the code to eliminate the redundant evaluation

One technique for accomplishing both is called value numbering

Redundant Computation

An example

Original Code

```
    a ← x + y
*   b ← x + y
    a ← 17
*   c ← x + y
```

Value Numbering

The key notion

- Assign an identifying number, V(n), to each expression and operand
  - → V(x+y) = V(j) iff x+y and j have the same value ∀ path
  - → Use hashing over the value numbers to make it efficient
- Use these numbers to replace redundant expressions

Simple extensions to value numbering

- Simplify algebraic identities
- Discover constant-valued expressions, fold & propagate them

Local Value Numbering

The Algorithm

For each operation *o* = <operator, o_1, o_2> in the block

1. Get value numbers for operands from hash lookup
2. Hash <operator,VN(o_1),VN(o_2)> to get a value number for *o*
3. If *o* already had a value number, replace *o* with a reference

Using hashing, the algorithm runs in linear time

Local Value Numbering

An example

Original Code

a ← b + c
b ← a - d
c ← b + c
d ← a - d

How many redundancies:

- Eliminate redundant stmts with references

Local Value Numbering

An example

Original Code	With VNs	Rewritten
$a \leftarrow b + c$	$a^3 \leftarrow b^1 + c^2$	$a^3 \leftarrow b^1 + c^2$
$b \leftarrow a - d$	$b^5 \leftarrow a^3 - d^4$	$b^5 \leftarrow a^3 - d^4$
$c \leftarrow b + c$	$c^6 \leftarrow b^5 + c^2$	$c^6 \leftarrow b^5 + c^2$
* $d \leftarrow a - d$	* $d^5 \leftarrow a^3 - d^4$	* $d^3 \leftarrow b^5$

How many redundancies:

- Eliminate redundant stmts with references

Local Value Numbering

An example

Original Code	With VNs	Rewritten
$a \leftarrow x + y$	$a^3 \leftarrow x^1 + y^2$	$a^3 \leftarrow x^1 + y^2$
* $b \leftarrow x + y$	* $b^3 \leftarrow x^1 + y^2$	* $b^3 \leftarrow a^3$
$a \leftarrow 17$	$a^4 \leftarrow 17$	$a^4 \leftarrow 17$
* $c \leftarrow x + y$	* $c^3 \leftarrow x^1 + y^2$	* $c^3 \leftarrow a^3$ (oops!)

Two redundancies:

- Eliminate stmts with a *
- Coalesce results ?

Options:

- Use $c^3 \leftarrow b^3$
- Save a^3 in t^3
- Rename around it

Local Value Numbering

Example (continued)

Original Code	**With VNs**	**Rewritten**
$a_0 \leftarrow x_0 + y_0$	$a_0^3 \leftarrow x_0^1 + y_0^2$	$a_0^3 \leftarrow x_0^1 + y_0^2$
* $b_0 \leftarrow x_0 + y_0$	* $b_0^3 \leftarrow x_0^1 + y_0^2$	* $b_0^3 \leftarrow a_0^3$
$a_1 \leftarrow 17$	$a_1^4 \leftarrow 17$	$a_1^4 \leftarrow 17$
* $c_0 \leftarrow x_0 + y_0$	* $c_0^3 \leftarrow x_0^1 + y_0^2$	* $c_0^3 \leftarrow a_0^3$

Renaming:

- Give each value a unique name
- Makes it clear

Notation:

- While complex, the meaning is clear

Result:

- a_0^3 is available
- Rewriting just works

Simple Extensions to Value Numbering

Constant folding

- Add a bit that records when a value is constant
- Evaluate constant values at compile-time
- Replace with load immediate or immediate operand

Algebraic identities

- Must check (many) special cases
- Replace result with input VN

Identities:

x←y, x+0, x-0, x*1, x÷1, x-x, x*0, x÷x

max(x,MAXINT), min(x,MININT), max(x,x), min(y,y), and so on ...

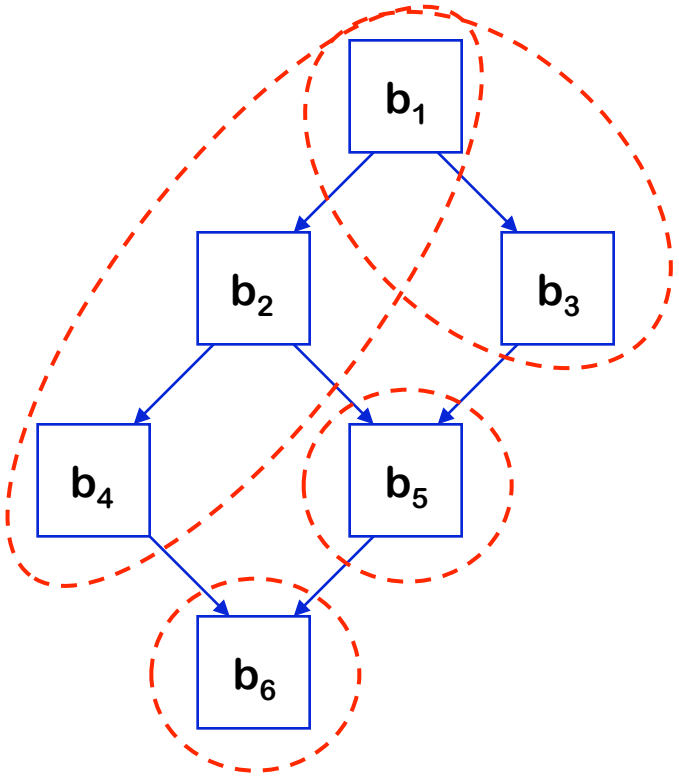
With values, not names

Handling Larger Scopes

Extended Basic Blocks

- Initialize table for b_i with table from b_{i-1}
- With single-assignment naming, can use scoped hash table

Otherwise, it is complex



The Plan:

→ Process b_1, b_2, b_4
Pop two levels
→ Process b_3 relative to b_1
→ Start clean with b_5
→ Start clean with b_6