# PERFORMANCE ENGINEERING

## Lecture 5: Analytical modeling wrap-up and some examples

April 28th, 2022

Ana Lucia Varbanescu

a.l.varbanescu@uva.nl

# TODO list

- Wrap-up analytical modeling
  - Calibration continued
    - Machine model revisited
  - Validation

- Examples of analytical models
  - Heterogeneous computing
  - GPU computing
  - The ECM model

# Recap analytical modeling

- Capture the behavior of the system as a function of problem and machine parameters.
- Advantages:
  - Easy to compute/apply
  - Insight into the system's inner working ("white-box" model)
- Disadvantages
  - Difficult to design
  - Difficult to calibrate
  - Difficult to capture dynamic behavior (e.g., data-dependent performance)
    - Usually fixed through probabilistic models for different parameters

# Analytical modeling in practice

- Challenges
  - At what level to model ?
    - From blocks of instructions to micro-operations
  - How to calibrate ?
    - From own benchmarks to microbenchmarks
  - How to include machine features ?
    - Typically microbenchmarking + understanding the hardware
      - Hardware-level parallelism
      - Instruction scheduling
  - How to validate/test the model?
    - Empirical analysis: modeled vs observed
      - Accuracy relates to the metric of interest!

# Instructions or μops?

- All instructions are “decomposed” into micro-operations

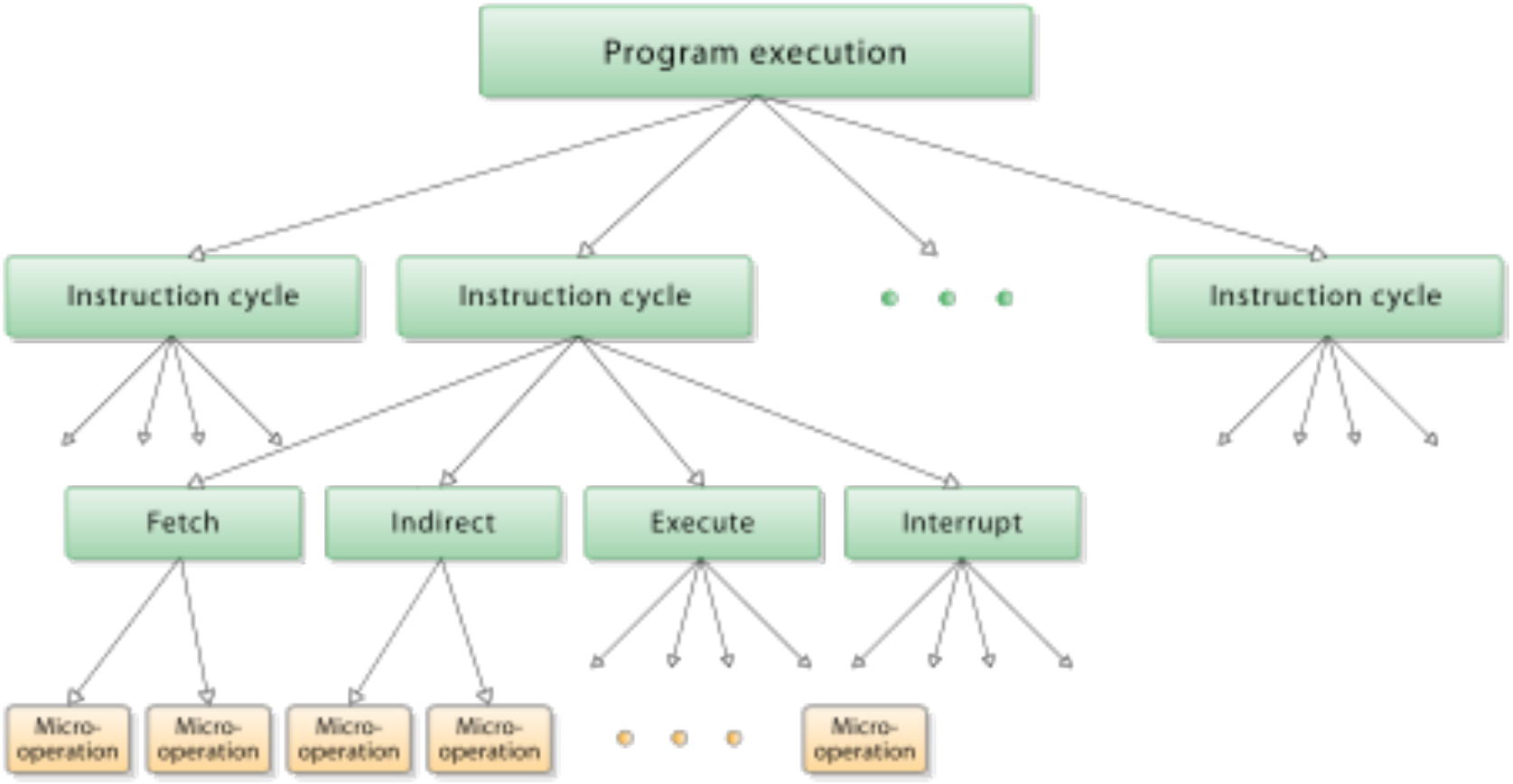
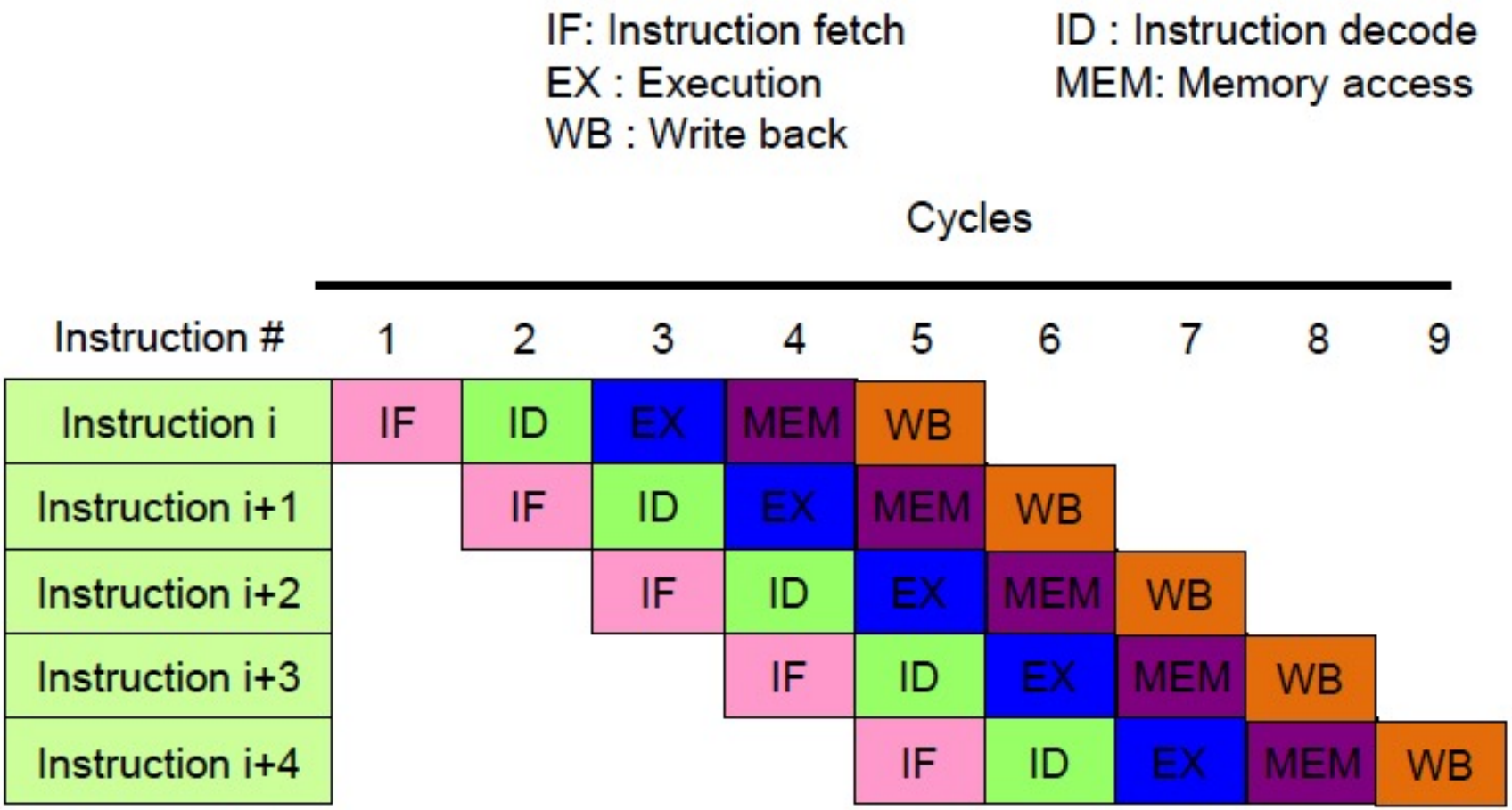



Image courtesy of Wikipedia

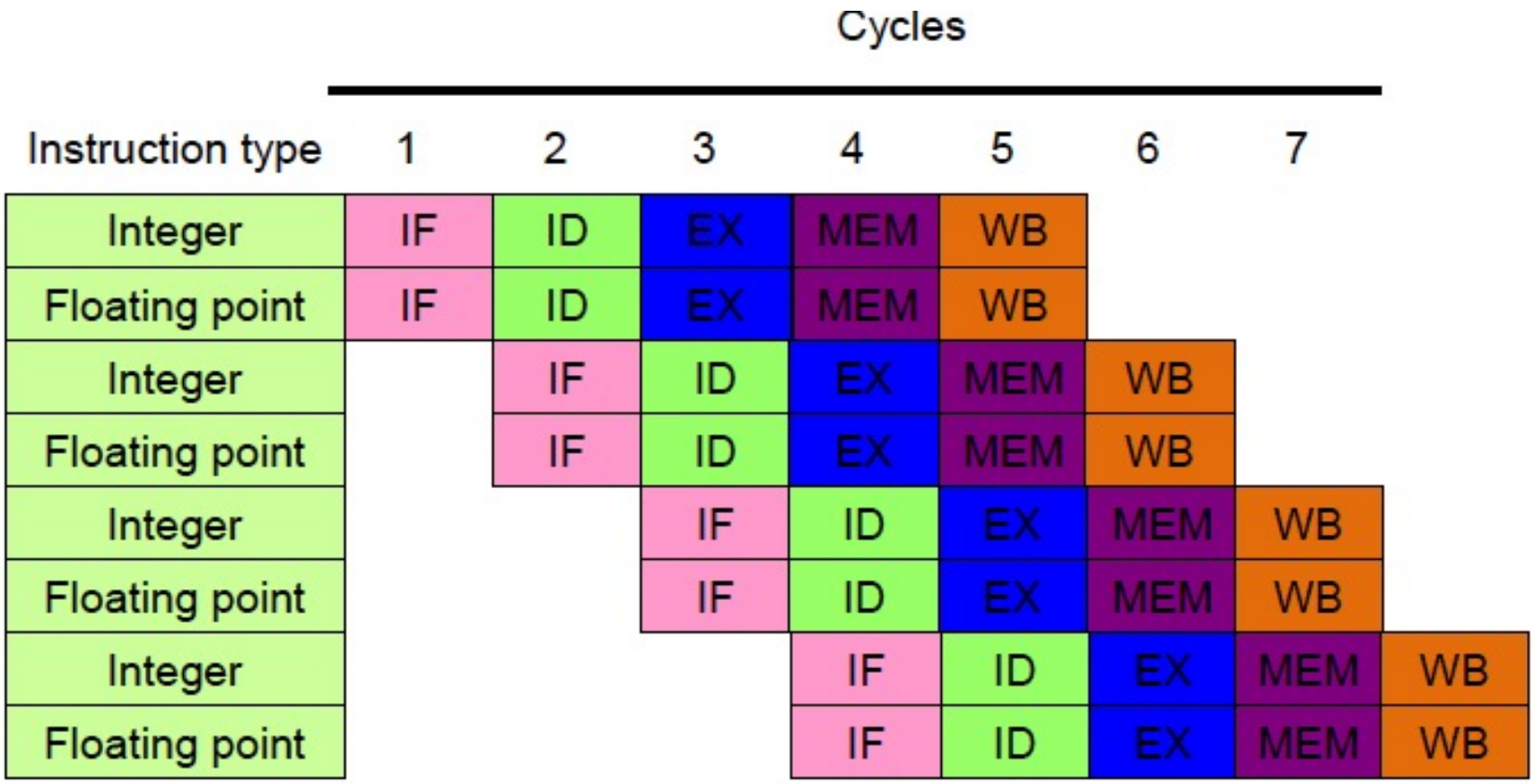
# Complex machine features (for ILP)

- Instruction pipelining
  - Multiple instructions “in-flight” in different stages
- Superscalar execution
  - Multiple execution units
  - Multiple instructions “in-flight” in the same stage
- Out-of-order execution
  - Any order that does not violate data dependencies
  - Useful for keeping execution units busy
- Branch prediction
  - May avoid penalty for jumps
- Speculative execution
  - Keeps the pipeline full

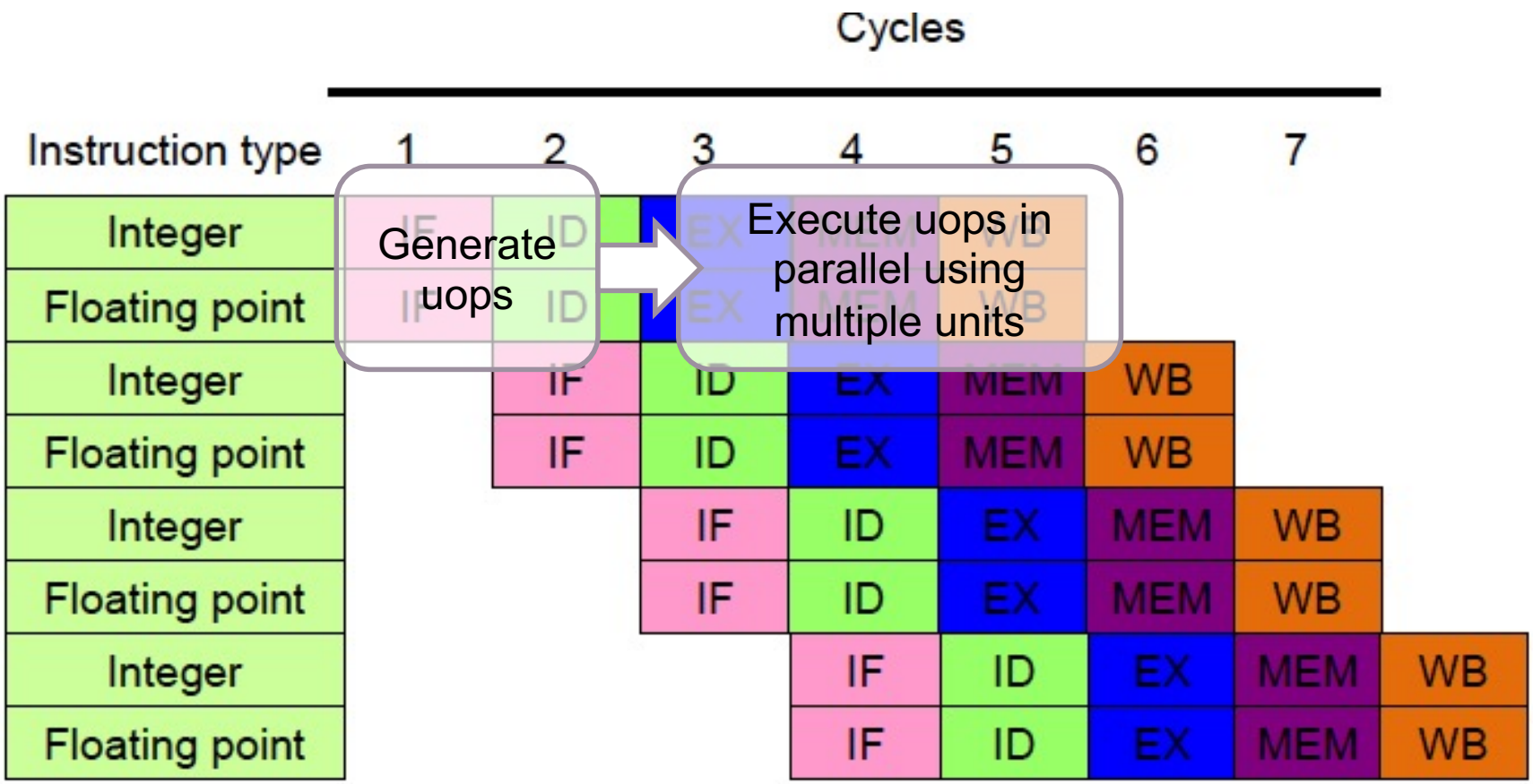
# BTW … pipelining?



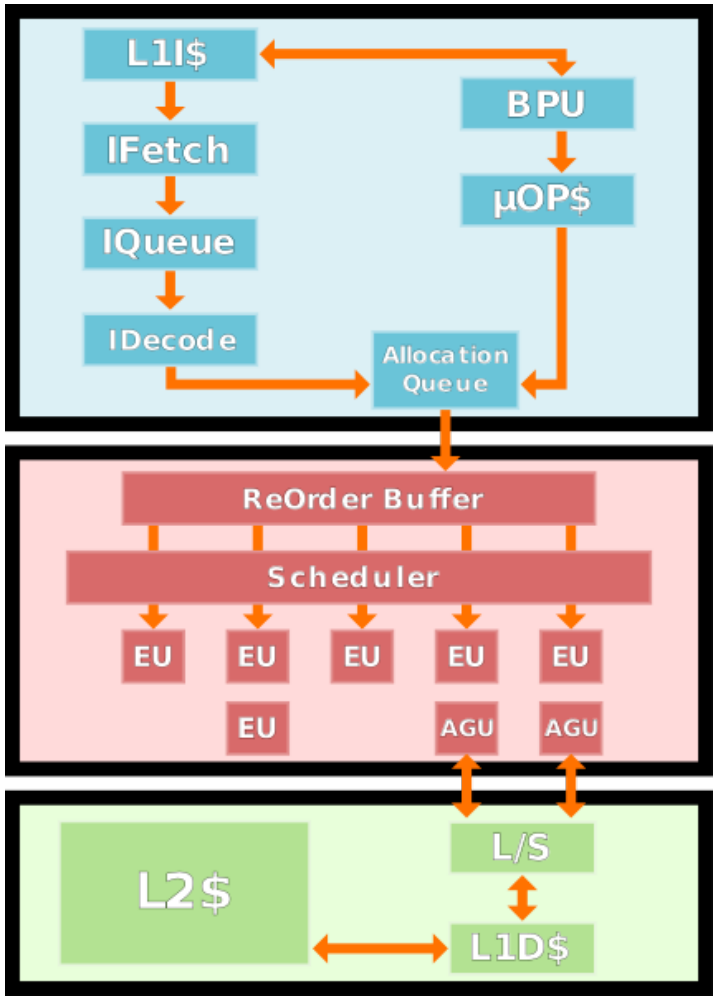
# BTW … superscalar?



# BTW … µops ?


Cycles
Instruction type
1
2
3
4
5
6
7
Integer
Floating point
Integer
Floating point
Integer
Floating point
Integer
Floating point
IF
ID
EX
MEM
WB
Generate uops
Execute uops in parallel using multiple units

# A super-scalar CPU




Front-end:
- Reads instructions
- Decodes to uOPs
- Pushes uOPs to the “Allocation queue”

Back-end:
- Optimizes uOPs and reorders them
- Schedules uOPs
- Executes uOPs on functional units

Memory system
- Load/store buffers
- L1 data cache
- Unified L2 cache

# In a nutshell …

- Front-end feeds the the back-end with sufficient operations
  - By decoding instructions coming from memory.
- All instructions arrive in the decode queue.
  - Interfaces with the backend
- Back-end:
  - IDQ to reorder buffer: micro-operations visit the reorder buffer
    - register allocation, renaming, and retiring
  - µOPs are sent to the unified scheduler
  - Scheduler assigns work to the execution units
    - queuing the µOPs on the appropriate ports
  - Once a uOP has been executed, it is retired
- Memory system
  - dedicated 32 KiB L1 data cache and 32 KiB L1 instruction cache.
  - Core-private 256 KiB L2 cache that is shared by both of the L1 caches.
  - Slice of the shared L3 cache (LLC)

Adapted from Wikichip:
https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)#Overview_2

# A more realistic picture … [1]

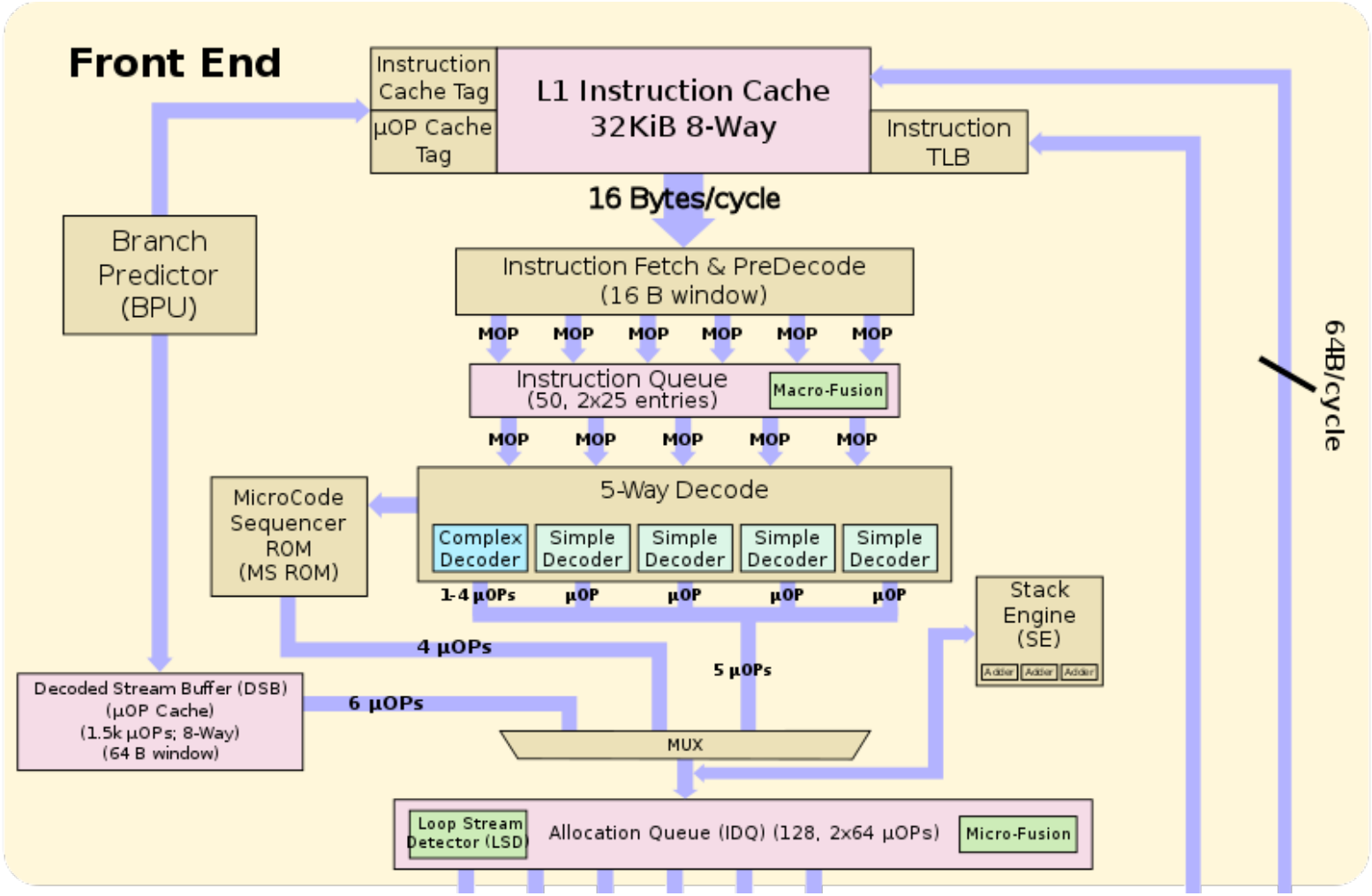



Image courtesy of Wikichip

# A more realistic picture … [2]

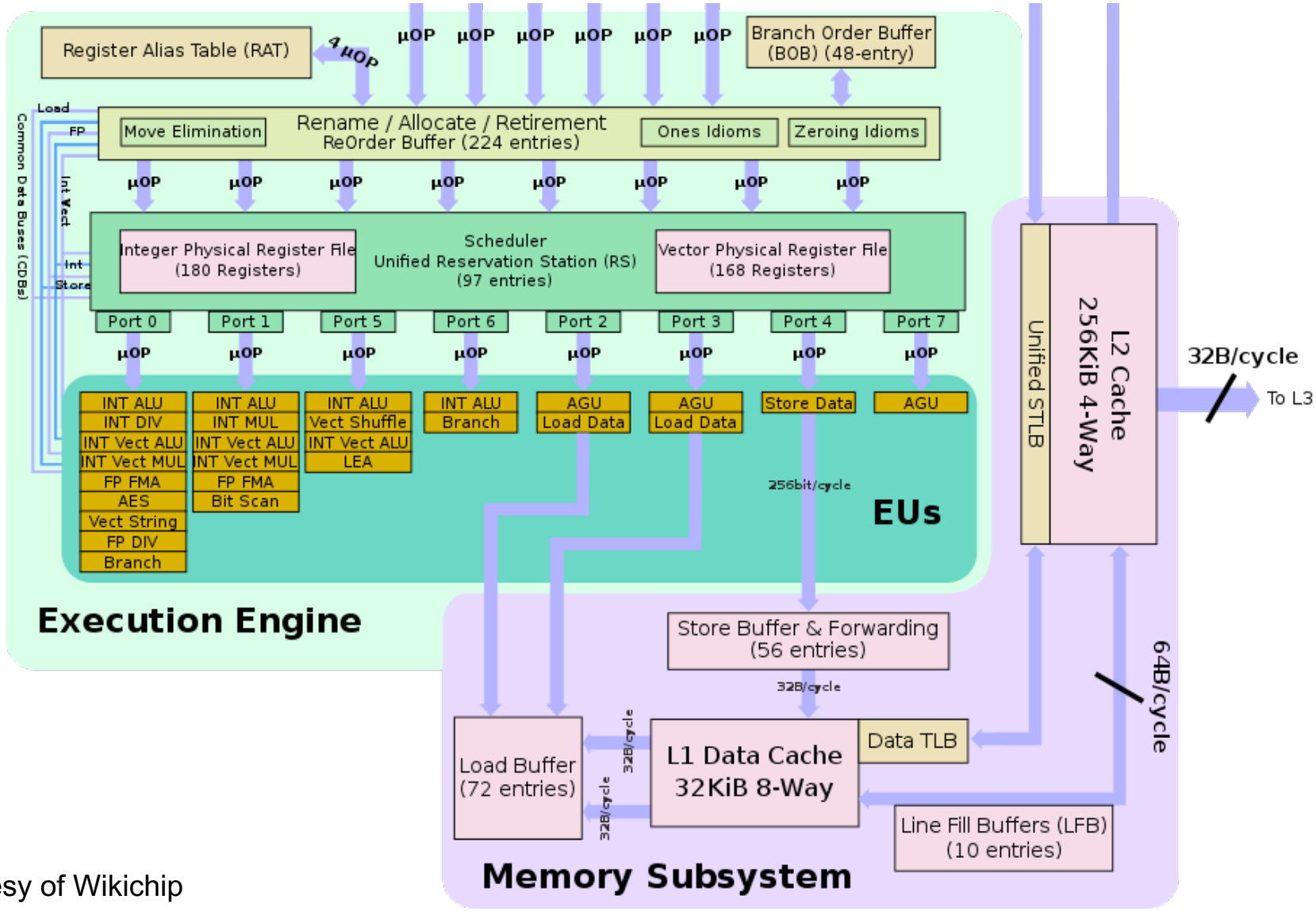



Image courtesy of Wikichip

# Execution units & ports

**Execution Units**

| Execution Unit | # of Units |
|---|---|
| ALU | 4 |
| DIV | 1 |
| Shift | 2 |
| Shuffle | 1 |
| Slow Int | 1 |
| Bit Manipulation | 2 |
| FP Mov | 1 |
| SIMD Misc | 1 |
| Vec ALU | 3 |
| Vec Shift | 2 |
| Vec Add | 2 |
| Vec Mul | 2 |

**Scheduler Ports Designation**

| | |
|---|---|
| **Port 0** | Integer/Vector Arithmetic, Multiplication, Logic, Shift, and String ops |
| | FP Add, Multiply, FMA |
| | Integer/FP Division and Square Root |
| | AES Encryption |
| | Branch2 |
| **Port 1** | Integer/Vector Arithmetic, Multiplication, Logic, Shift, and Bit Scanning |
| | FP Add, Multiply, FMA |
| **Port 5** | Integer/Vector Arithmetic, Logic |
| | Vector Permute |
| | x87 FP Add, Composite Int, CLMUL |
| **Port 6** | Integer Arithmetic, Logic, Shift |
| | Branch |
| **Port 2** | Load, AGU |
| **Port 3** | Load, AGU |
| **Port 4** | Store, AGU |
| **Port 7** | AGU |

# Execution unit performance

- Latency: how many cycles an operation takes
- Throughput: maximum number of independent instructions of the same type executed per cycle
- Reciprocal throughput: 1/throughput
- Issue latency: the minimum interval (in cycles) between 2 consecutive independent instructions of the same kind
  - Equivalent to reciprocal throughput (see Agner Fog's lists)

| Operation | Integer | | Single precision | | Double precision | |
|---|---|---|---|---|---|---|
| | Latency | Issue | Latency | Issue | Latency | Issue |
| Addition | 1 | 0.3 | 3 | 1.0 | 3 | 1.0 |
| Multiplication | 3 | 1.0 | 4 | 1.0 | 5 | 1.0 |
| Division | 11-21 | 5-13 | 10-15 | 6-11 | 10-23 | 6-19 |

Fully pipelined unit.

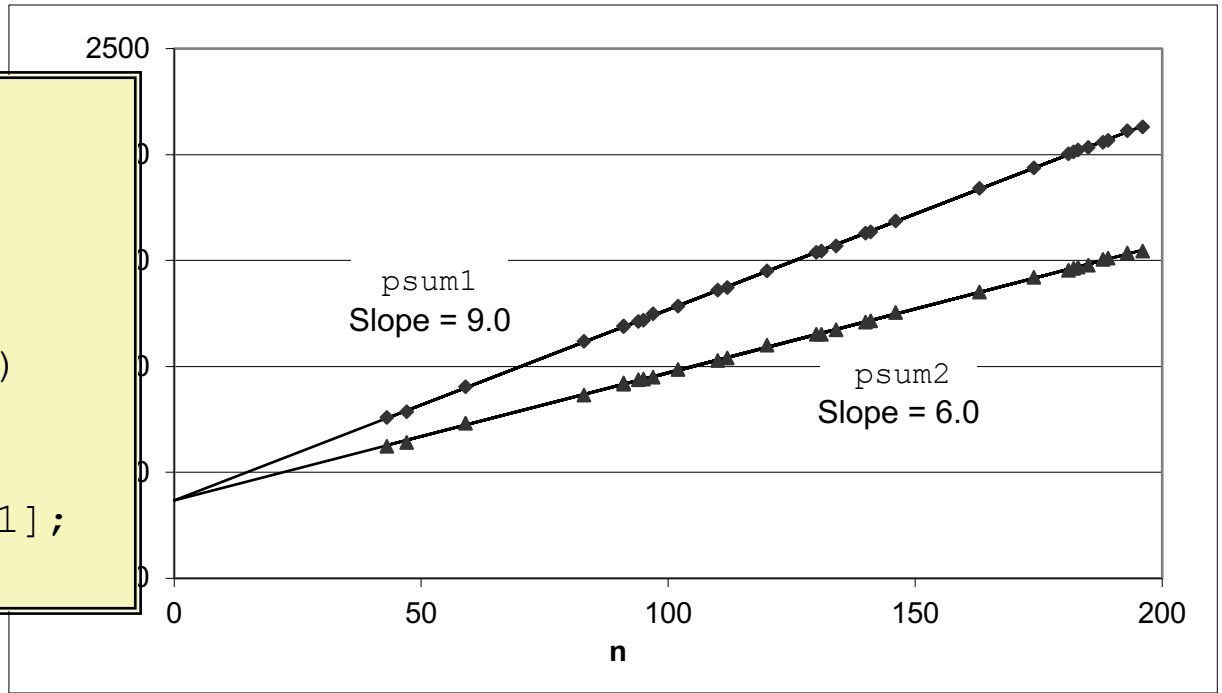
# Loop performance

- Execution time
  - Measured and/or approximated in cycles
- CPE = cycles per element/iteration
  - Performance of program that operates on vectors or lists == in general, loops!
- Execution time: T = CPE*n + Overhead
  - CPE is slope of line

How do we determine CPE?

```
void psum1(float *a,
```

```
void psum2(float *a,
           float *p,
           int n) {
  int i;
  p[0]=a[0];
  for (i=1; i<n; i+=2)
    tmp=p[i-1]+a[i];
    p[i]=tmp;
    p[i+1]=tmp + a[i+1];
}
```



# The CPE metric

Performance metric: CPE (cycles per element/iteration)

- Useful for loops and iterative computation

For a given CPU:

- **the latency bound** => a minimum bound for CPE when execution must be in order (think dependency!).
- **the throughput bound** => a minimum bound for CPE based on the maximum rate of the functional units.

| Bound | Integer | | Single precision | | Double precision | |
|---|---|---|---|---|---|---|
| | + | * | + | * | + | * |
| Latency | 1 | 3 | 3 | 4 | 3 | 5 |
| Throughput | 1 | 1 | 1 | 1 | 1 | 1 |

CPE = cycles per element

# CPE vs. the bounds

- Analyzing the application using CPE: ***lower is better***
  - CPE > max{latency bound, throughput bound}
  - CPE = latency bound > throughput bound
    - Investigate potential dependencies between operations of the same kind
      - Not enough instruction mix diversity
      - Usually loop unrolling helps
  - CPE = throughput bound
    - This is the best you can do ☺

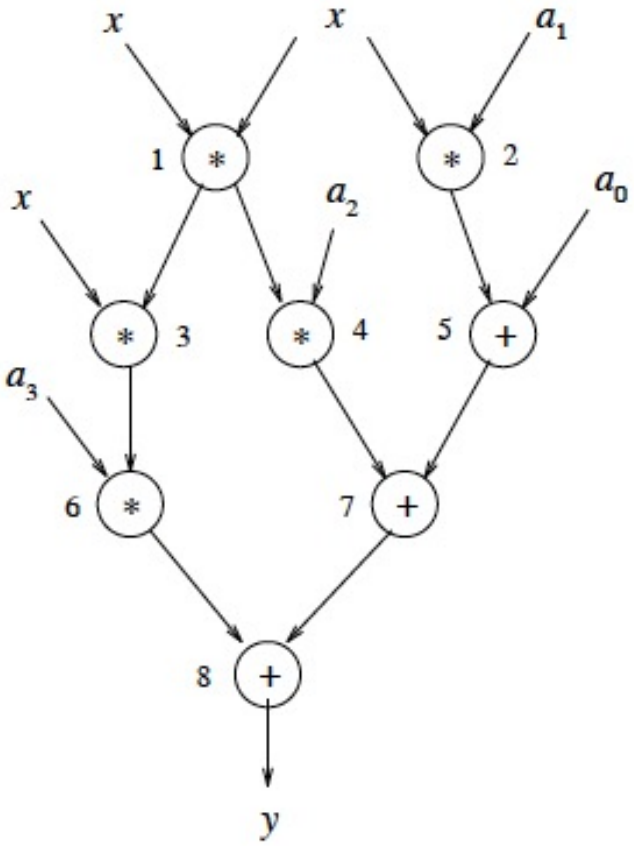
# So … the impact of ILP?

- Can easily make errors when counting operations.

- Instructions are sequences of microoperations (uops)
  - Generated by the compiler
  - Reordered by the compiler AND the hardware
- Mapping:
  - Which instruction is mapped on which unit?
    - i.e., where does it execute?
- Scheduling
  - How are the instructions scheduled for execution?
    - i.e., when do they execute?

- Why do we care?!

# An example




- Example:
  Compute: $y = a_0 + a_1 * x + a_2 * x * x + a_3 * x * x * x$
  Compute: $y = a_0 + a_1 * x + a_2 * x * x + a_3 * (x * x^2)$
- Analytical performance model:
  Work: T = 5 x t_mul + 3 x t_add
  Span: 3 x t_mul + 1 x t_add
- Task graph
  - Parallelism: 2 mul, 2 mul + 1 add, 1mul + 1 add, 1 add
- Scheduling for 4 execution units: 2 mul and 2 add
  - Execution time: T = T_span
- Scheduling for 2 execution units: 1 mul and 1 add
  - Parallelism reduced by scheduling: T_span + 2 mul
    - For example: (2), (1,3), (4), (3,7), (6), (8)

# Our simple models revisited

- What is the impact of DAGs, scheduling, and contention for our previous models?
  - Assumption?
  - Reality?
- Refine the model!
  - Scheduling of operations on cores / processing units …
    - ILP (instruction level parallelism) constraints
    - Memory and operations overlap
- … but VERY architecture dependent!
  - Understand the architecture in detail
  - Use a simulator
  - Use performance counters & microbenchmarking

https://software.intel.com/en-us/node/544393

# Model validation

# Validating the model: empirically!

- Step 1: What is the goal of the model?
  - Ranking, prediction, behavior trend, performance bounds, …
- Step 2: What are the relevant metrics?
- Step 3: Design a (set of) experiment(s) to measure the behavior you are modeling
  - Careful with the dataset!
- Step 4: Reason about accuracy
  - How good are your predictions?
  - How about trends?
  - Stable vs. unstable regimes?
- Step 5: Reason about in-accuracies
  - What? Why? How to determine?

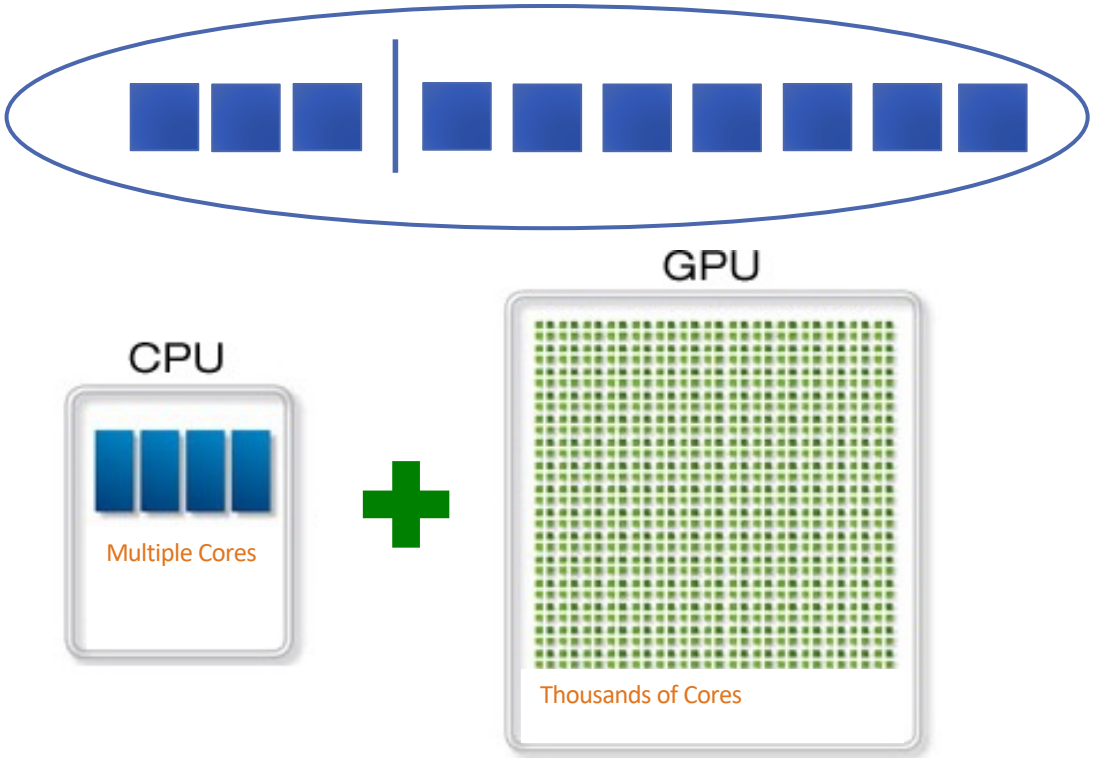
# Examples of analytical models

# Examples

- Heterogeneous computing model (from TUDelft)
- GPU application model (from UvA)
- ECM = a generic analytical model (from University of Erlangen)

# Heterogeneous computing

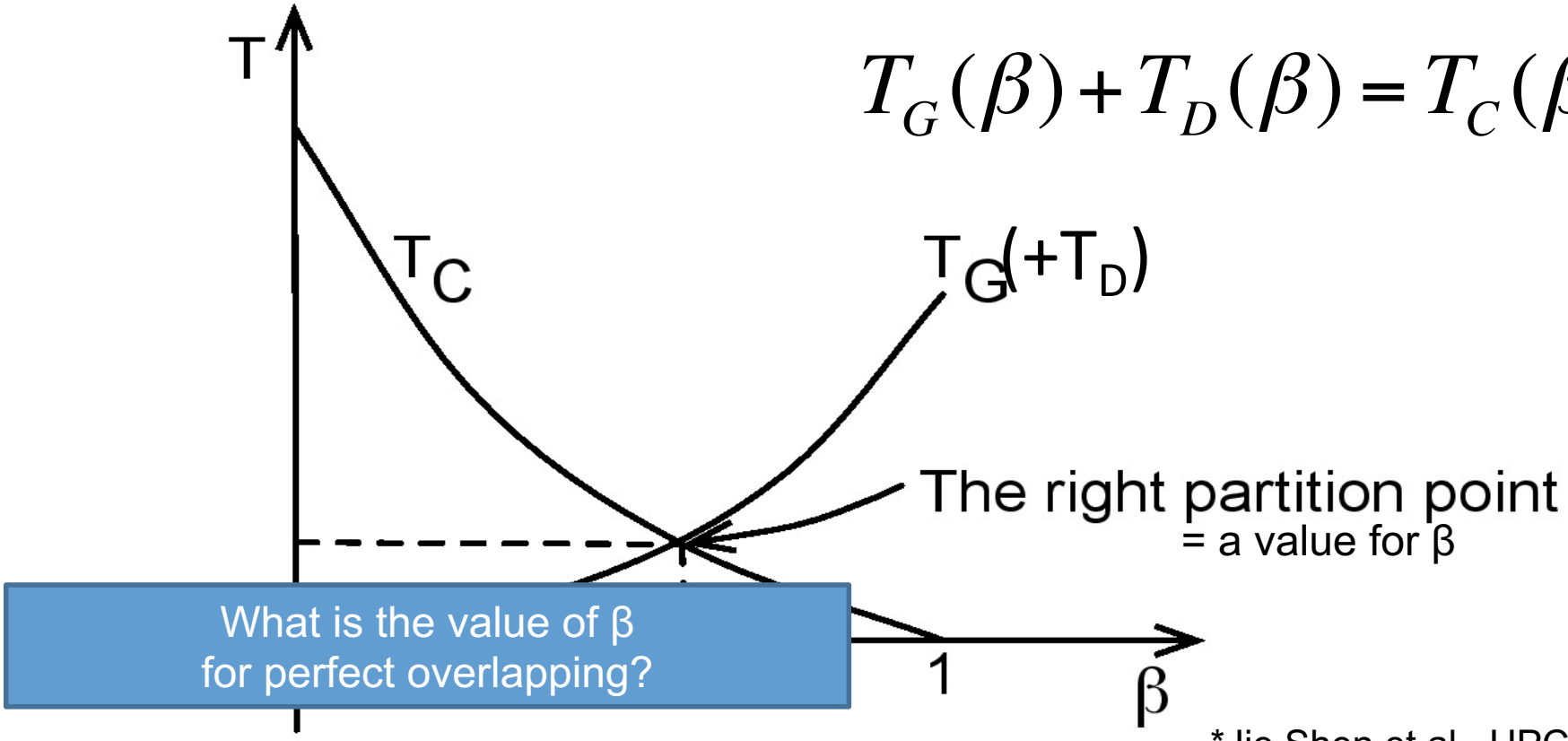
- Assume a system using a CPU and a GPU
- What is the perfect partitioning for the CPU-GPU system?



# Optimal partitioning

We assume the following:

- An application
  - Application workload is a function of the input size, n, and the workload per data element.
- A platform with a CPU and a GPU

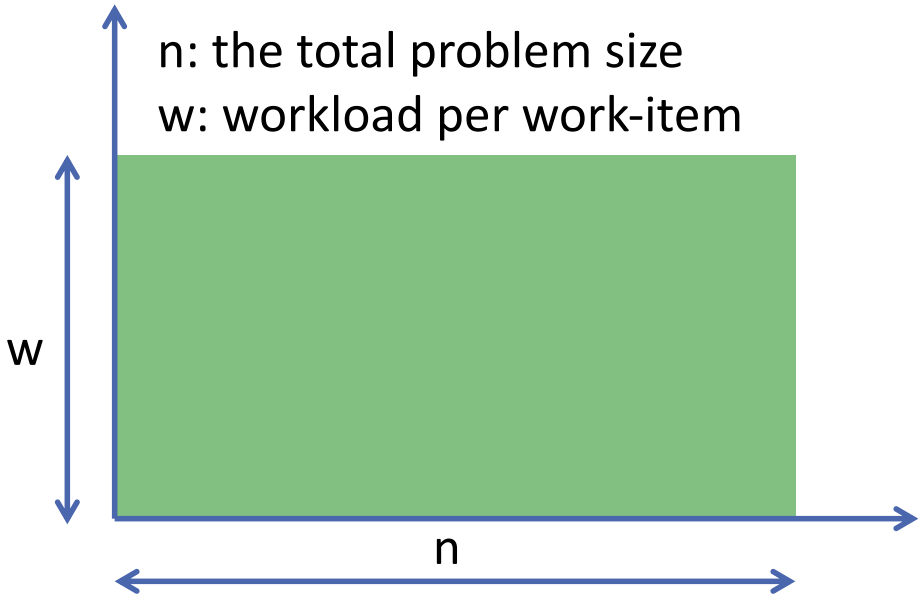


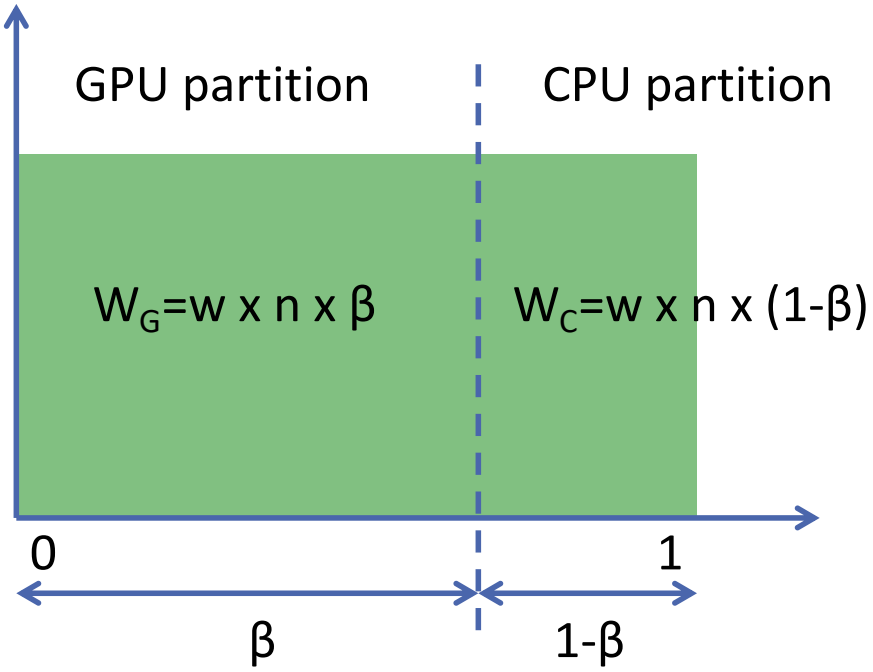

$$T_G(\beta) + T_D(\beta) = T_C(\beta)$$

*Jie Shen et al., HPCC'14.

"Look before you Leap: Using the Right Hardware Resources to Accelerate Applications

# Application workload




W (total workload size) quantifies how much work has to be done in a partition

# Modeling the partitioning

$$T_G = \frac{W_G}{P_G} \quad T_C = \frac{W_C}{P_C} \quad T_D = \frac{O}{Q}$$

* $W = W_G + W_C$

Two pairs of metrics

*W*: total workload size
*P*: processing throughput (*W*/second)

*O*: data-transfer size
*Q*: data-transfer bandwidth (bytes/second)

Measured (by profiling or (micro)benchmarking)

Compute β

$$T_G + T_D = T_C \quad \rightsquigarrow \quad \frac{W_G}{W_C} = \frac{\beta}{1-\beta} = \frac{P_G}{P_C} \times \frac{1}{1 + \frac{P_G}{Q} \times \frac{O}{W_G}}$$

# How about GPUs?

# GPU application modelling: PageRank

- Calculates the PR value for all vertices
  - Assign value to each vertex
  - Repeat until convergence
    - Collect PR for all incoming edges
    - Update vertex PR
- Sensitive to …
  - Graph density
  - Degree distribution
  - ”sink” nodes
- Challenges
  - No computation
  - Load-balancing
  - Irregular memory accesses

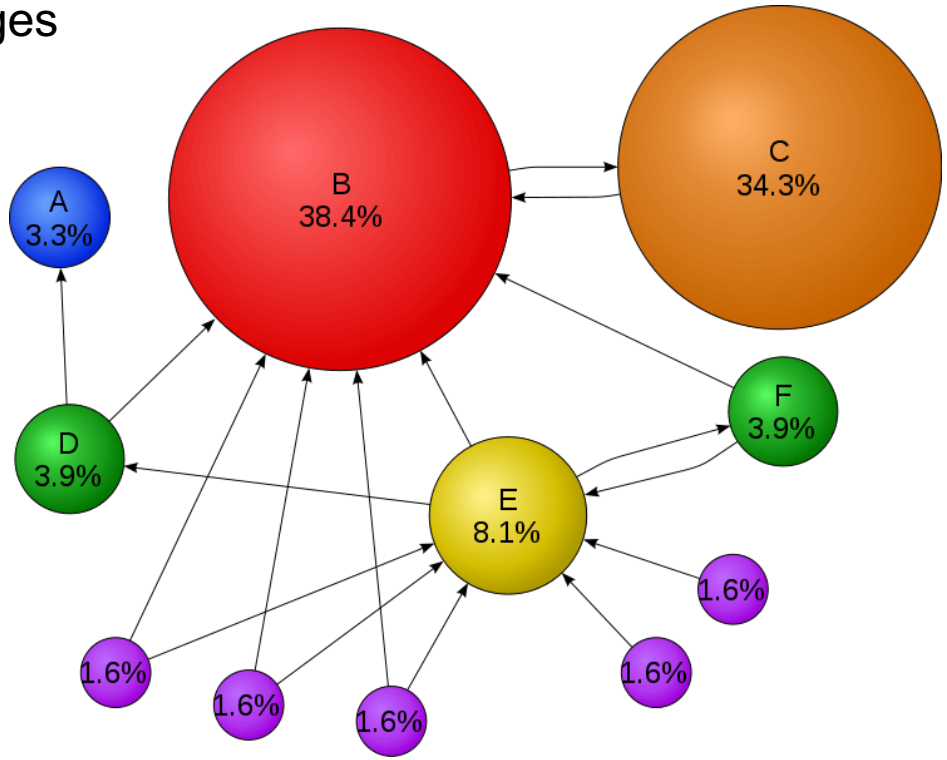



Image courtesy of: https://en.wikipedia.org/wiki/PageRank

# GPU application modelling: PageRank

- Calculates the PR value for all vertices
  - Assign value to each vertex
  - Repeat until convergence
    - Collect PR for all incoming edges
    - Update vertex PR

We use 7 versions + 2 warp-parallelism parameterized ones

- Challenges
  - No computation
  - Load-balancing
  - Irregular memory accesses

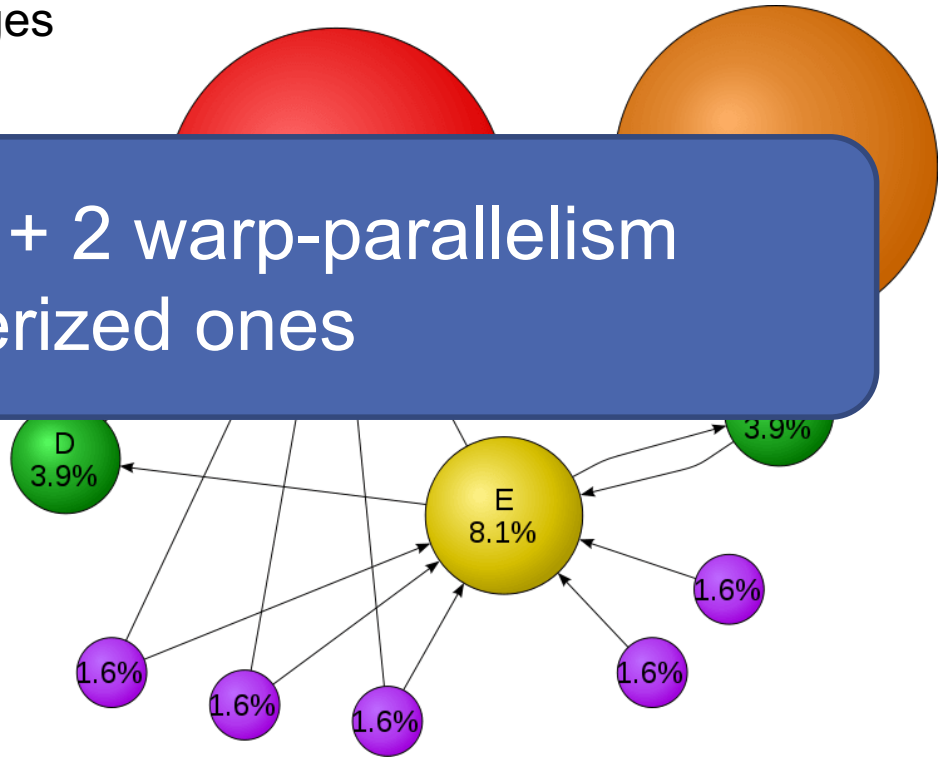


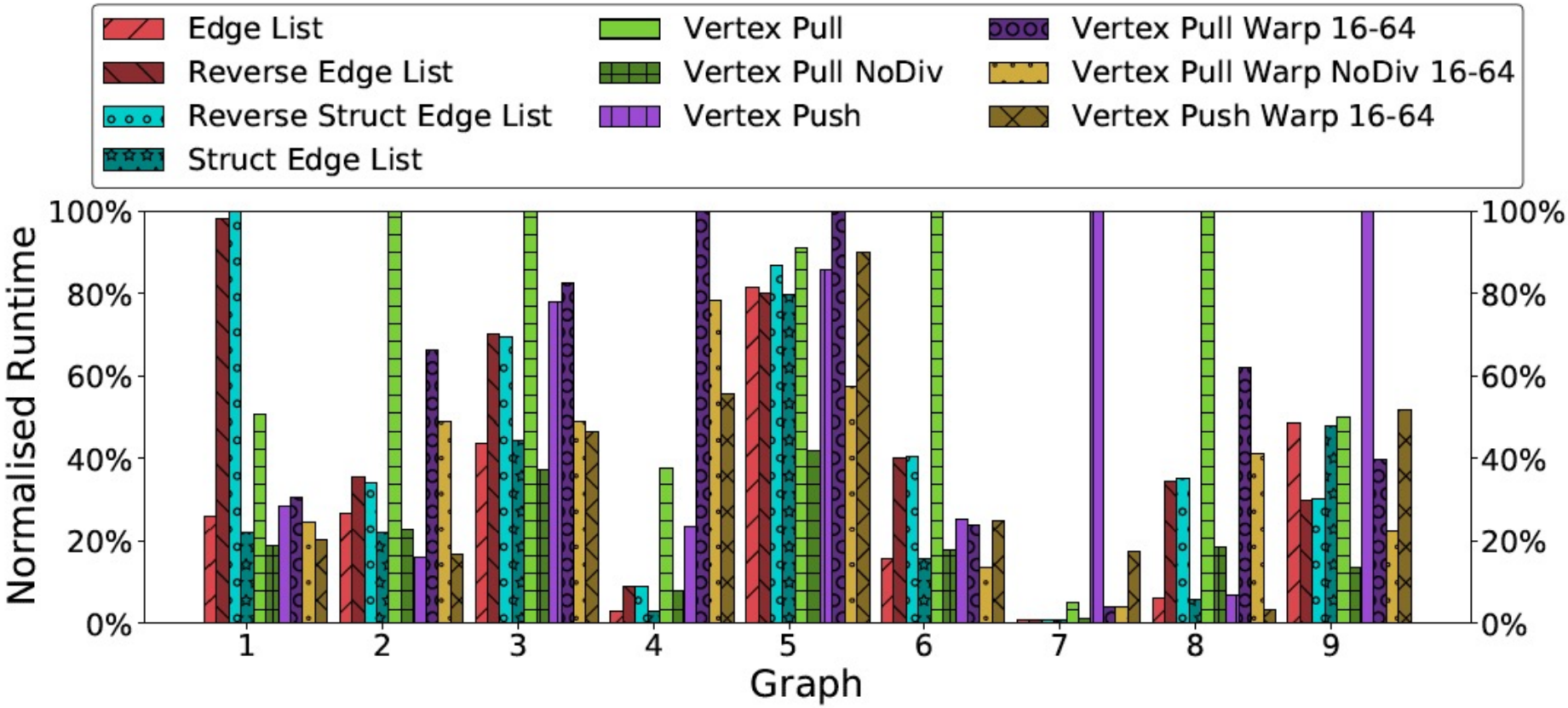

Image courtesy of: https://en.wikipedia.org/wiki/PageRank

# All experiments …

- NVIDIA TitanX + CUDA 10.0
- Results presented on 9 graphs

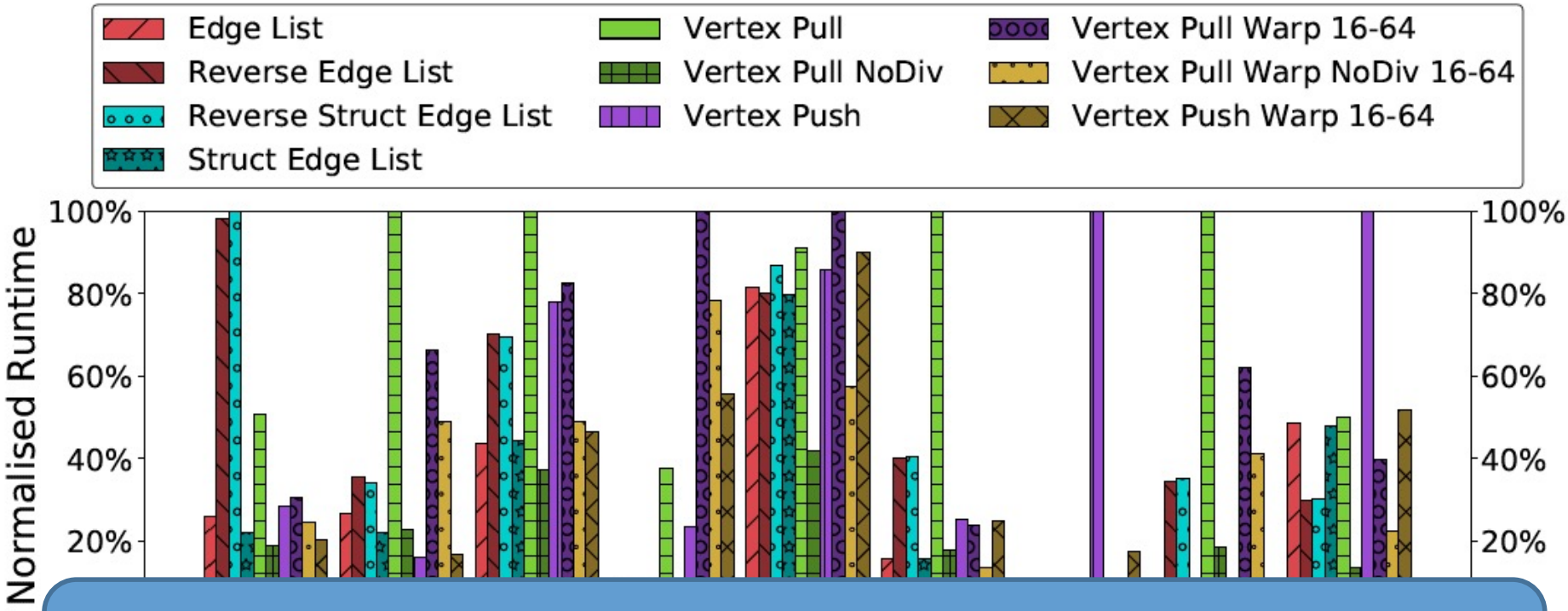
| Id | Graph | # Vertices | # Edges | Dataset |
|---|---|---|---|---|
| 1 | actor-collaboration | 382,219 | 30,076,200 | KONECT |
| 2 | amazon0601 | 403,394 | 3,387,390 | KONECT |
| 3 | flixster | 2,523,390 | 15,837,600 | KONECT |
| 4 | jester1 | 73,512 | 8,272,720 | KONECT |
| 5 | patentcite | 3,774,770 | 16,518,900 | KONECT |
| 6 | wikipedia_link_en | 12,151,000 | 378,142,000 | KONECT |
| 7 | wiki_talk_ru | 457,017 | 919,790 | KONECT |
| 8 | higgs-social_network | 456,626 | 14,855,800 | SNAP |
| 9 | sx-stackoverflow-c2q | 1,655,350 | 11,226,800 | SNAP |

# PageRank: results




https://github.com/merijn/Belewitte

# PageRank: results




- Different algorithms behave best.
- Different algorithms behave worst.
- The gap in execution time can be up to 2 orders of magnitude.

Choosing the right / wrong algorithm can really make a difference!

# Choose the best algorithm

- Model the algorithm
  - Basic analytical model (work & span)
- Calibrate to platform
  - GPU, CPU, …
- Model the dataset
  - Size, dimension, topology …

**T = f(P, A, D)**

- Predict performance
  - Plug the platform and graph parameters into algorithm model

- Rank solutions and pick best.

# Example implementations

**Algorithm 1** Edge List Update & Reverse Edge List Update

```
function EdgeList(edges, pageranks, new_pageranks, idx)
    origin ← edges[idx].origin
    dest ← edges[idx].destination
    outgoingRank ← 0
    if degree(origin) ≠ 0 then
        outgoingRank ← pageranks[origin] / degree(origin)
    new_pageranks[dest].atomicAdd(outgoingRank)
```

# Example implementations

**Algorithm 1** Edge List Update & Reverse Edge List Update

**Algorithm 2** Vertex Push Update

**function** VERTEXPUSH(*vertices*, *pageranks*, *new_pageranks*, *idx*)

$outgoingRank \leftarrow 0$

**if** $degree(idx) \neq 0$ **then**

$outgoingRank \leftarrow \frac{pageranks[idx]}{degree(idx)}$

**for** $nbr \in vertices[idx].neighbours$ **do**

$new\_pageranks[nbr].atomicAdd(outgoingRank)$

# Example implementations

**Algorithm 1** Edge List Update & Reverse Edge List Update

**Algorithm 2** Vertex Push Update

**Algorithm 3** Vertex Pull Update

**function** VERTEXPULL(*vertices*, *pageranks*, *new_pageranks*, *idx*)

$newRank \leftarrow 0$

**for** $nbr \in vertices[idx].neighbours$ **do**

$newRank \mathrel{+}= \frac{pageranks[nbr]}{degree(nbr)}$

$new\_pageranks[idx] \leftarrow newRank$

# PageRank: Analytical models

- Different algorithms => different models

$$T_{\text{edge}} = \sum_{e \in E} (4 * T_{\text{read}} + T_{\text{atom}})$$
$$= 4 * |E| * T_{\text{read}} + |E| * T_{\text{atom}}$$

# PageRank: Analytical models

- Different algorithms => different models

$$T_{\text{edge}} =$$

$$= 4$$

$$T_{\text{push}} = \sum_{v \in V} (2 * T_{\text{read}} + d_v * T_{\text{atom}})$$

$$= 2 * |V| * T_{\text{read}} + |E| * T_{\text{atom}}$$

# PageRank: Analytical models

- Different algorithms => different models

$$T_{\text{edge}} = \sum$$

$$T_{\text{push}} = \sum (2 * T_{\text{read}} + d_v * T_{\text{atom}})$$

$$T_{\text{pull}} = \sum_{v \in V} (2 * d_v * T_{\text{read}} + T_{\text{write}})$$

$$= 2 * |E| * T_{\text{read}} + |V| * T_{\text{write}}$$

- Extracted from the algorithms' pseudocode
  - Not accurate enough, as there are more operations executed in practice …

# PageRank: Analytical models

- Different algorithms => different models

$$
\begin{aligned}
T_{\text{edge}} &= (7 * |E| * T_{\text{read}} + |E| * T_{\text{atom}}) \\
&\quad + (2 * |V| * T_{\text{read}} + 2 * |V| * T_{\text{write}} + \frac{|V|}{32} * T_{\text{atom}}) \\
&= (7 * |E| + 2 * |V|) * T_{\text{read}} + 2 * |V| * T_{\text{write}} + (|E| + \frac{|V|}{32}) * T_{\text{atom}}
\end{aligned}
$$

- Based on PTX (that is, assembly)

# PageRank: Analytical models

- Different algorithms => different models

$$T_{\text{edge}} = (7 * |E| * T_{\text{read}} + |E| * T_{\text{atom}})$$

$$T_{\text{push}} = (6 * |V| * T_{\text{read}} + |E| * T_{\text{read}} + |E| * T_{\text{atom}}) + (2 * |V| * T_{\text{read}} + 2 * |V| * T_{\text{write}} + \frac{|V|}{32} * T_{\text{atom}})$$

$$= (8 * |V| + |E|) * T_{\text{read}} + 2 * |V| * T_{\text{write}} + (|E| + \frac{|V|}{32}) * T_{\text{atom}}$$

- Based on PTX (that is, assembly)

# PageRank: Analytical models

- Different algorithms => different models

$$T_{\text{edge}} = (7 * |E| * T_{\text{read}} + |E| * T_{\text{atom}})$$

$$T_{\text{push}} = (6 * |V| * T_{\text{read}} + |E| * T_{\text{read}} + |E| * T_{\text{atom}})$$

$$\begin{aligned} T_{\text{pull}} &= (5 * |V| * T_{\text{read}} + 3 * |E| * T_{\text{read}} + |V| * T_{\text{write}}) \\ &\quad + (2 * |V| * T_{\text{read}} + 2 * |V| * T_{\text{write}} + \frac{|V|}{32} * T_{\text{atom}}) \\ &= (7 * |V| + 3 * |E|) * T_{\text{read}} + 3 * |V| * T_{\text{write}} + \frac{|V|}{32} * T_{\text{atom}} \end{aligned}$$

- Based on PTX (that is, assembly)
- Calibrate for the platform : $T_{\text{read}}$, $T_{\text{write}}$, $T_{\text{atom}}$ …
- Use dataset features: |E| and |V| from the graph specs

# Validate models

- Work-models are correct
  - We capture correctly the number of operations
- Model calibration has failed
  - Workload imbalance between threads within a warp
  - Non-uniform memory access times due to coalescing, caching, and atomic contention.
- Can we do any better?
  - Model parallelism to better understand the variation in T's
  - Use performance counters to capture different aspects of T's

# How about generic analytical models?

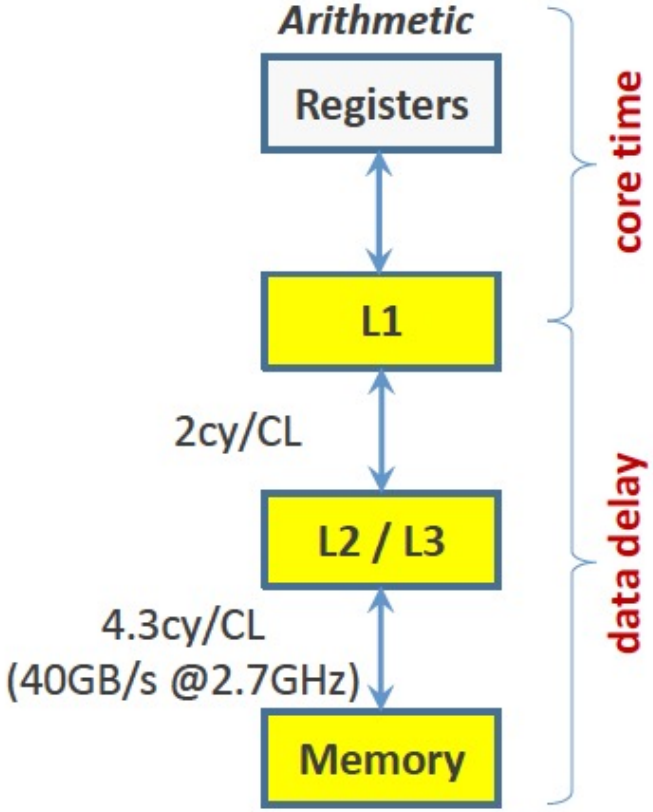
# The ECM model*

- Execution-Cache-Memory
  - Models the interaction between the execution and memory subsystems
- Estimates: cycles per iteration/element
- Uses:
  - Cycles as principal metric
  - Cycles/cache line for the data transfers

https://hpc.fau.de/research/ecm/

# The ECM model*

- Basic principles:
  - Single core performance is determined by:
    - Core execution time (assume data in L1)
    - Data delay in the memory system
  - Per socket-scaling
    - Limited by the shared resource (LLC, usually)
- Input:
  - Benchmark all levels (using STREAM)
  - Determine cycles per layer
  - Identify bottleneck as compute or memory



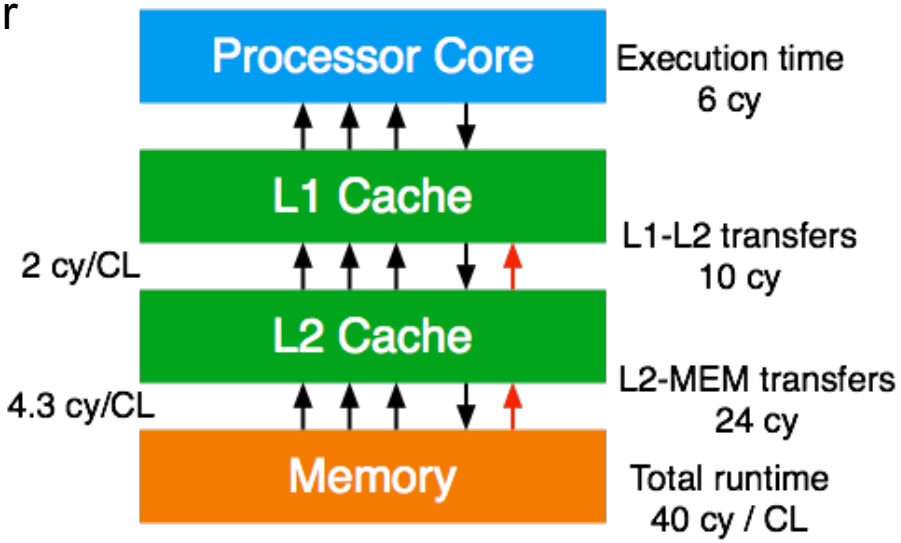

https://hpc.fau.de/research/ecm/

# Example: Triad (adapted from STREAM)

- Assume the following code:

```
for (int i=0; i<size; i++) A[i] = B[i] + C[i] * D[i];
```

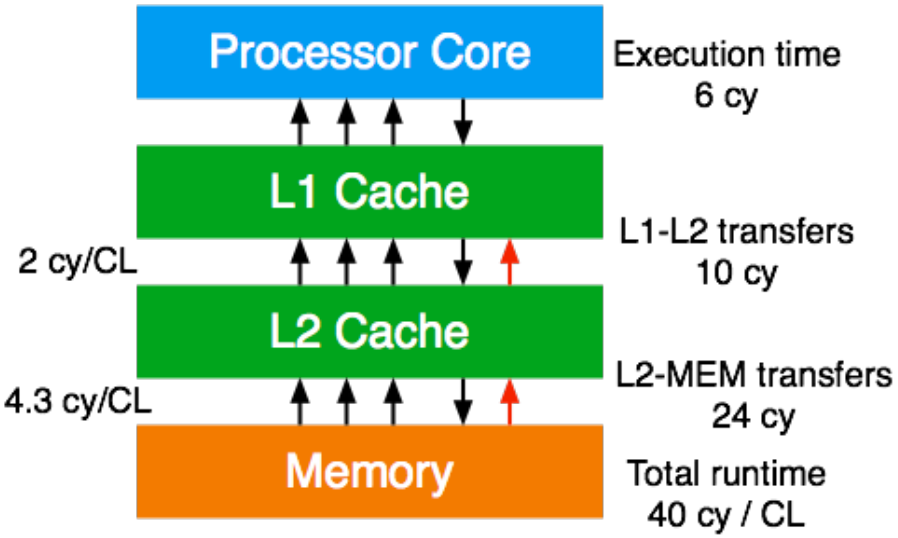
- Assume the following machine:
  - CL = 64b
  - 1xSIMD ADD and 1xSIMD MUL per cycle (32b)
  - 1x32b LD and 0.5x32b ST per cycle
  - Inclusive cache, write-back
    - Store miss => write-allocate transfer
  - Memory bandwidth: 40GB/s
  - No overlap in memory transfers!
- Thus:
  - 2xMUL, 2xADD: 2cy
  - 6xLD: 6 cy
  - 2xST: 4 cy



# Example: Triad (adapted from STREAM)

```
for (int i=0; i<size; i++) A[i] = B[i] + C[i] * D[i];
```

- Execution time: 6 cycles  (max of all ops)
- L2-L1: per 8 iterations, 3CL loads + 1CL store + 1 transfer
  - 5CL transfers => 10 cycles
- L2-Mem : same volume
  - 5CL transfers => 21.5 cycles



# Triad (adapted from STREAM)

```
for (int i=0; i<size; i++) A[i] = B[i] + C[i] * D[i];
```

Notation:

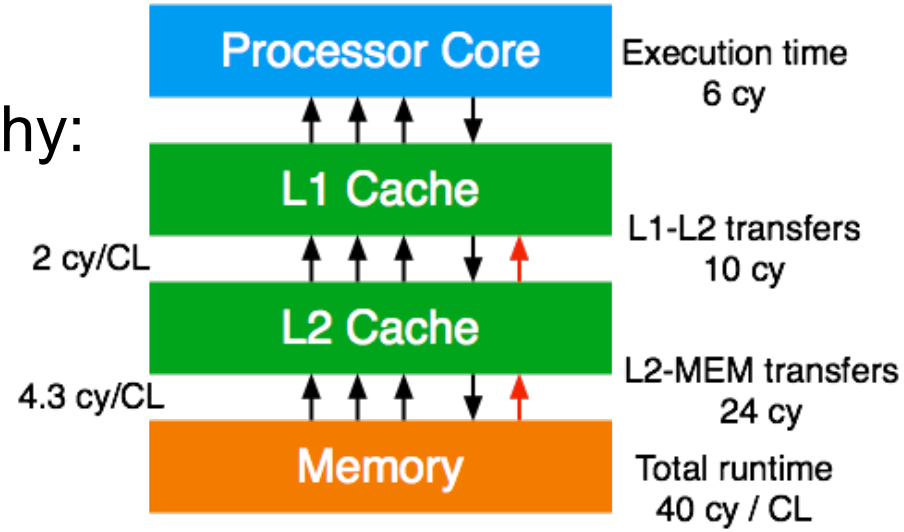
$$\{ T_{OL} \parallel T_{nOL} \mid T_{L1L2} \mid T_{L2L3} \mid T_{L3Mem} \}$$

{6 || 3 | 10 | 21.5}

Cycle predictions for the hierarchy:

$$\{ T_{ECM}^{L1} \rceil T_{ECM}^{L2} \rceil T_{ECM}^{L3} \rceil T_{ECM}^{Mem} \}$$

{6 | 16 | 37.5} cycles



# Multi-core?

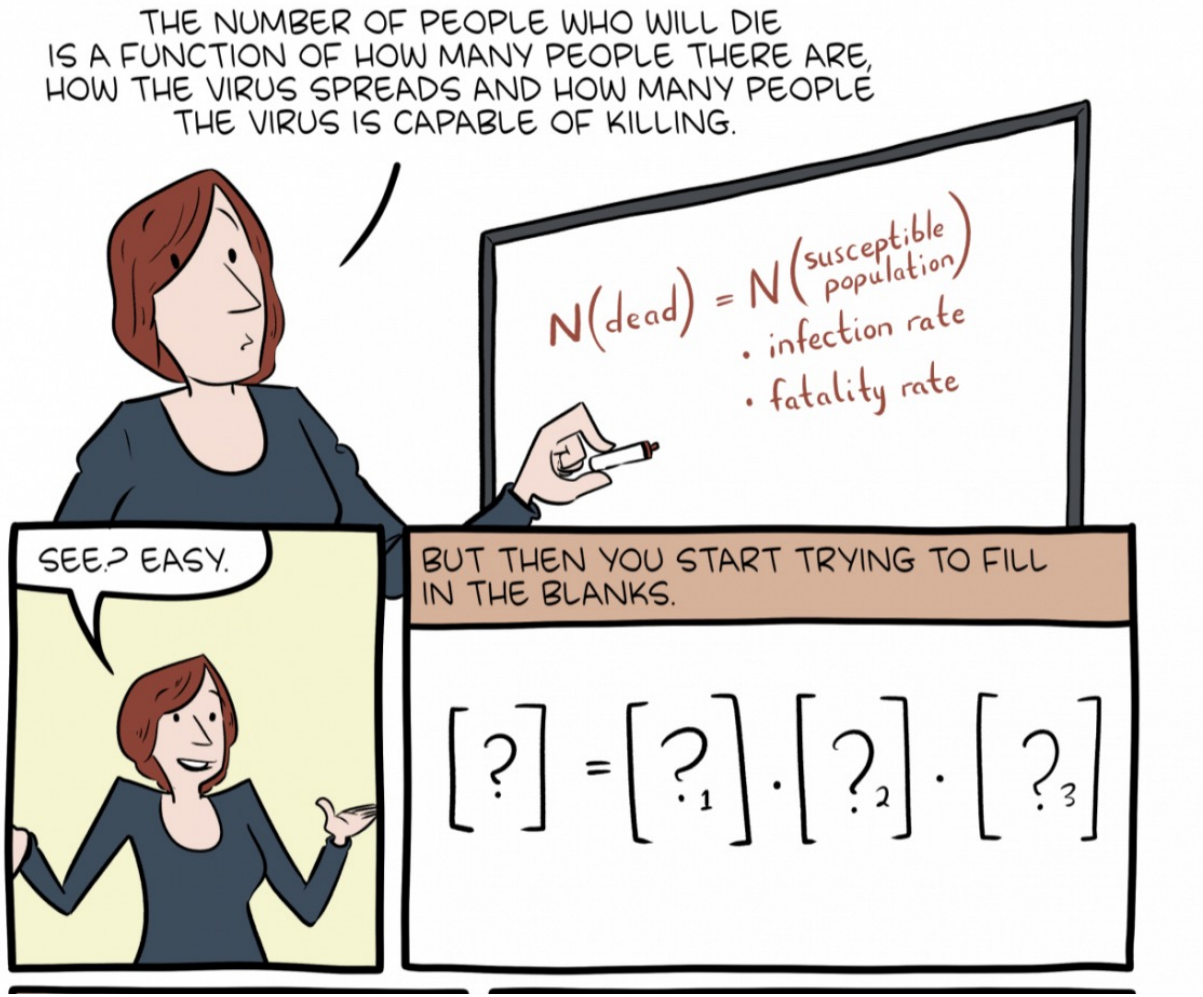
- Main assumption: performance scales linearly till bottleneck
  - n cores => n x CL in the same time : $T^{Mem}_{ECM}$
  - $BW_{n\ cores} = n \times CL / T^{Mem}_{ECM}$
- Most common bottleneck is LLC-M bandwidth
  - $T_{L3Mem} = X$ cy => $BW_{L2Mem} = CL / X$
- Saturation?
  - $BW_{n\ cores} = BW_{L3Mem}$ => $n \times CL / T^{Mem}_{ECM} = CL / X$ => $n = T^{Mem}_{ECM} / X$
- Our example: n = 37.5 / 21.5 ~ 2

# Kerncraft performance modeling

- Automated system for performance modeling
- Uses:
  - Machine model
    - Manually provided
  - Lots of (micro)benchmarking
    - Bandwidths between different memory modules
  - Microinstruction knowledge / scheduling
  - Actual code
- Provides:
  - Performance model in the Roofline form
  - Performance model in the ECM form
  - Multi-core saturation due to LLC

https://github.com/RRZE-HPC/kerncraft

# Done analytical modeling …




https://www.graphicmedicine.org/covid-19-comics/