Introduction to Scheduling Under Memory Constraints and Pebble Game Models

Grégoire Pichon, Bora Uçar & Frédéric Vivien

(Original slides by Loris Marchal)

CNRS, INRIA, Université Lyon 1 & ENS Lyon

CR15: Januray 2023

https://gpichon.gitlabpages.inria.fr/m2if-numerical_algorithms/

Part 1: Introduction and Pebble Game models

Introduction & Motivation

- (Fast) Memory: place to store data for computation
- Always been a limited resource (4kB in Apollo 11 computer)
- Not limited anymore?
  (last iPhone: $\geq$ 64GB, workstation: $\approx$ 1TB)

Introduction & Motivation

- (Fast) Memory: place to store data for computation
- Always been a limited resource (4kB in Apollo 11 computer)
- Not limited anymore?
  (last iPhone: $\geq$ 64GB, workstation: $\approx$ 1TB)
- But problem size always gets bigger...

Introduction & Motivation

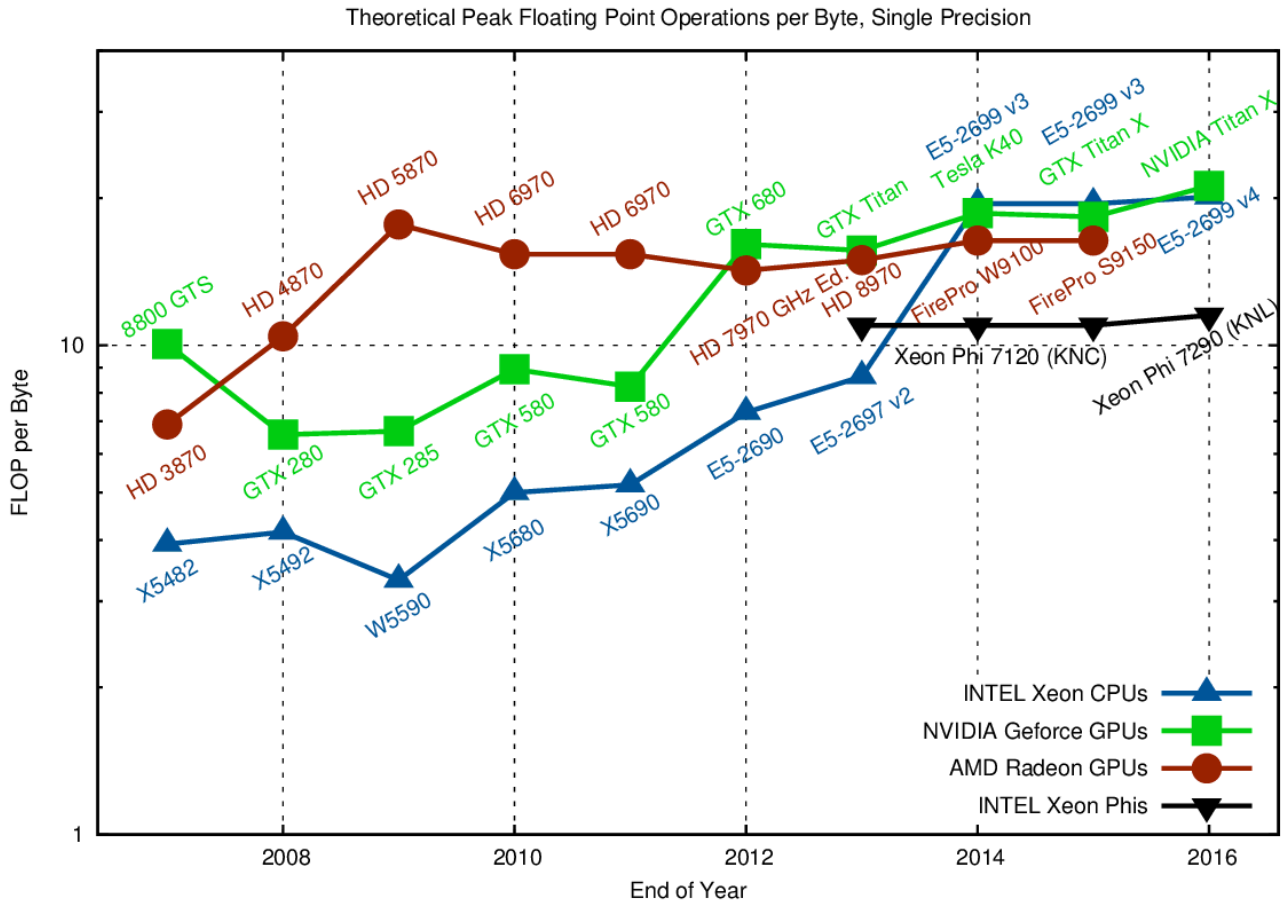
- (Fast) Memory: place to store data for computation
- Always been a limited resource (4kB in Apollo 11 computer)
- Not limited anymore?
  (last iPhone: $\geq$ 64GB, workstation: $\approx$ 1TB)
- But problem size always gets bigger...

  ...And the problem is rather a question of speed!
- Annual improvements:
  - Number of flops per chip (computation): 59%
  - Data movement:

	Bandwidth	Latency
Network	26%	15%
DRAM	23%	5%

Figures from *Getting up to speed: The future of supercomputing*, 2005, National Academies Press (2004 figure based on data on the period 1988-2002)

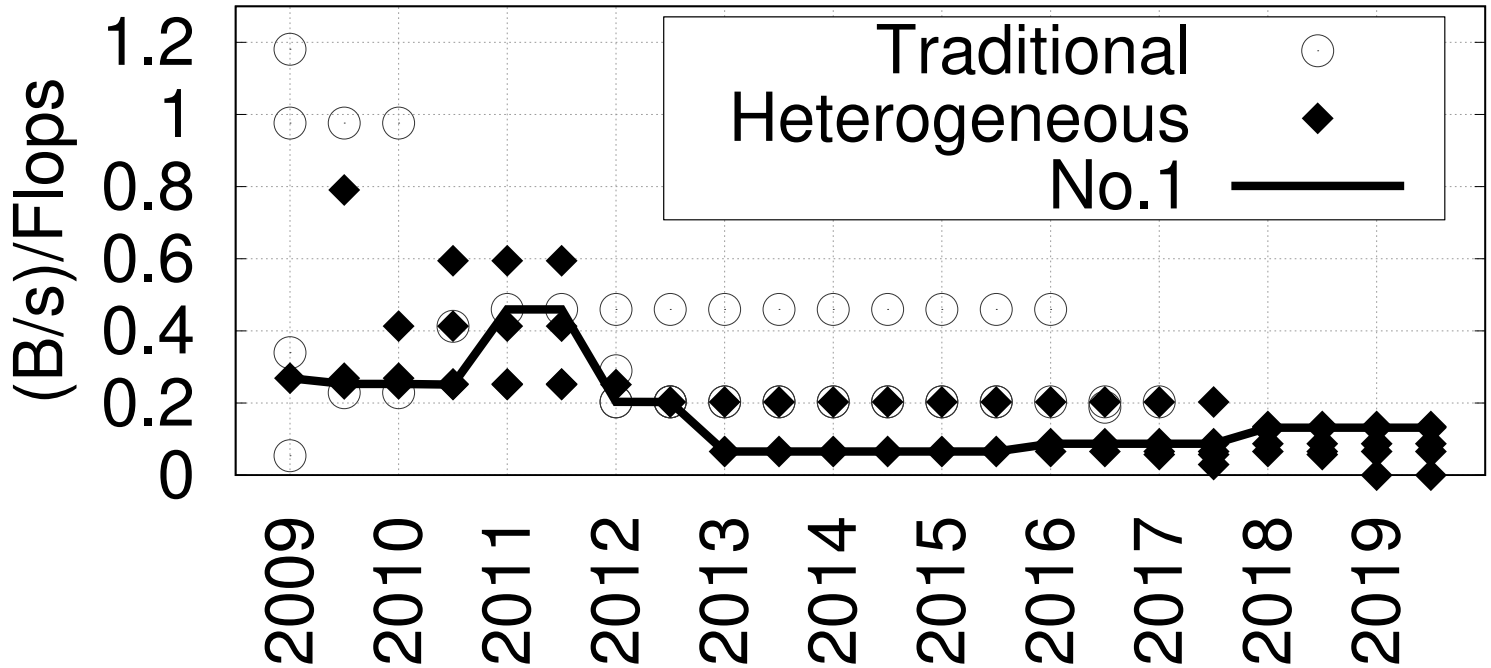
Ratio of flops per byte moved



- number of flops perform in the time needed to move a byte
- $\frac{\text{computing speed}}{\text{communication speed}}$

From `http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/`

Performance balance



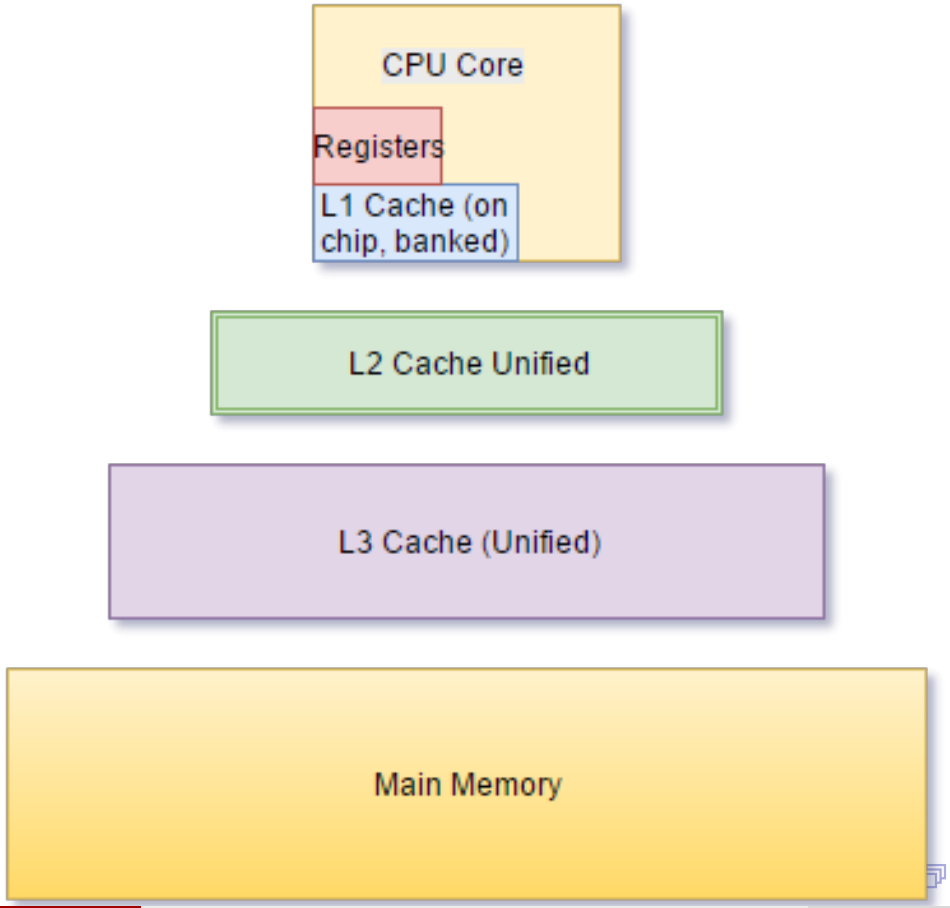
Performance balance between the aggregate memory bandwidth and the peak processing power of supercomputers from the Top500 list

From "An Analysis of System Balance and Architectural Trends Based on Top500 Supercomputers"

`https://people.cs.vt.edu/~butta/docs/HPCAsia2021-top500.pdf`

Bypass the memory wall

- Time to move the data $>$ Time to compute on the data
- Similar problem in microprocessor design: "memory wall"
- Traditional workaround: add a faster but smaller "cache" memory
- Now a hierarchy of caches !



Computing with caches

- Limited amount of fast cache
- Performance sensitive to data locality
- Optimize data reuse
- Avoid data movements between memory and cache(s) (time- and energy-consuming)

Part 1: Introduction and Pebble Game models

Example: matrix-matrix product

- Consider two square matrices A and B (size $n \times n$)
- Compute generalized matrix product: $C \leftarrow C + AB$

Simple-Matrix-Multiply(n, C, A, B)
for $i = 0 \rightarrow n - 1$ **do**
  for $j = 0 \rightarrow n - 1$ **do**
    for $k = 0 \rightarrow n - 1$ **do**
      $C_{i,j} = C_{i,j} + A_{i,k} B_{k,j}$

Assume a simple two-level memory model:

- Slow but infinite disk storage
  (where A and B are originally stored)
- Fast and limited memory (size M)

Objective: limit data movement between disk/memory

NB: also applies to other two-level systems (memory/cache, etc.)

Simple algorithm analysis

Simple-Matrix-Multiply(n, C, A, B)

for $i = 0 \rightarrow n - 1$ **do**

  for $j = 0 \rightarrow n - 1$ **do**

    for $k = 0 \rightarrow n - 1$ **do**

      $C_{i,j} = C_{i,j} + A_{i,k} B_{k,j}$

- Assume the memory cannot store half of a matrix: $M < n^2/2$
- Question: How many data movement in this algorithm?

Simple algorithm analysis

Simple-Matrix-Multiply(n, C, A, B)
for $i = 0 \rightarrow n-1$ **do**
  for $j = 0 \rightarrow n-1$ **do**
    for $k = 0 \rightarrow n-1$ **do**
      $C_{i,j} = C_{i,j} + A_{i,k} B_{k,j}$

- Assume the memory cannot store half of a matrix: $M < n^2/2$
- Question: How many data movement in this algorithm?

Answer:

- All elements of B accessed during one iteration of the outer loop
- At most half of B stays in memory
- At least $n^2/2$ elements must be read per iteration of the outer loop
- At least $n^3/2$ read for the entire algorithm
- Same order of magnitude as computations: $\Omega(n^3)$
- Very bad data reuse ☹ Question: Can we do better? How?

Blocked matrix-matrix product

- Divide each matrix into blocks of size $b \times b$:
  $A^b_{i,k}$ is the block of A at position (i, k)
- Perform "coarse-grain" matrix product on blocks
- Perform each block product with previous algorithms

Blocked-Matrix-Multiply(n,A,B,C)
$b \leftarrow \sqrt{M/3}$
for $i = 0 \rightarrow n/b - 1$ **do**
  for $j = 0 \rightarrow n/b - 1$ **do**
    for $k = 0 \rightarrow n/b - 1$ **do**
      Simple-Matrix-Multiply$(n, C^b_{i,j}, A^b_{i,k}, B^b_{k,j})$

Blocked matrix-matrix product – Analysis

Blocked-Matrix-Multiply(n,A,B,C)
$b \leftarrow \sqrt{M/3}$
for $i = 0, \rightarrow n/b - 1$ **do**
  for $j = 0, \rightarrow n/b - 1$ **do**
    for $k = 0, \rightarrow n/b - 1$ **do**
      Simple-Matrix-Multiply(n, $C^b_{i,j}$, $A^b_{i,k}$, $B^b_{k,j}$)

Question: Number of data movements?

Blocked matrix-matrix product – Analysis

Blocked-Matrix-Multiply(n,A,B,C)
$b \leftarrow \sqrt{M/3}$
for $i = 0, \rightarrow n/b - 1$ **do**
  for $j = 0, \rightarrow n/b - 1$ **do**
    for $k = 0, \rightarrow n/b - 1$ **do**
      Simple-Matrix-Multiply(n, $C^b_{i,j}$, $A^b_{i,k}$, $B^b_{k,j}$)

Question: Number of data movements?

- Iteration of inner loop: 3 blocks of size $b \times b = \sqrt{M/3}^3 = M/3$ $\rightarrow$ fits in memory
- At most $M + M/3$ ($O(M)$) data movements for each inner loop (reading/writing)
- Number of inner iterations: $(n/b)^3 = \left(n/\sqrt{M/3}\right)^3 = O(n^3/M\sqrt{M})$
- Total number of data movements: $O(n^3/\sqrt{M})$

Blocked matrix-matrix product – Analysis

Blocked-Matrix-Multiply(n,A,B,C)
$b \leftarrow \sqrt{M/3}$
for $i = 0, \rightarrow n/b - 1$ **do**
  for $j = 0, \rightarrow n/b - 1$ **do**
    for $k = 0, \rightarrow n/b - 1$ **do**
      Simple-Matrix-Multiply(n, $C^b_{i,j}$, $A^b_{i,k}$, $B^b_{k,j}$)

Question: Number of data movements?

- Iteration of inner loop: 3 blocks of size $b \times b = \sqrt{M/3}^3 = M/3$ $\rightarrow$ fits in memory
- At most $M + M/3$ ($O(M)$) data movements for each inner loop (reading/writing)
- Number of inner iterations: $(n/b)^3 = \left(n/\sqrt{M/3}\right)^3 = O(n^3/M\sqrt{M})$
- Total number of data movements: $O(n^3/\sqrt{M})$

Question: Can we do (significantly) better?

Blocked matrix-matrix product – Analysis

Blocked-Matrix-Multiply(n,A,B,C)
$b \leftarrow \sqrt{M/3}$
for $i = 0, \rightarrow n/b - 1$ **do**
  for $j = 0, \rightarrow n/b - 1$ **do**
    for $k = 0, \rightarrow n/b - 1$ **do**
      Simple-Matrix-Multiply(n, $C^b_{i,j}$, $A^b_{i,k}$, $B^b_{k,j}$)

Question: Number of data movements?

- Iteration of inner loop: 3 blocks of size $b \times b = \sqrt{M/3}^3 = M/3$ $\rightarrow$ fits in memory
- At most $M + M/3$ ($O(M)$) data movements for each inner loop (reading/writing)
- Number of inner iterations: $(n/b)^3 = \left(n/\sqrt{M/3}\right)^3 = O(n^3/M\sqrt{M})$
- Total number of data movements: $O(n^3/\sqrt{M})$

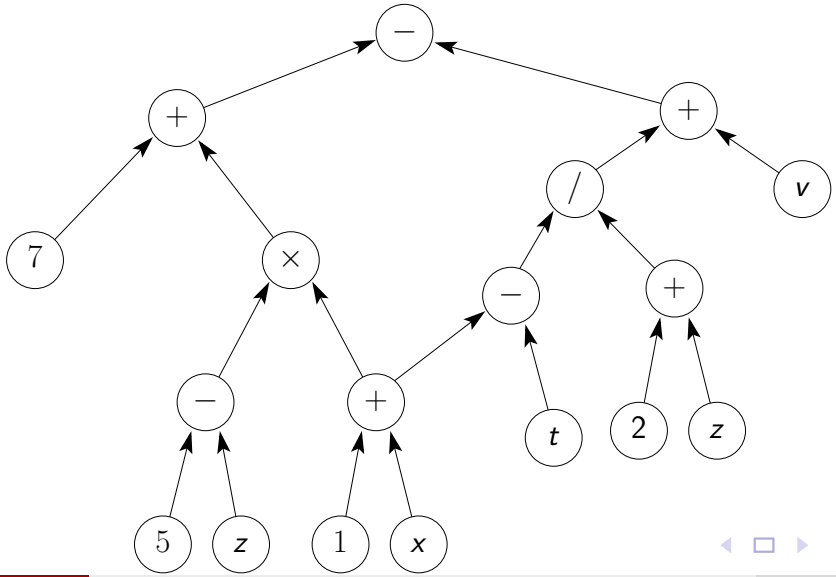
Question: Can we do (significantly) better? Answer: next lesson!

Part 1: Introduction and Pebble Game models

Pebble Game and Register Minimization

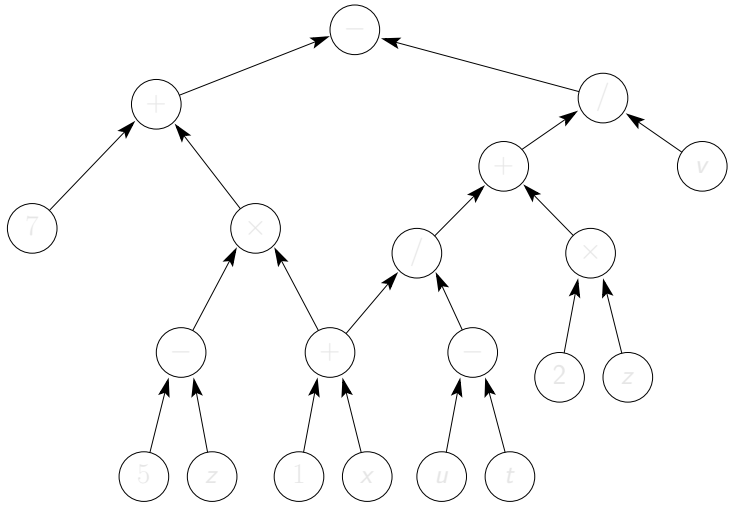
- First model introduced in the 70s
- Motivation: limit the usage of registers for a computation (scarce resource, typically 16/32 for CPUs)
- Registers: at the top of the memory hierarchy
- Restrict to straight-line program:
  control flow independent from input data
- Modeled as Directed Acyclic Graph:

$$7 + (5 - z) \times (1 + x) - ((1 + x - t)/(2 + z) + v)$$



(Black) Pebble Game – Rules

1. A pebble may be removed from a vertex at any time.
2. A pebble may be placed on a source node at any time.
3. If all (direct) predecessors of an unpebbled vertex v are pebbled, a pebble may be placed on v.

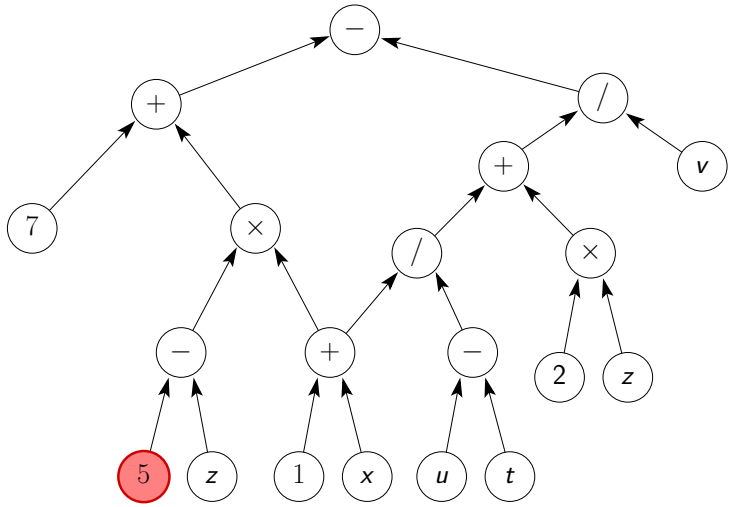


Analogy with register allocation:

- Rule 1: Load in register
- Rule 2: Compute new value in a register
- Objective: Pebble all vertices at least once using a limited number of pebbles

(Black) Pebble Game – Rules

1. A pebble may be removed from a vertex at any time.
2. A pebble may be placed on a source node at any time.
3. If all (direct) predecessors of an unpebbled vertex v are pebbled, a pebble may be placed on v.



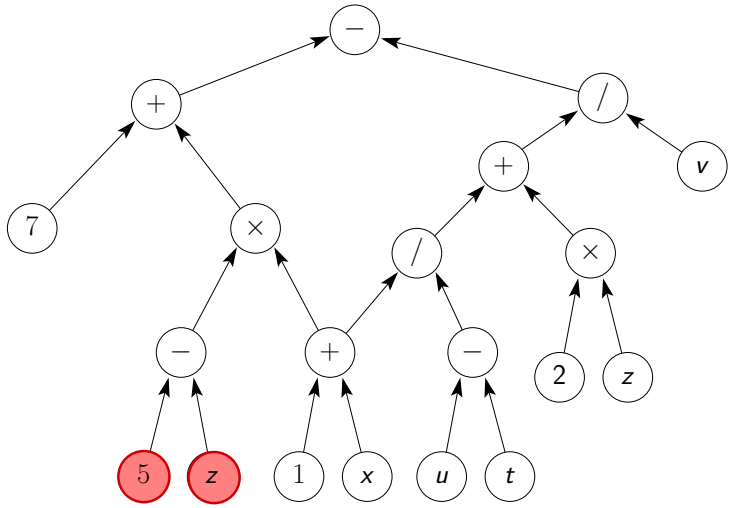
Analogy with register allocation:

- Rule 2: Load in register
- Rule 3: Compute new value (in new register)

Objective: Pebble all vertices at least once using a minimum number of pebbles

(Black) Pebble Game – Rules

1. A pebble may be removed from a vertex at any time.
2. A pebble may be placed on a source node at any time.
3. If all (direct) predecessors of an unpebbled vertex v are pebbled, a pebble may be placed on v.



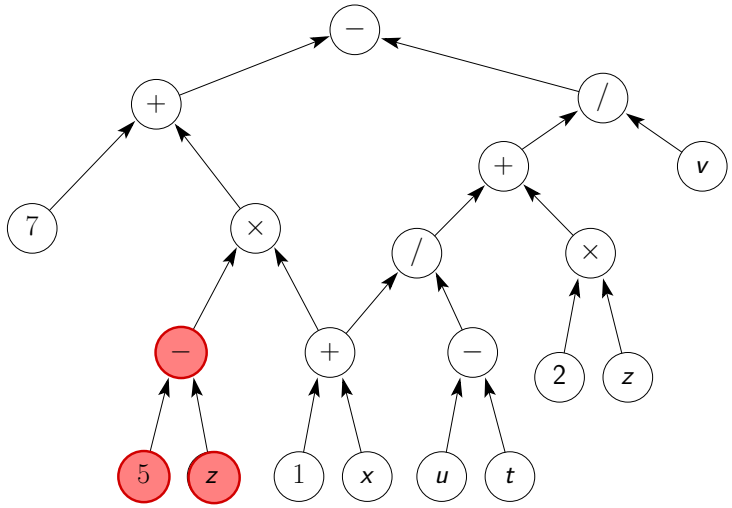
Analogy with register allocation:

- Rule 2: Load in register
- Rule 3: Compute new value (in new register)

Objective: Pebble all vertices at least once using a minimum number of pebbles

(Black) Pebble Game – Rules

1. A pebble may be removed from a vertex at any time.
2. A pebble may be placed on a source node at any time.
3. If all (direct) predecessors of an unpebbled vertex v are pebbled, a pebble may be placed on v.



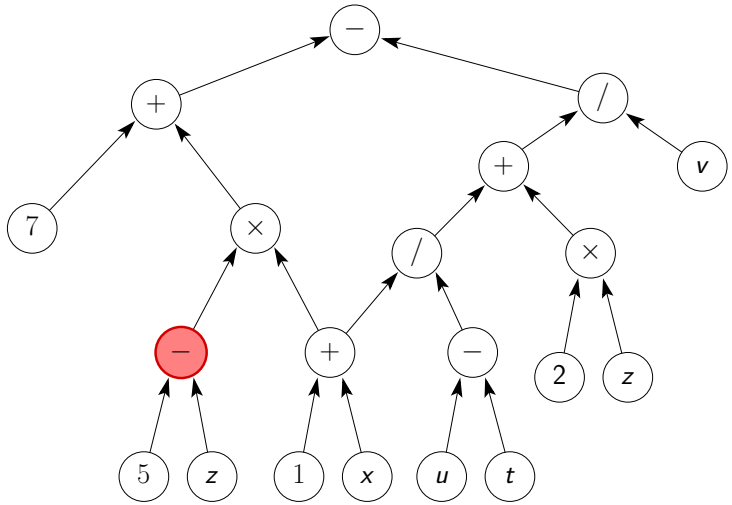
Analogy with register allocation:

- Rule 2: Load in register
- Rule 3: Compute new value (in new register)

Objective: Pebble all vertices at least once using a minimum number of pebbles

(Black) Pebble Game – Rules

1. A pebble may be removed from a vertex at any time.
2. A pebble may be placed on a source node at any time.
3. If all (direct) predecessors of an unpebbled vertex v are pebbled, a pebble may be placed on v.



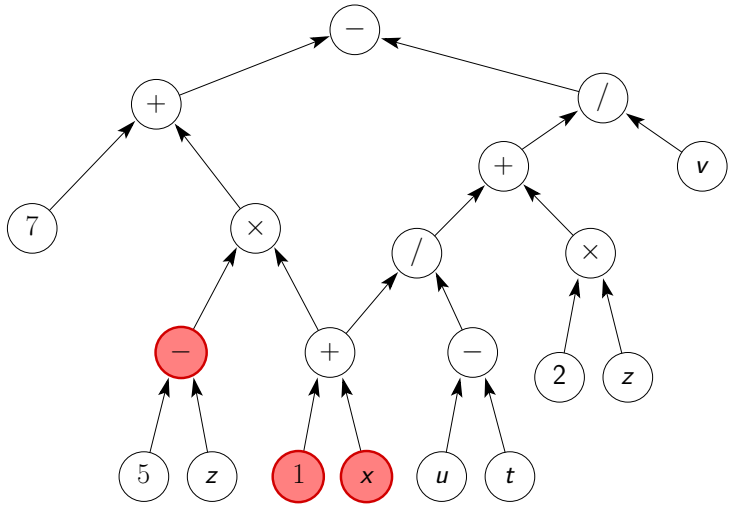
Analogy with register allocation:

- Rule 2: Load in register
- Rule 3: Compute new value (in new register)

Objective: Pebble all vertices at least once using a minimum number of pebbles

(Black) Pebble Game – Rules

1. A pebble may be removed from a vertex at any time.
2. A pebble may be placed on a source node at any time.
3. If all (direct) predecessors of an unpebbled vertex v are pebbled, a pebble may be placed on v.



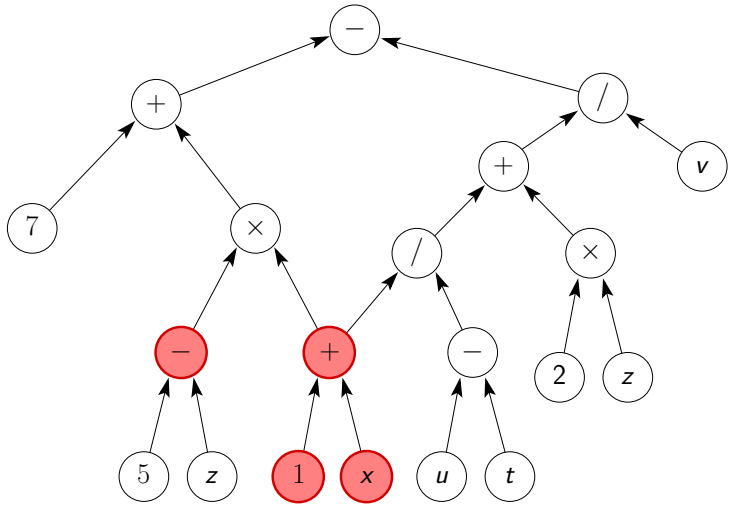
Analogy with register allocation:

- Rule 2: Load in register
- Rule 3: Compute new value (in new register)

Objective: Pebble all vertices at least once using a minimum number of pebbles

(Black) Pebble Game – Rules

1. A pebble may be removed from a vertex at any time.
2. A pebble may be placed on a source node at any time.
3. If all (direct) predecessors of an unpebbled vertex v are pebbled, a pebble may be placed on v.



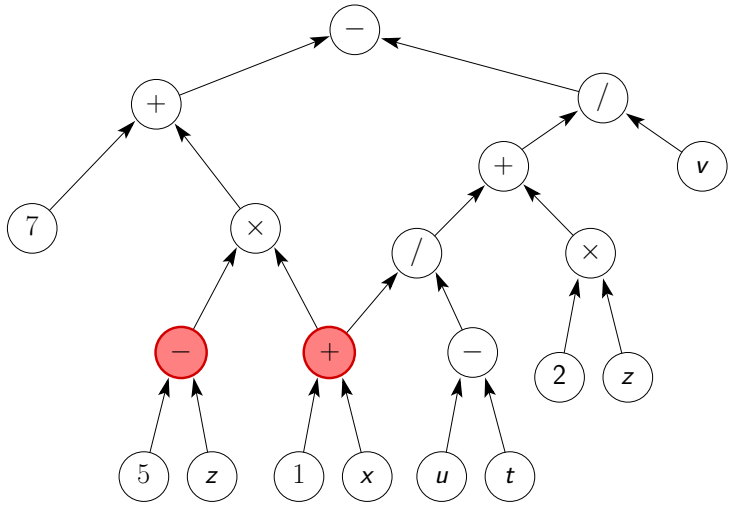
Analogy with register allocation:

- Rule 2: Load in register
- Rule 3: Compute new value (in new register)

Objective: Pebble all vertices at least once using a minimum number of pebbles

(Black) Pebble Game – Rules

1. A pebble may be removed from a vertex at any time.
2. A pebble may be placed on a source node at any time.
3. If all (direct) predecessors of an unpebbled vertex v are pebbled, a pebble may be placed on v.



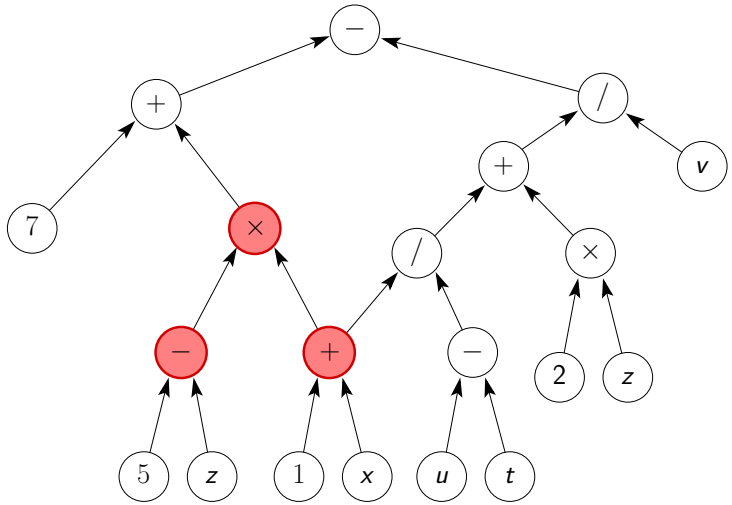
Analogy with register allocation:

- Rule 2: Load in register
- Rule 3: Compute new value (in new register)

Objective: Pebble all vertices at least once using a minimum number of pebbles

(Black) Pebble Game – Rules

1. A pebble may be removed from a vertex at any time.
2. A pebble may be placed on a source node at any time.
3. If all (direct) predecessors of an unpebbled vertex v are pebbled, a pebble may be placed on v.



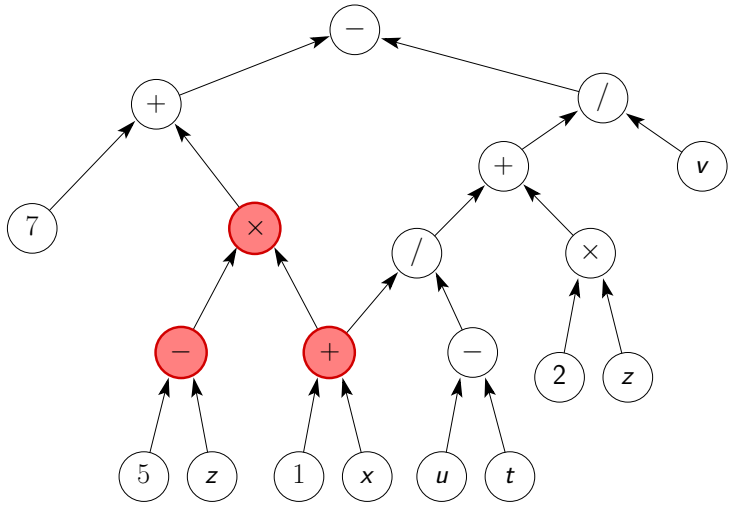
Analogy with register allocation:

- Rule 2: Load in register
- Rule 3: Compute new value (in new register)

Objective: Pebble all vertices at least once using a minimum number of pebbles

(Black) Pebble Game – Rules

1. A pebble may be removed from a vertex at any time.
2. A pebble may be placed on a source node at any time.
3. If all (direct) predecessors of an unpebbled vertex v are pebbled, a pebble may be placed on v.



Analogy with register allocation:

- Rule 2: Load in register
- Rule 3: Compute new value (in new register)

Objective: Pebble all vertices at least once using a minimum number of pebbles

(Black) Pebble Game – Complexity and variants

Progressive pebble game:

- Forbid pebbling twice the same vertex
- NP-Hard

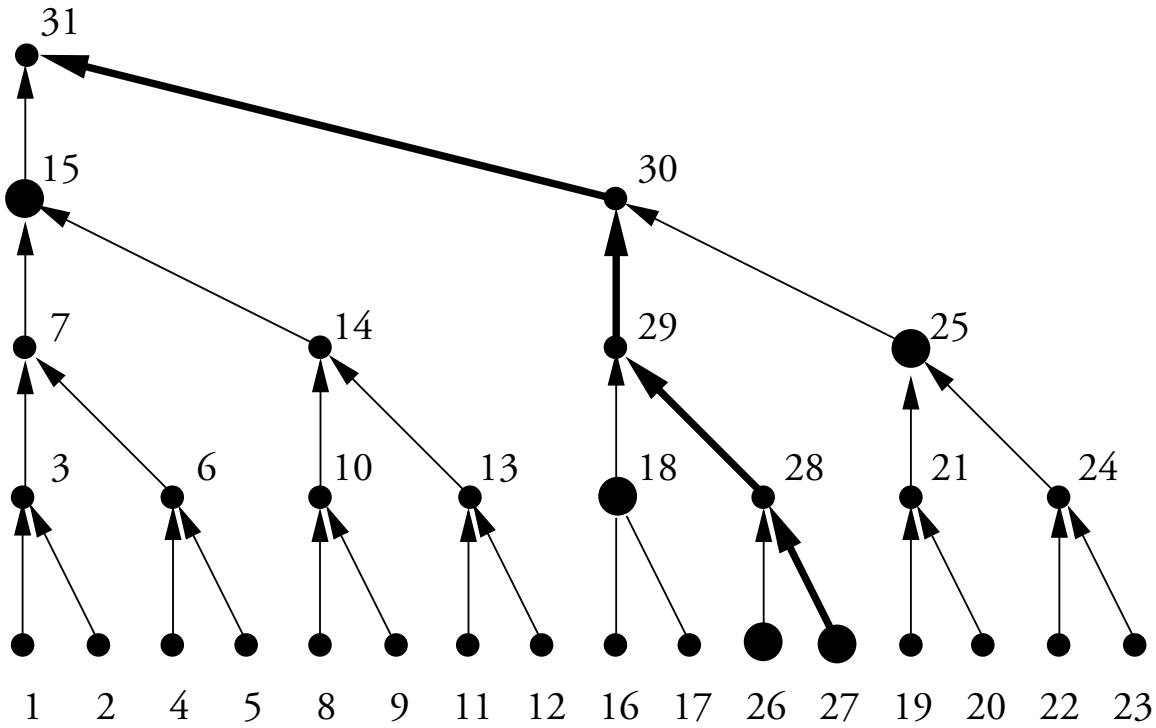
More general problem with re-computation:

- PSpace-complete

Variant with pebble shifting:

- Rule 3 $\rightarrow$ If all predecessors of an unpebbled vertex v are pebbled, a pebble may be shifted from a predecessor to v.
- Decrease the minimum number of pebbles required for a graph by at most one (may induce large number of recomputations)

Complete binary tree of depth $k = 4$



Theorem

Any pebbling strategy (with or without recomputation, without shifting) for the complete balanced binary tree of depth $k \geq 1$ uses at least $k + 2$ pebbles and $2^{k+1} - 1$ steps. There exists a strategy reaching both bounds.

NB: All proofs on the board will be made available online.

General trees

Lemma

Depth-First Traversal are dominant

Depth-First: Totally pebble a subtree, remove all pebbles except on its root, before starting a sibling subtree.

Theorem

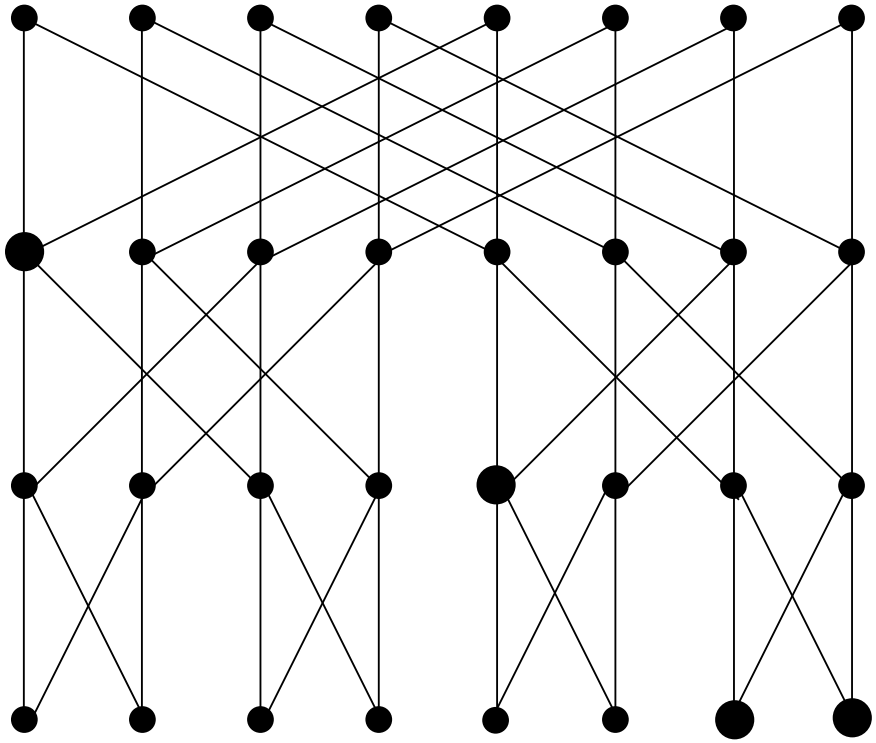
An optimal solution is obtained by ordering subtrees by non-increasing value of $P(i)$, where the peak $P(v)$ of the subtree rooted at v is recursively defined by:

$$P(v) = \begin{cases} 1 & \text{if } v \text{ is a leaf} \\ \max(k+1, \max_{i=1\ldots k} P(c_i) + i - 1) & \end{cases}$$

where $c_1, \ldots c_k$ are the children of v ordered such that $P(c_1) \geq P(c_2) \geq \cdots \geq P(c_k)$.

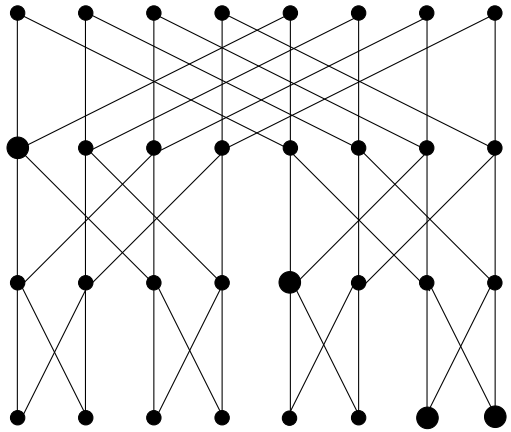
Space-Time tradeoffs – FFT example

- Fast-Fourrier Transform
- Recursive graph based on the "exchange graph" with 2 inputs and 2 outputs



FFT graph with 8 input/output vertices (depth $k = 3$)
$n = 2^k$ vertices at each level

Space-Time tradeoffs – FFT example



Strategy 1:

- Pebble one tree up to one output, then start over
  (variant: pebble two outputs before re-starting)
- Uses $k + 2$ pebbles
  (minimum value since it contains binary tree of depth k)
- Large number of recomputations

Strategy 2:

- Pebble level by level
- Requires $2n = 2^{k+1}$ pebbles
- No recomputations (minimum number of steps)