HETEROGENEOUS CPU+GPU COMPUTING

Ana Lucia Varbanescu – University of Amsterdam
a.l.varbanescu@uva.nl

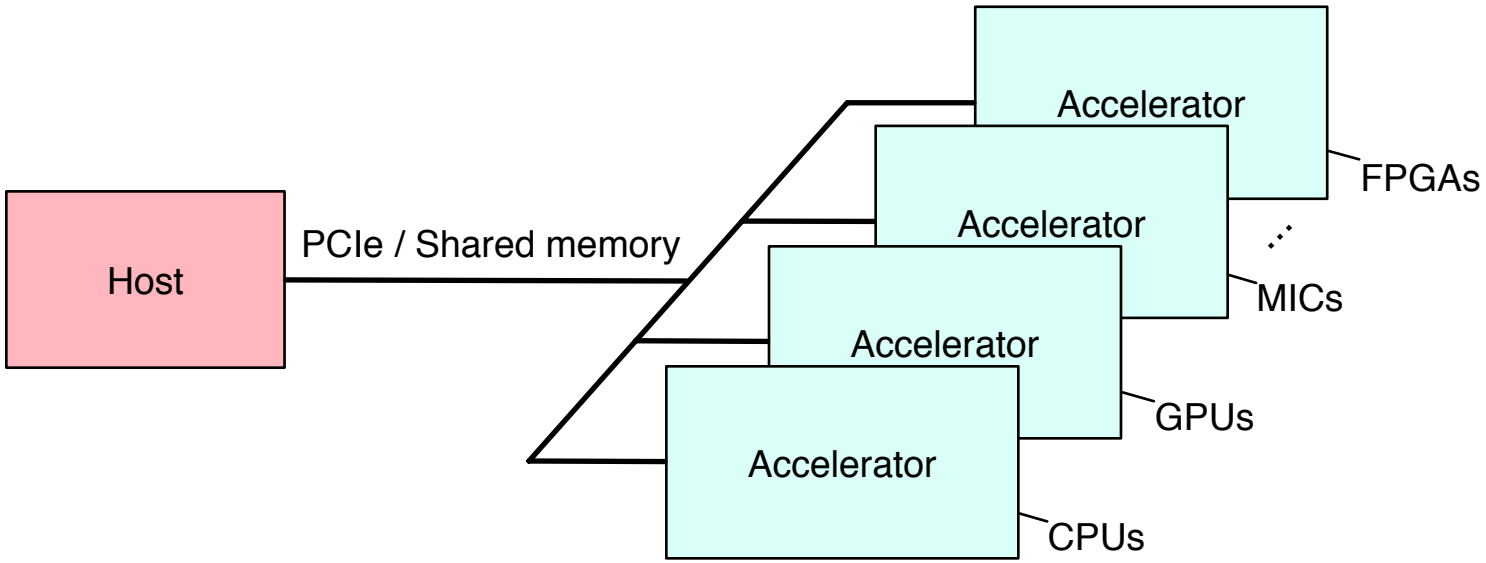
Significant contributions by: **Stijn Heldens** (U Twente),
Jie Shen (NUDT, China),

Heterogeneous platforms

- Systems combining main processors and accelerators
  - e.g., CPU + GPU, CPU + Intel MIC, AMD APU, ARM SoC
  - Everywhere from supercomputers to mobile devices

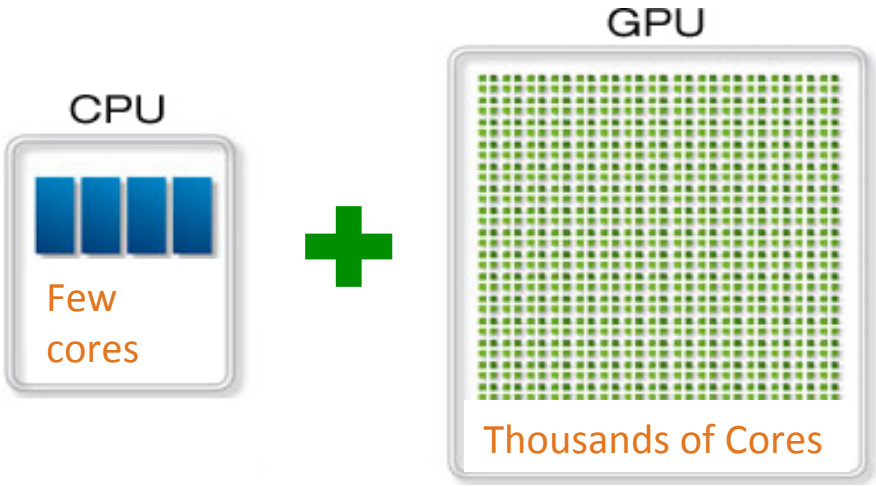
Heterogeneous platforms

- Host-accelerator hardware model

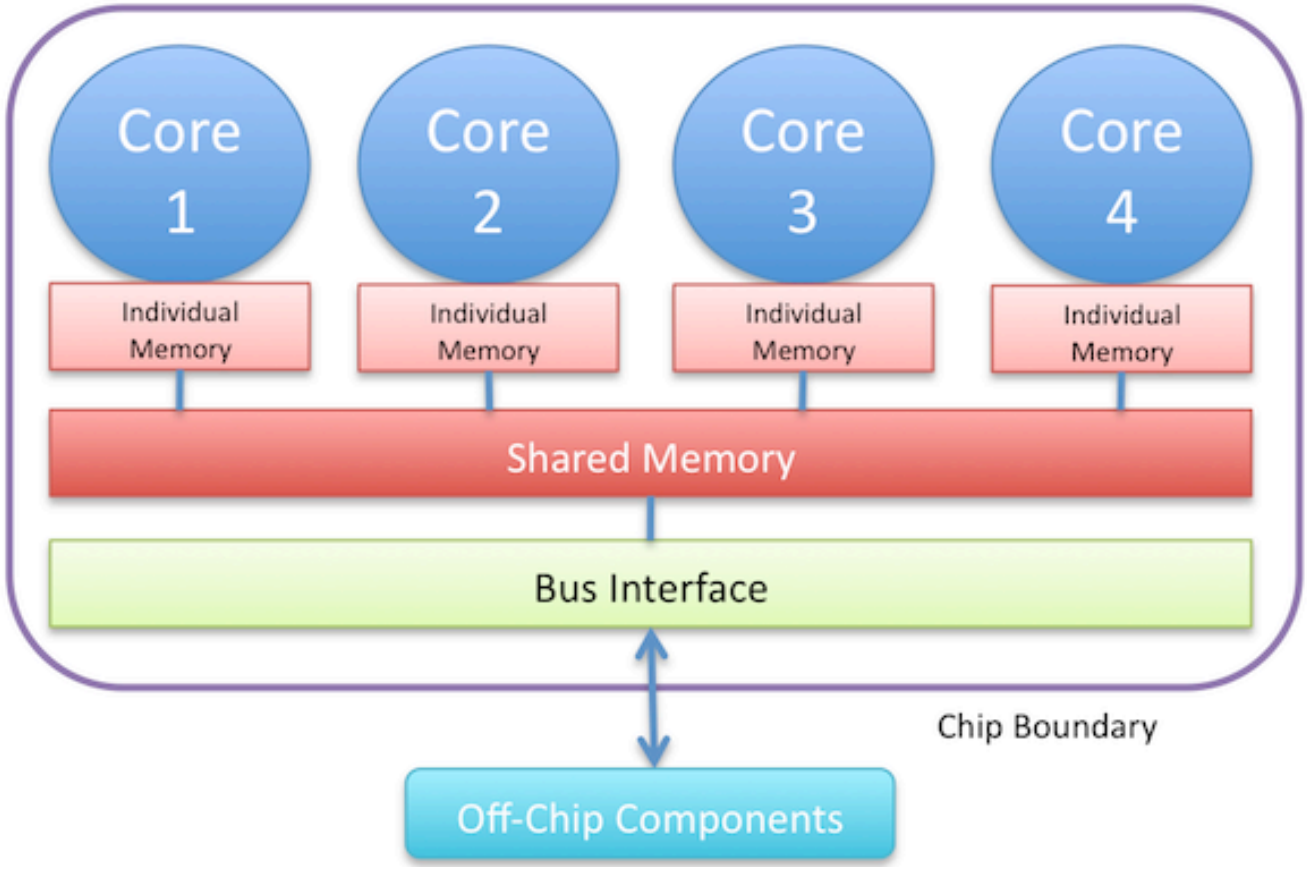


Our focus today …

- A heterogeneous platform = **CPU** + **GPU**
  - Most solutions work for other/multiple accelerators
- An application workload = an application + its input dataset
- Workload partitioning = workload distribution among the processing units of a heterogeneous system



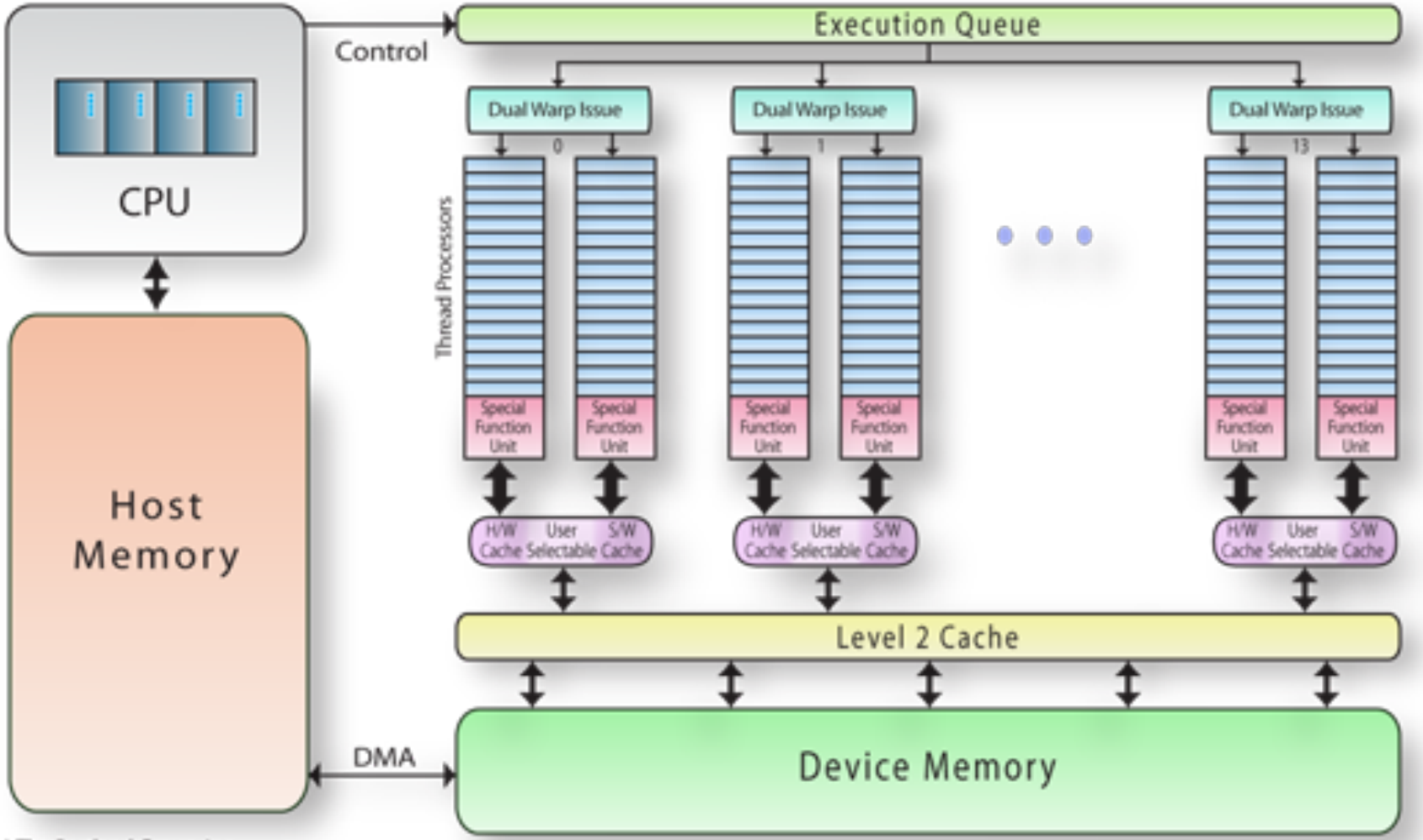
Generic multi-core CPU



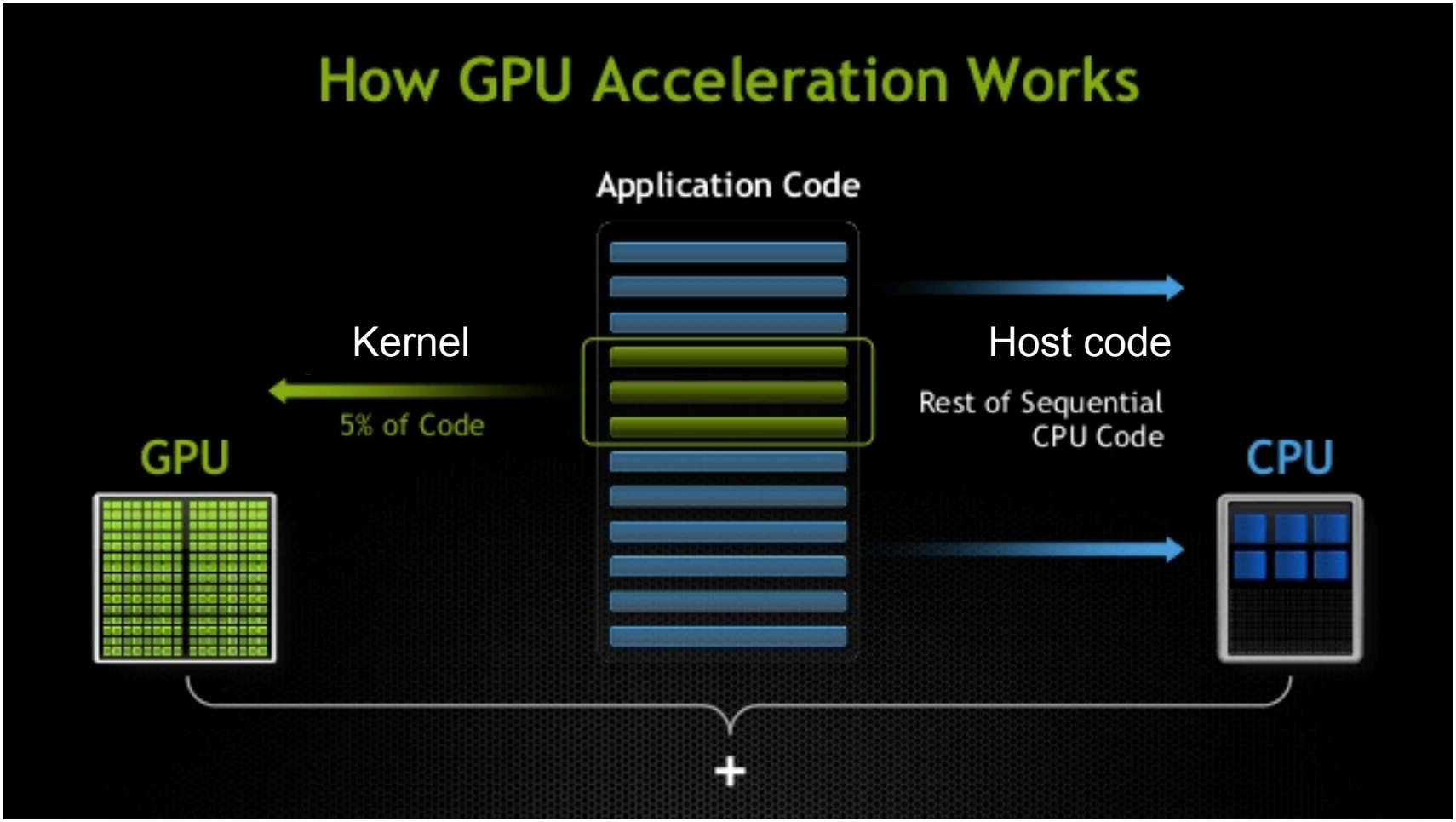
Programming models

- Pthreads + intrinsics
- TBB – Thread building blocks
  - Threading library
- OpenCL
  - To be discussed …
- OpenMP
  - Traditional parallel library
  - High-level, pragma-based
- Cilk
  - Simple divide-and-conquer model

A GPU Architecture



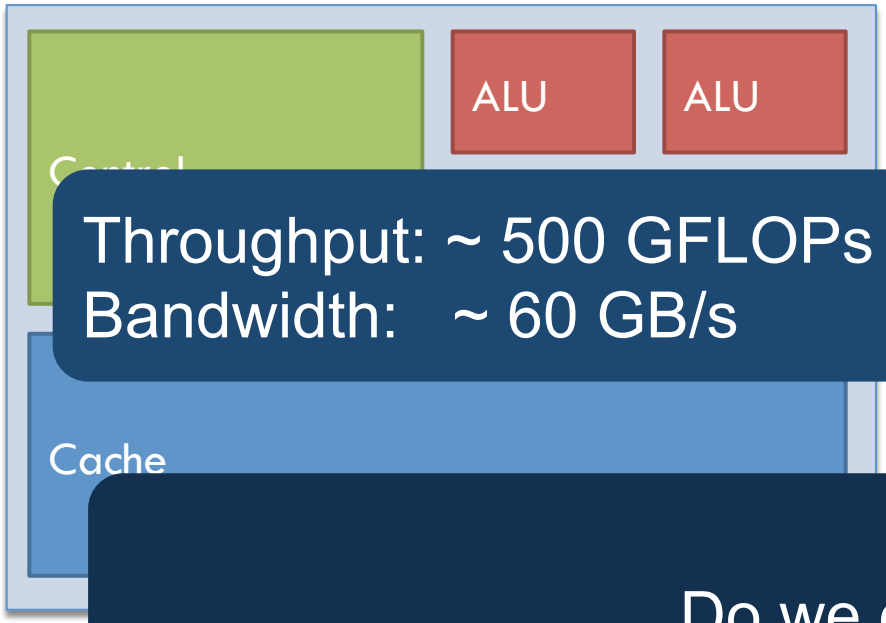
Offloading model



Programming models

- CUDA
  - NVIDIA proprietary
- OpenCL
  - Open standard, functionally portable across multi-cores
- OpenACC
  - High-level, pragma-based
- Different libraries, programming models, and DSLs for different domains

CPU vs. GPU



Throughput: ~ 500 GFLOPs
Bandwidth: ~ 60 GB/s

CPU

Low latency, high flexibility.
Excellent for irregular codes with limited parallelism.

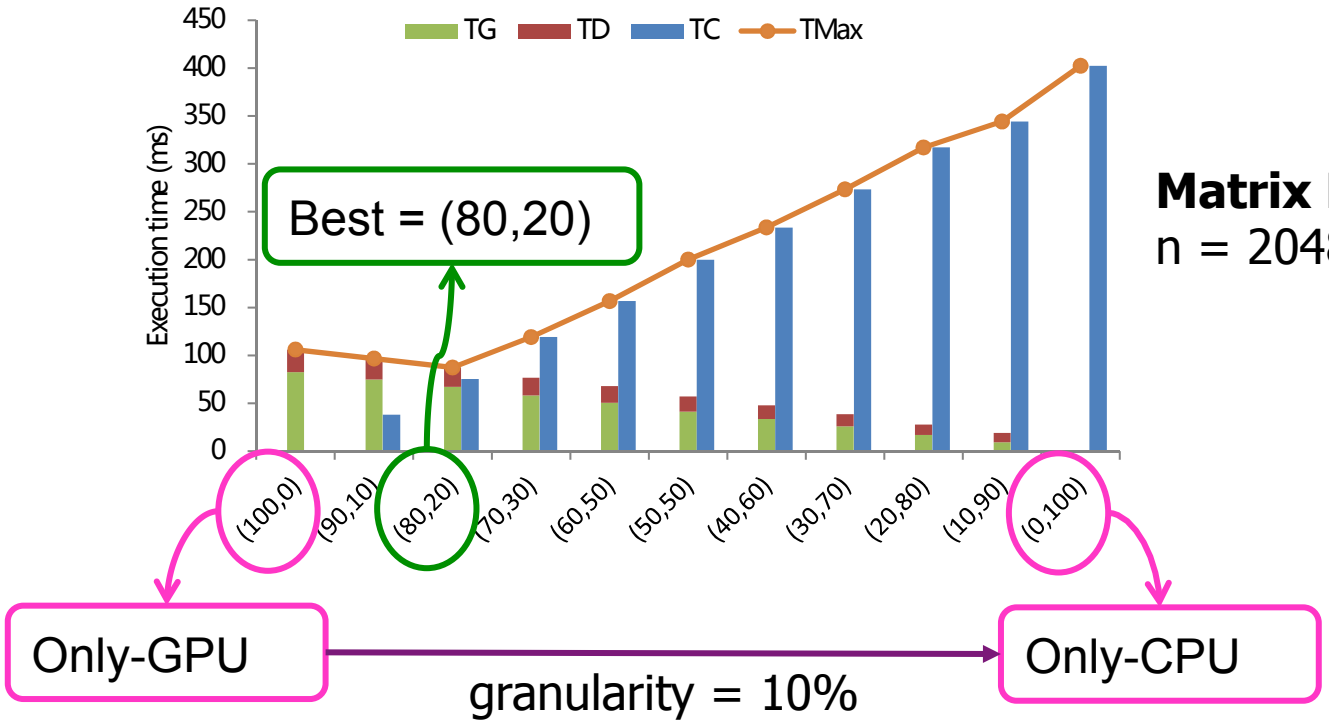
Do we gain anything?

GPU

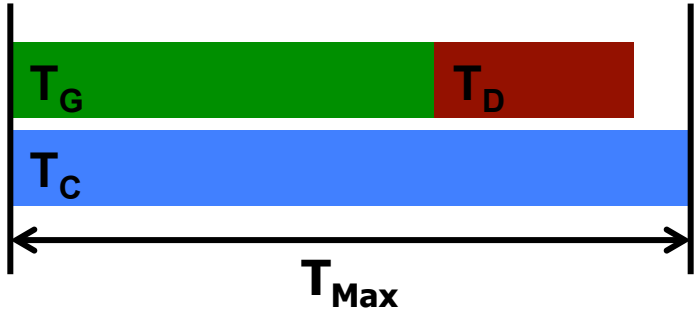
High throughput.
Excellent for massively parallel workloads.

Throughput: ~ 5.500 GFLOPs
Bandwidth: ~ 350 GB/s

Case studies

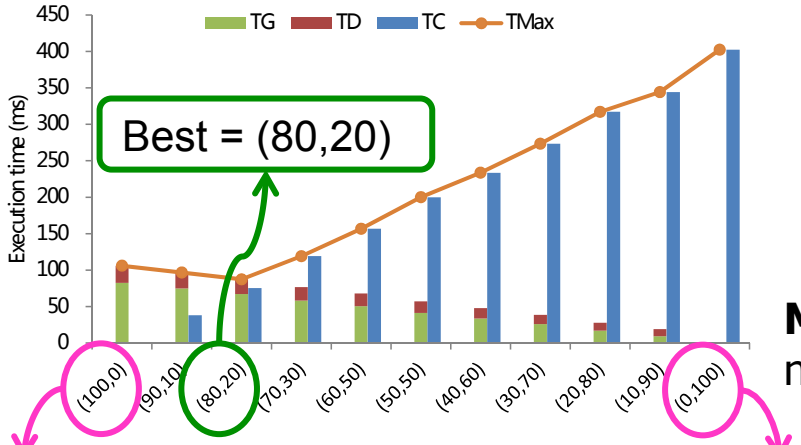


Matrix Multiplication (SGEMM)
n = 2048 x 2048 (48MB)



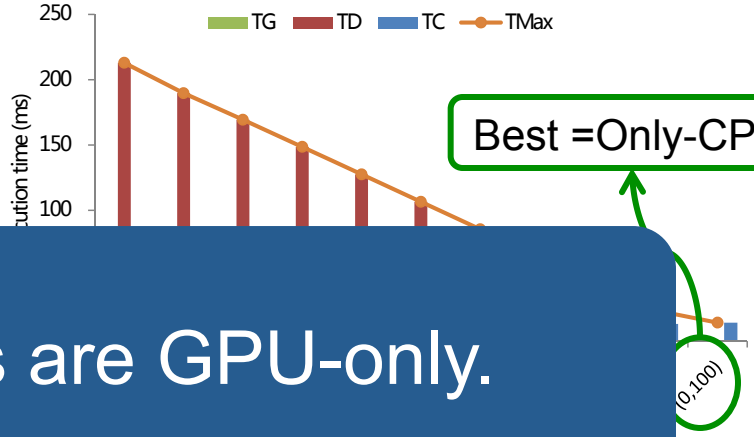
T_G = GPU kernel execution time
T_D = Data transfer time
T_C = CPU kernel execution time
T_{Max}= max(T_G+T_D, T_C)

Case studies



T_G = GPU kernel execution time
T_D = Data transfer time
T_C = CPU kernel execution time
T_{Max} = max(T_G+T_D, T_C)

Matrix Multiplication (SGEMM)
n = 2048 x 2048 (48MB)



Very few applications are GPU-only.

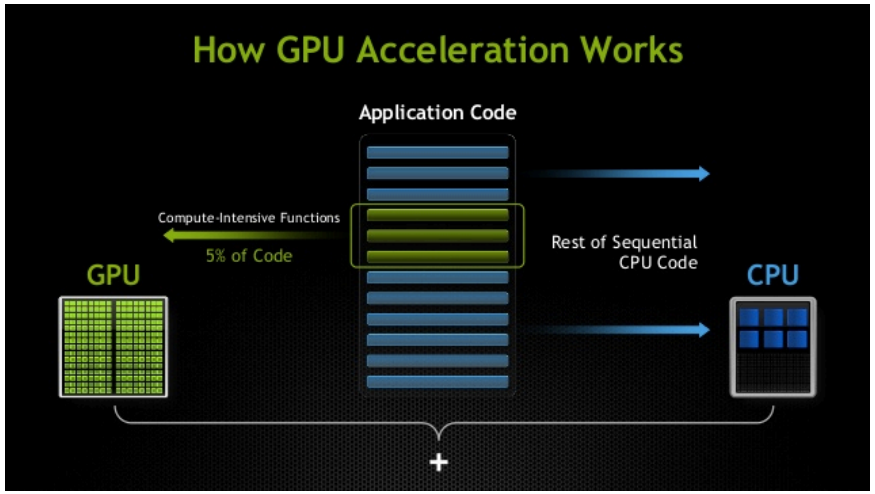
Separable Convolution (SConv)
n = 6144x6144 (432MB)

Dot Product (SDOT)
n = 14913072 (512MB)

Possible advantages of heterogeneity

- Increase performance
  - Both devices work in parallel
  - Decrease data communication
    - Which is often the bottleneck of the system
  - Different devices for different roles
- Increase flexibility and reliability
  - Choose one/all *PUs for execution
  - Fall-back solution when one *PU fails
- Increase energy efficiency
- Ch

Main challenges: **programming** and **workload partitioning**!



Challenge 1: Programming models

Programming models (PMs)

- Heterogeneous computing = a mix of different processors on the same platform.
- Programming
  - Mix of programming models
    - One(/several?) for CPUs – OpenMP
    - One(/several?) for GPUs – CUDA
  - Single programming model (unified)
    - OpenCL is a popular choice

Low level

OpenCL

High level

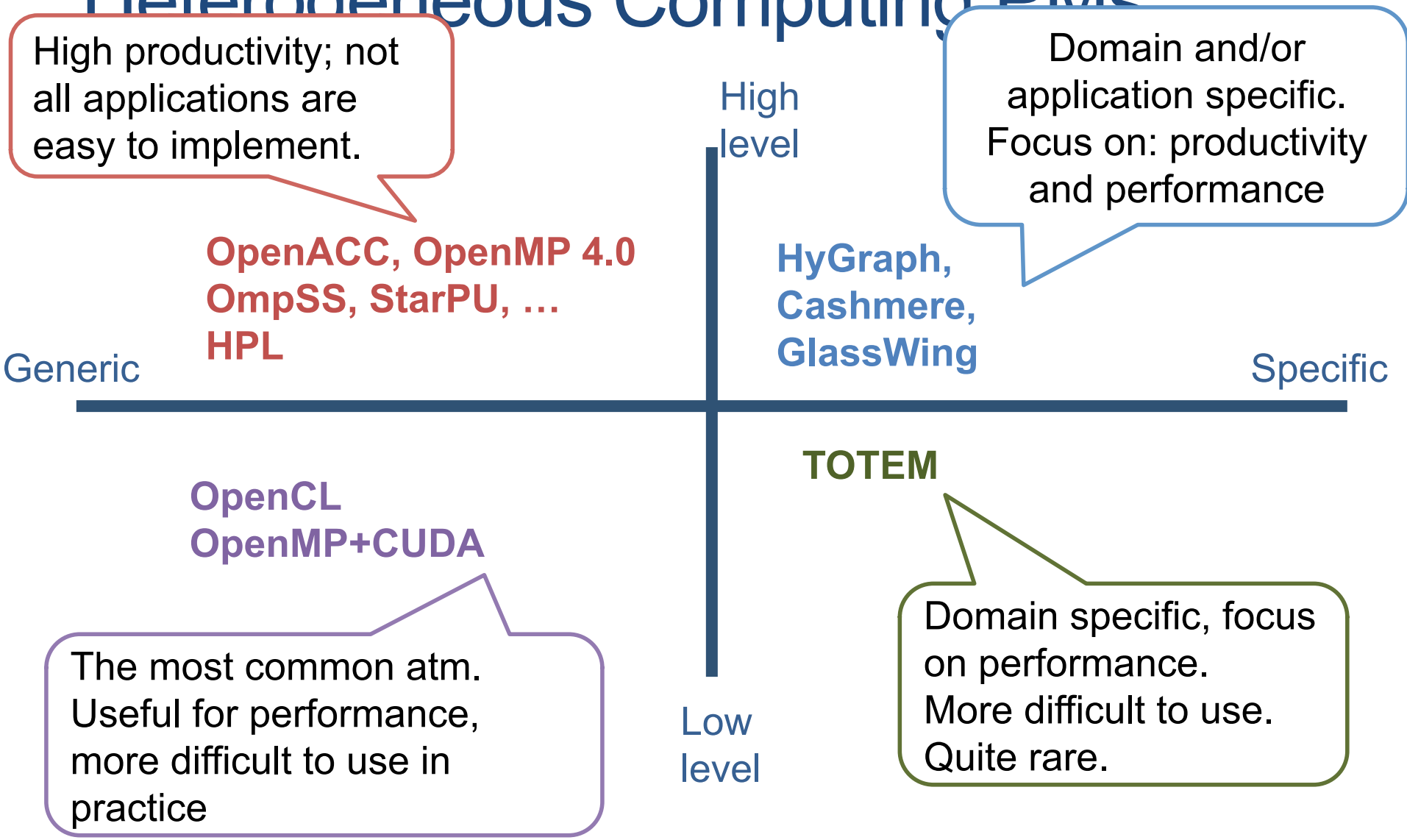
OpenACC
Directives for Accelerators

OpenMP 4.0

Heterogeneous Programming Library

OmpSs

Heterogeneous Computing PMs



Heterogeneous computing PMs

- CUDA + OpenMP/TBB
  - Typical combination for NVIDIA GPUs
  - Individual development per *PU
  - Glue code can be challenging
- OpenCL (KHRONOS group)
  - Functional portability => can be used as a unified model
  - Performance portability via code specialization
- HPL (University of A Coruna, Spain)
  - Library on top of OpenCL, to automate code specialization

Heterogeneous computing PMs

- StarPU (INRIA, France)
  - Special API for coding
  - Runtime system for scheduling
- OmpSS (UPC + BSC, Spain)
  - C + OpenCL/CUDA kernels
  - Runtime system for scheduling and communication optimization

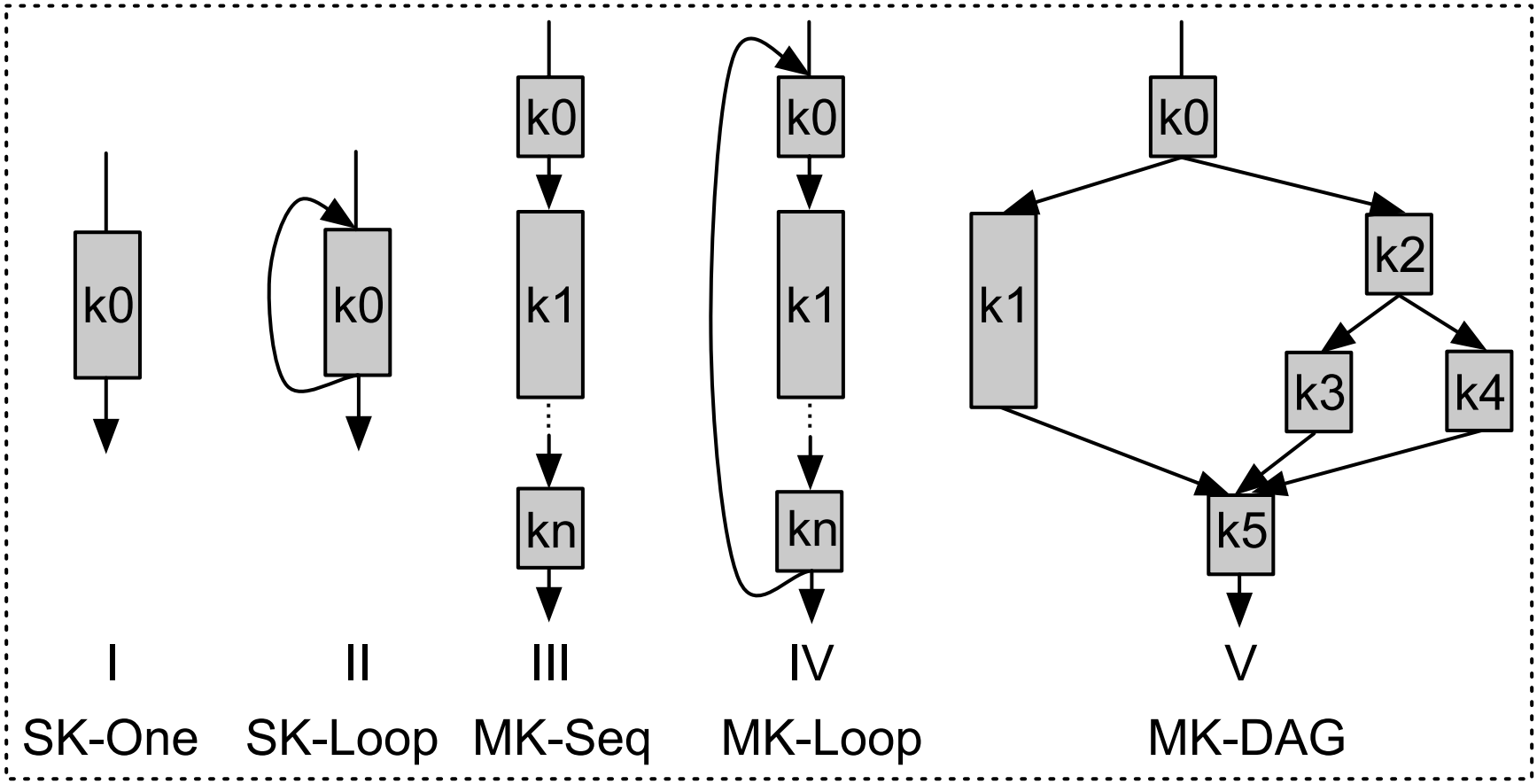
Heterogeneous computing PMs

- Cashmere (VU Amsterdam + NLeSC)
  - Dedicated to Divide-and-conquer solutions
  - OpenCL backend.
- GlassWing (VU Amsterdam)
  - Dedicated to MapReduce applications
- TOTEM (U. of British Columbia, Canada)
  - Graph processing
  - CUDA+Multi-threading
- HyGraph (TUDelft, UTwente, UvA, NL)
  - Graph processing
  - Based on CUDA+OpenMP

Challenge 2: Workload partitioning

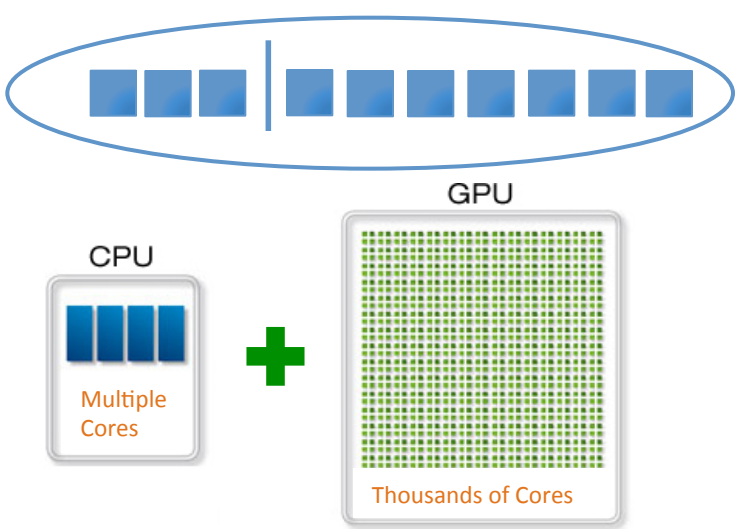
Workload

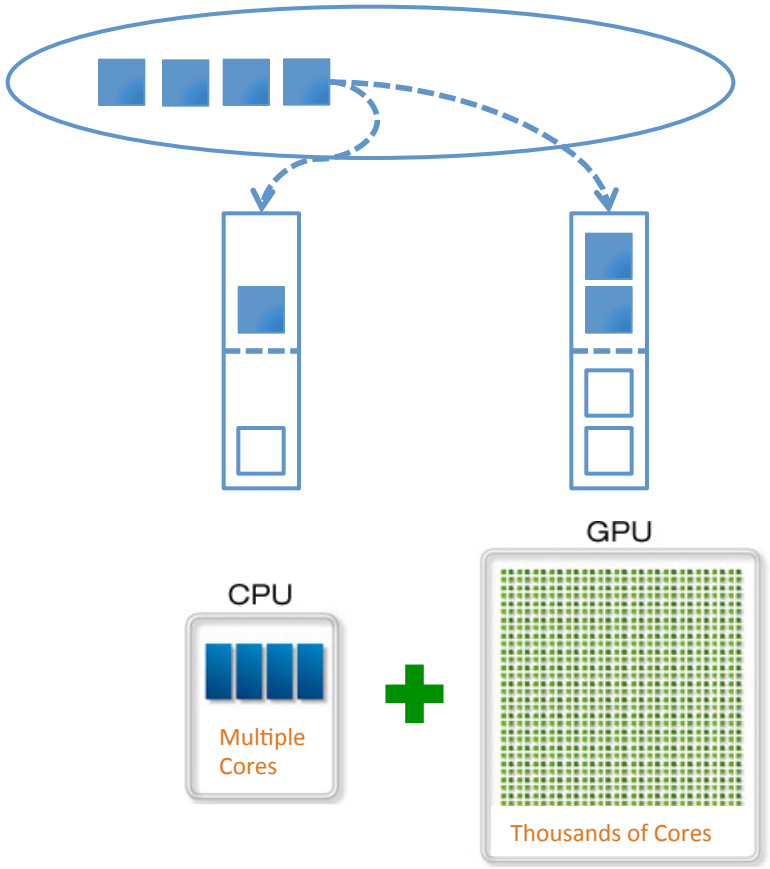
- DAG (directed acyclic graph) of “kernels”



Determining the partition

- Static partitioning (SP) vs. Dynamic partitioning (DP)



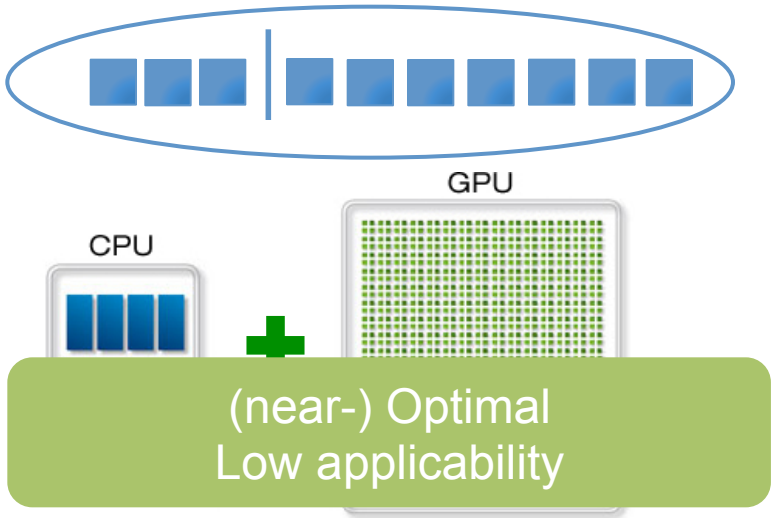


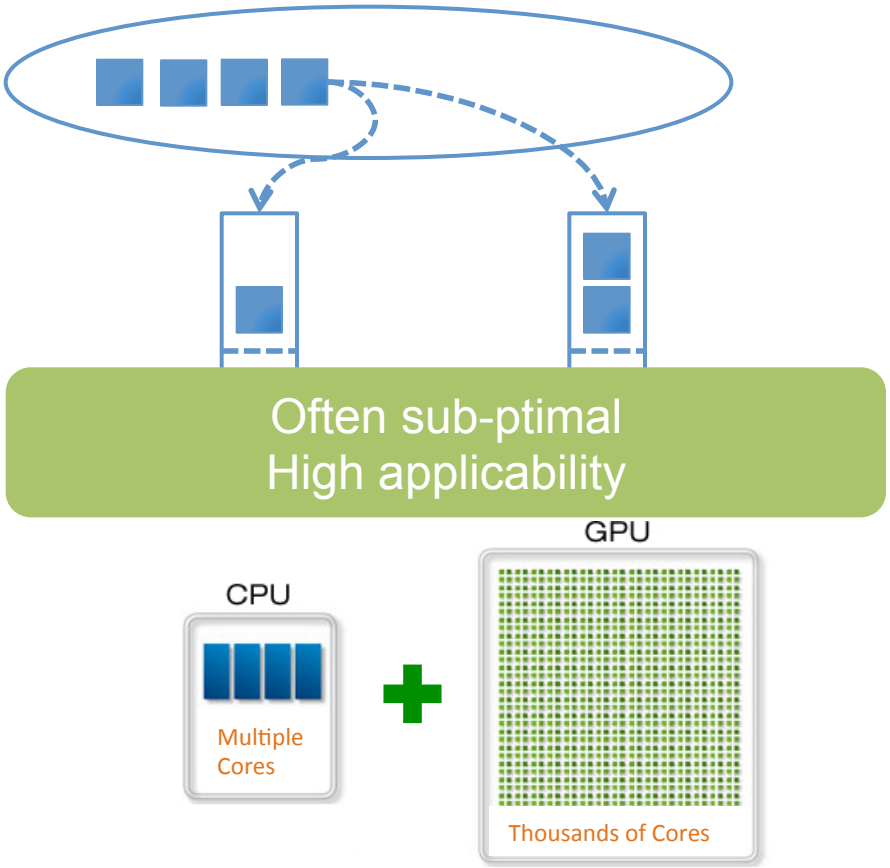
Static vs. dynamic

- Static partitioning
  - + can be computed before runtime => no overhead
  - + can detect GPU-only/CPU-only cases
  - + no unnecessary CPU-GPU data transfers
  - -- does not work for all applications
- Dynamic partitioning
  - + responds to runtime performance variability
  - + works for all applications
  - -- incurs (high) runtime scheduling overhead
  - -- might introduce (high) CPU-GPU data-transfer overhead
  - -- might not work for CPU-only/GPU-only cases

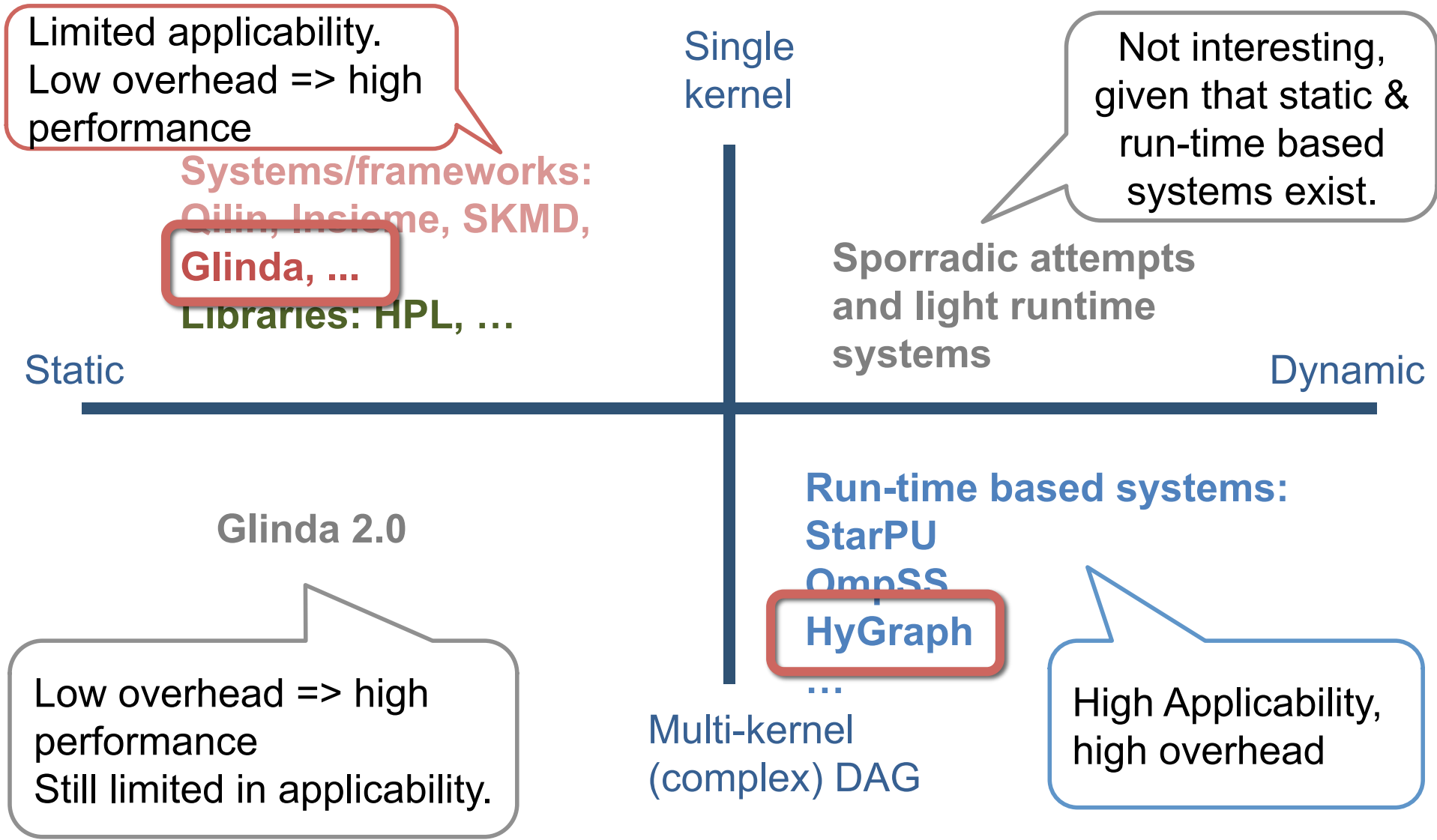
Determining the partition

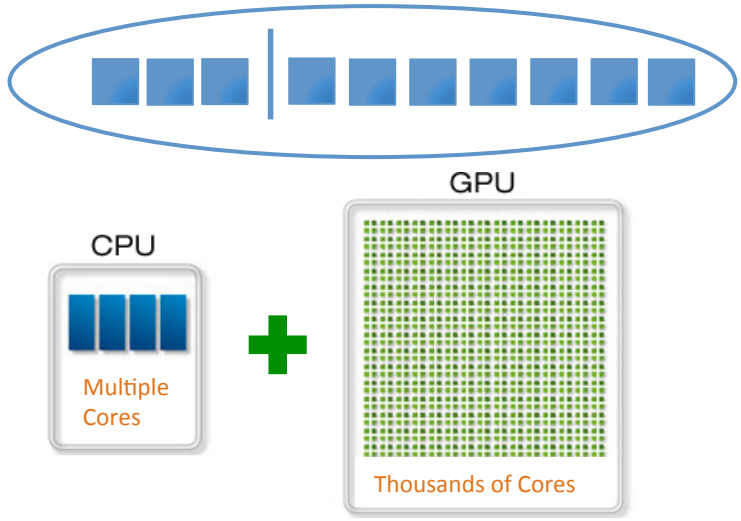
- Static partitioning (SP) vs. Dynamic partitioning (DP)





Heterogeneous Computing today





Static partitioning: Glinda

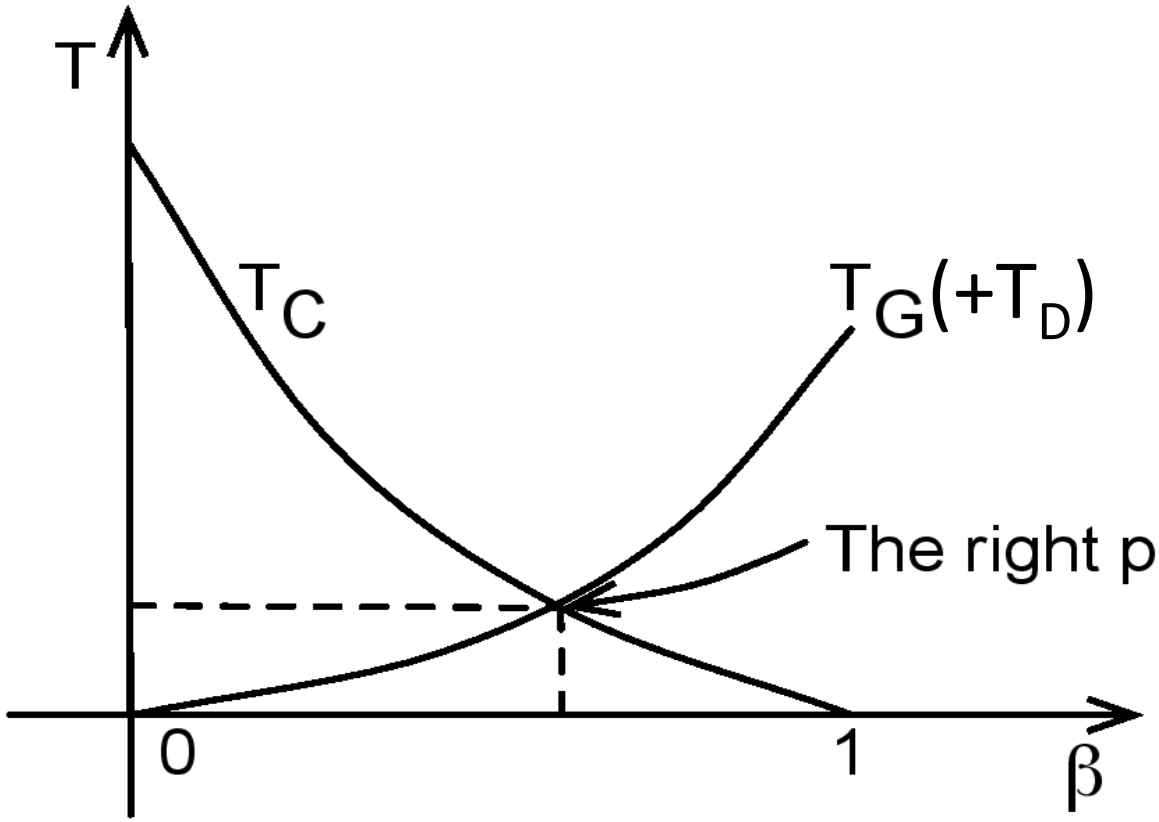
Glinda*

- Modeling the partitioning
  - The application workload
  - The hardware capabilities
  - The GPU-CPU data transfer
- Predict the optimal partitioning
- Making the decision in practice
  - Only-GPU
  - Only-CPU
  - CPU+GPU with the optimal partitioning

*Jie Shen et al., HPCC'14.
"Look before you Leap: Using the Right Hardware.
Resources to Accelerate Applications

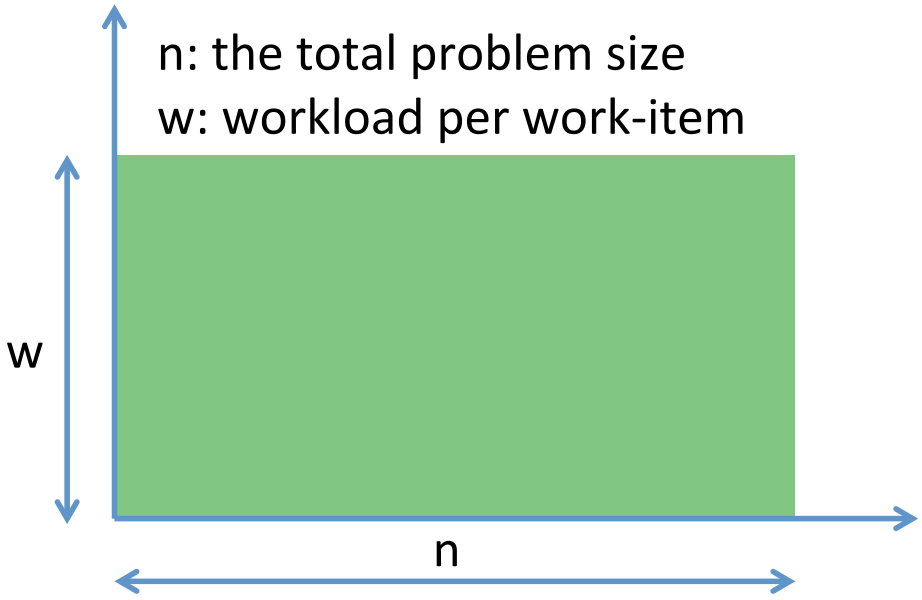
Modeling the partitioning

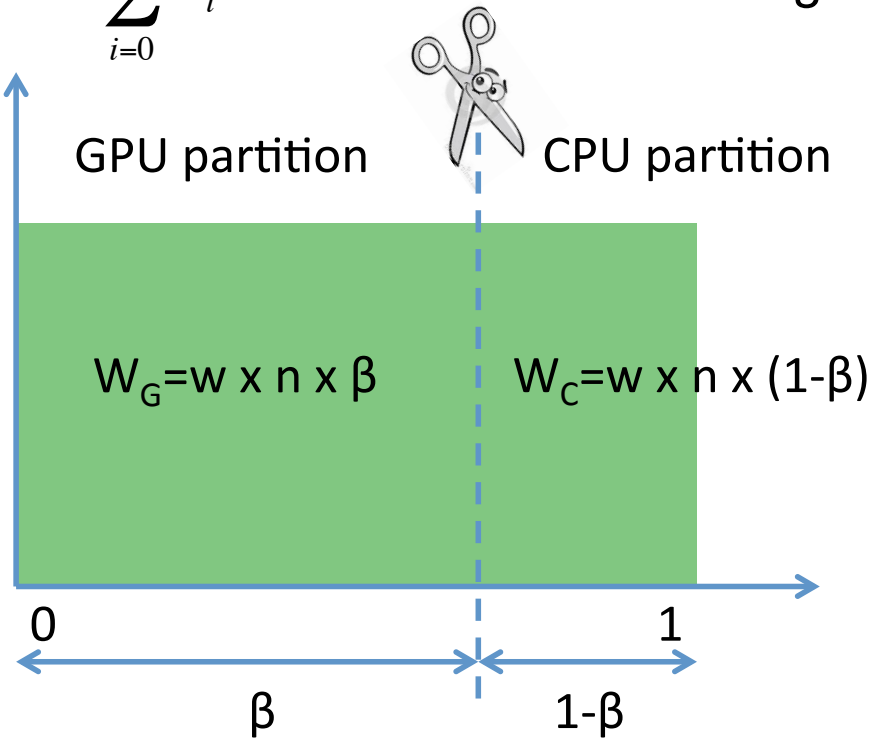
- Define the optimal (static) partitioning
  - β= the fraction of data points assigned to the GPU



Modeling the workload

$$W = \sum_{i=0}^{n-1} w_i \approx \text{the area of the rectangle}$$





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

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: total workload size

$W_C = (1-\beta) * W$ and $W_G = \beta * W$

P: processing throughput (*W*/second)

O: CPU-GPU data-transfer size

Q: CPU-GPU bandwidth (B/second)

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

R_{GC}

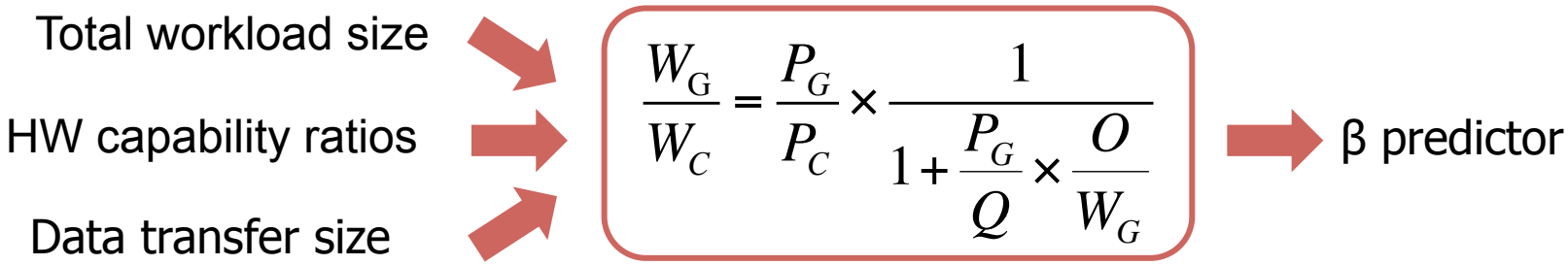
CPU kernel time vs.
GPU kernel time

R_{GD}

GPU data-transfer time vs.
GPU kernel execution time

Predicting the optimal partitioning

- Solving β from the equation



- There are three β predictors (by data transfer type)

$$\beta = \frac{R_{GC}}{1 + R_{GC}}$$

No data transfer

$$\beta = \frac{R_{GC}}{1 + \frac{v}{w} \times R_{GD} + R_{GC}}$$

Partial data transfer

$$\beta = \frac{R_{GC} - \frac{v}{w} \times R_{GD}}{1 + R_{GC}}$$

Full data transfer

Glinda outcome

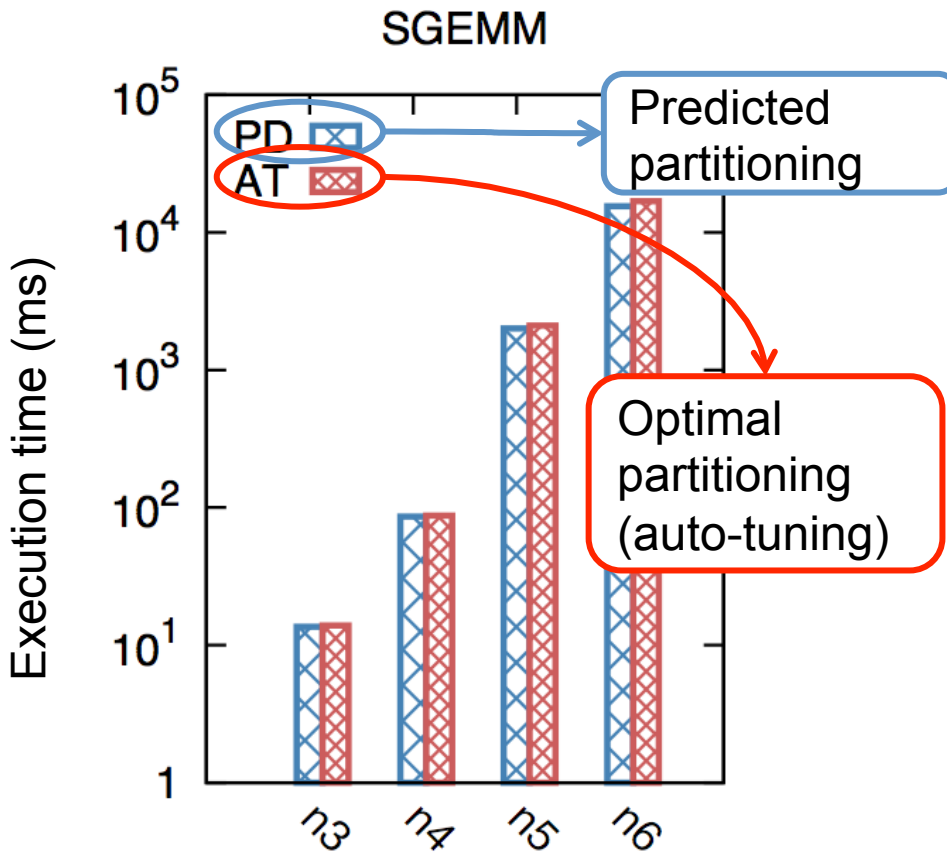
- (Any?) data-parallel application can be transformed to support heterogeneous computing

- A decision on the execution of the application
  - only on the CPU
  - only on the GPU
  - CPU+GPU
    - And the partitioning point

Results [1]

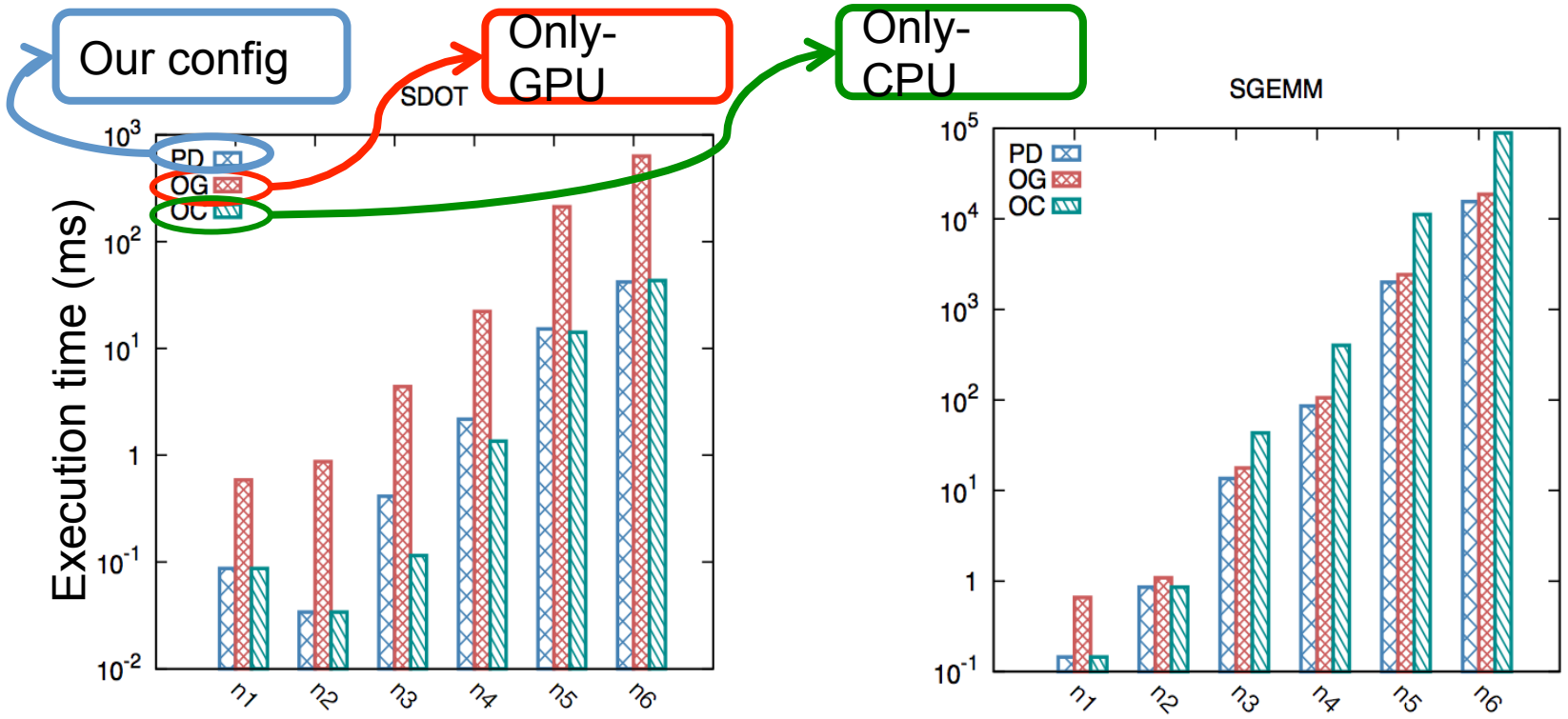
- Quality (compared to an oracle)
  - The right HW configuration in 62 out of 72 test cases
  - Optimal partitioning

  when CPU+GPU is selected



Results [2]

- Effectiveness (compared to Only-CPU/Only-GPU)
  - 1.2x-14.6x speedup
  - If taking GPU for granted, up to 96% performance will be lost



Related work

Glinda achieves similar or better performance than the other partitioning approaches with less cost

		Machine learning [1,2]	Qilin [3]	Ours	
				Online	Offline
Cost (in relative comparison, +++ large, ++ medium, + small, ~ minor, 0 zero)	Collection	+++	++/+ (depending on *m*)	~	+
	Training	+++/++	+	0	+
	Deployment	~ (including code analysis)	0	0	0
	Adaption	+++	++/+ (depending on *m*)	~	+
Performance		[1]: 85% [2] with SVM: 83.5% [2] with ANN: 87.5% (of the approximated optimal)	94% (of the approximated optimal)	91% (of the optimal)	90% (of the optimal)

More advantages …

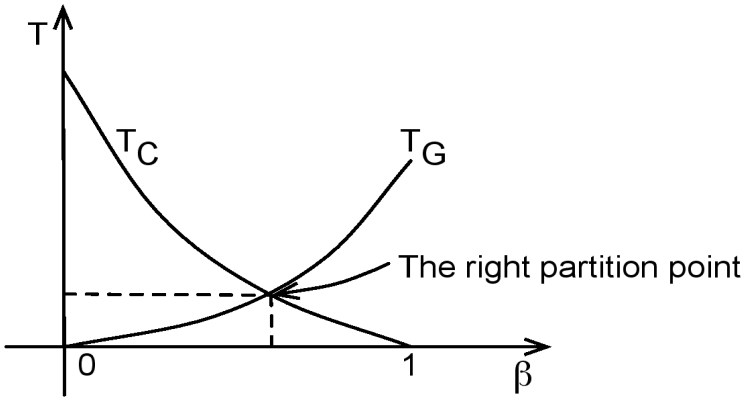
- Support different types of heterogeneous platforms
  - Multi-GPUs, identical or non-identical
- Support different profiling options suitable for different execution scenarios
  - Online or offline
  - Partial of full
- Determine not only the optimal partitioning but also the best hardware configuration
  - Only-GPU / Only-CPU / CPU+GPU with the optimal partitioning
- Support both balanced and imbalanced applications

Glinda: limitations?

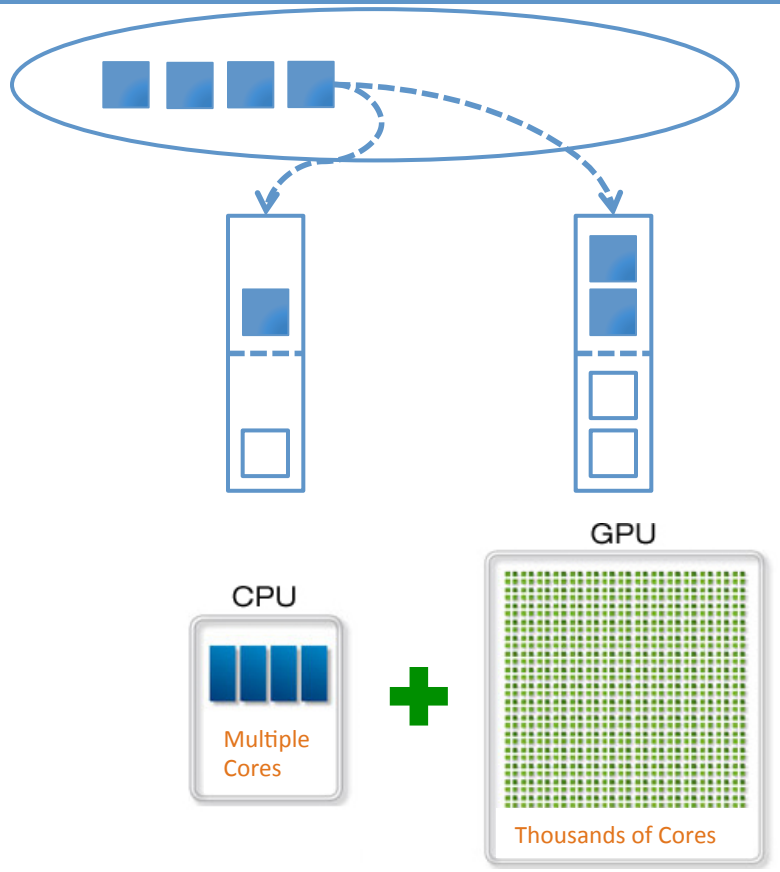
- Regular applications only >> NO, we support imbalanced applications, too.
- Single kernel only? >> NO, we support two types of multi-kernel applications, too.
- Single GPU only? >> NO, we support multiple GPUs/ accelerators on the same node
- Single node only? >> YES – for now .

What about energy?

- The Glinda model can be adapted to optimize for energy
  - Ongoing work at UvA
- Not yet clear whether “optimality” can be so easily determined

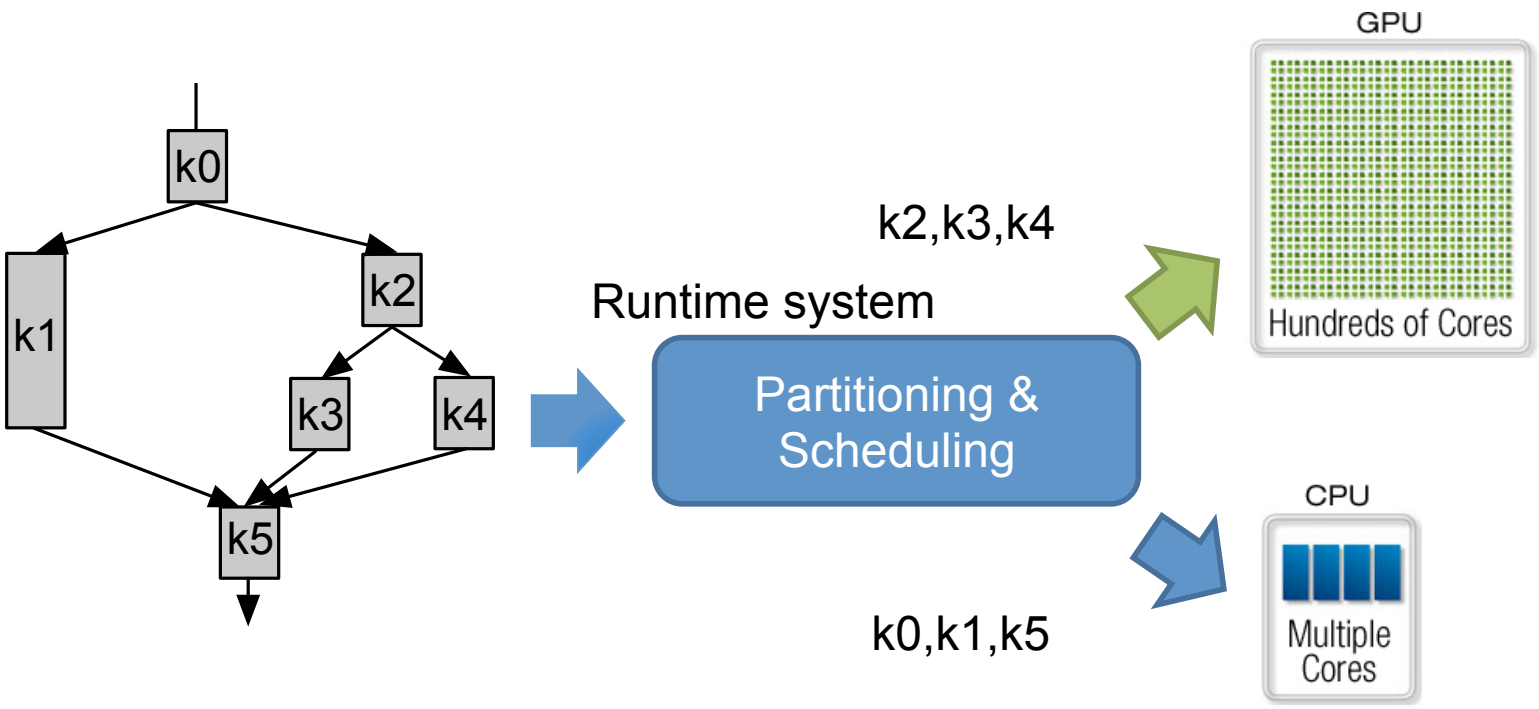


- Finding the right energy models
  - We started with statistical models
  - We plan to use existing analytical models in the next stage
  - Would probably be more limitations regarding the workload types…



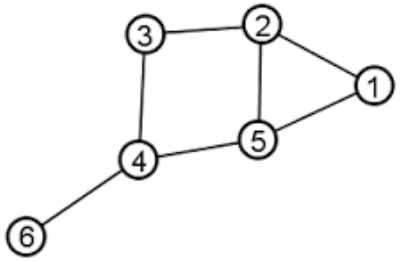
Dynamic partitioning: HyGraph

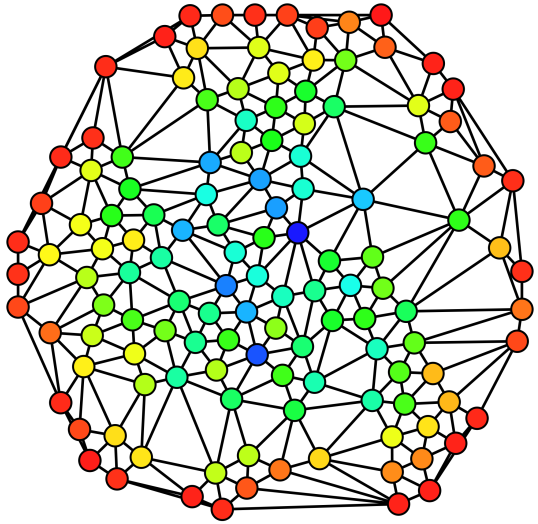
Dynamic partitioning: StarPU, OmpSS

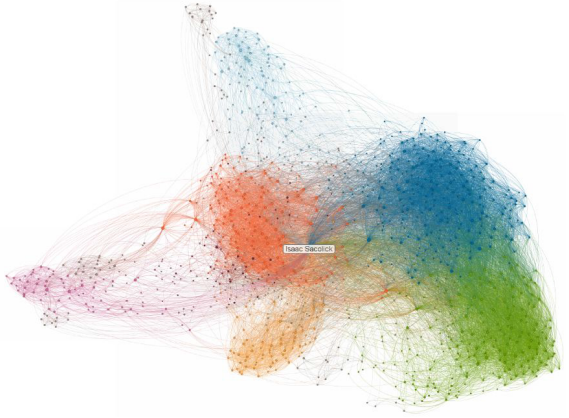


Our own take on it: graph processing.

Graph processing







Time

- Graphs are increasing in size and complexity
  - Facebook 1.7B users, LinkedIn ~500M users, Twitter ~400M users
- We need HPC solutions for graph processing

CPU or GPU?

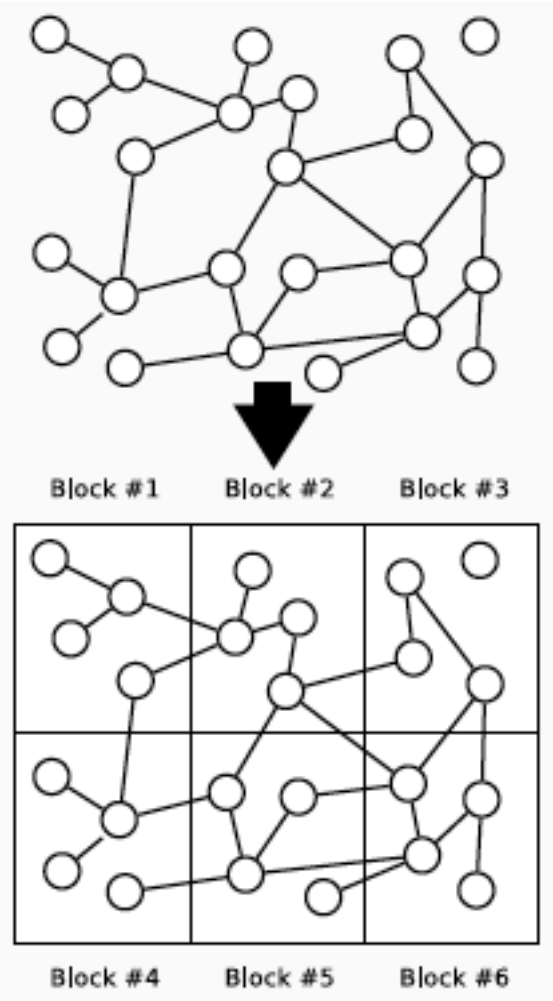
- Graph processing …
  - Many vertices => Massive parallelism.
  - Some vertices inactive => Irregular workload.
  - Varying vertex degree => Irregular tasks.
  - Low arithmetic intensity => Bandwidth limited.
  - Irregular access pattern => Poor caching.

- Ideal platform?
  - CPU: few cores, deep caches, asymmetric processing.
  - GPU: many cores, latency hiding, data-parallel processing.

No clear winner => Attempt CPU+GPU!

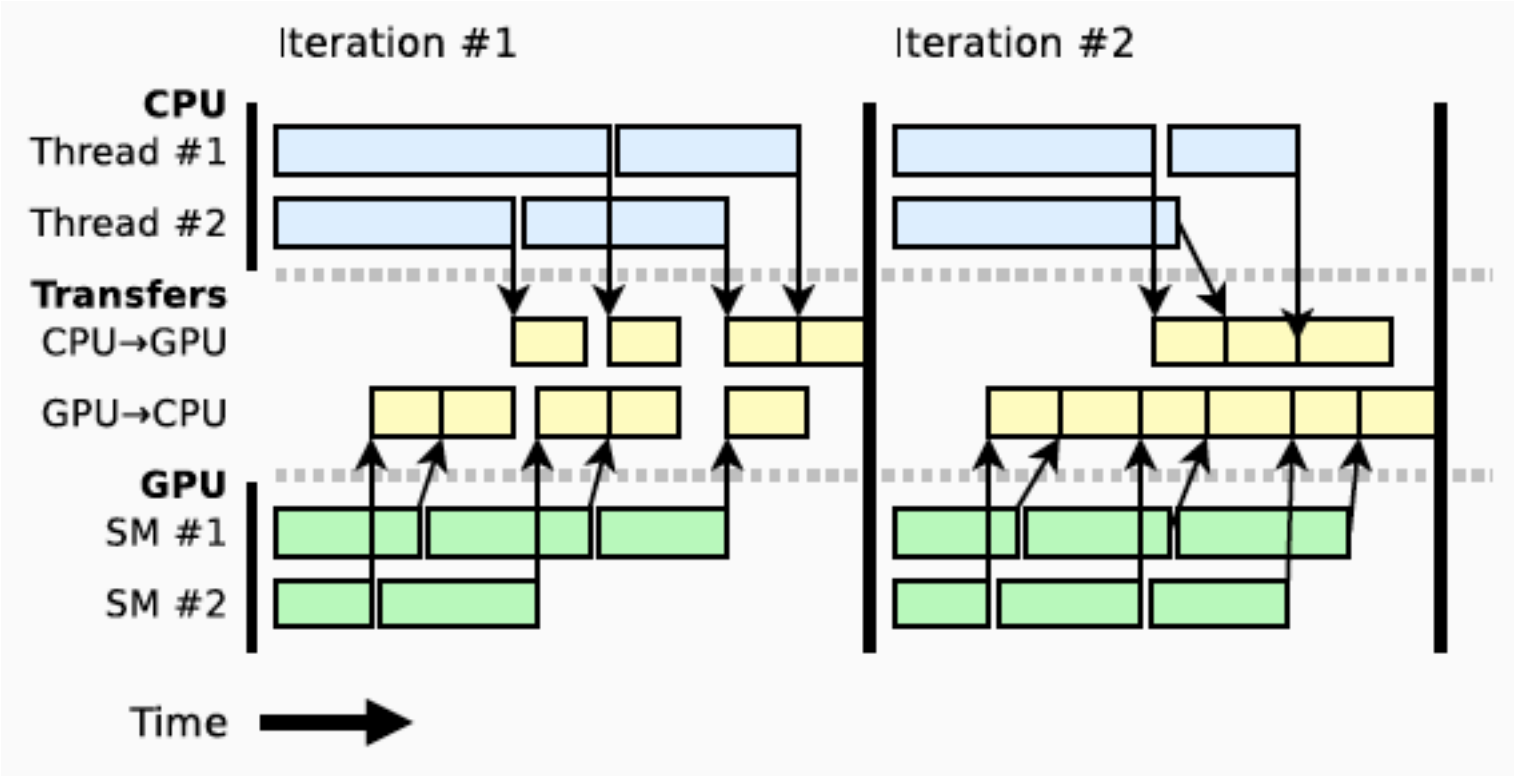
Design of HyGraph

- HyGraph: Dynamic scheduling
  - Replicate vertex states on CPU and GPU.
- Divide vertices into fixed-sized segments called blocks.
- Processing vertices inside block corresponds to one task.
  - CPU: 1 thread per block.
  - GPU: 1 SM (=1024 threads) per block



Design of HyGraph

- Overlap of computation with communication



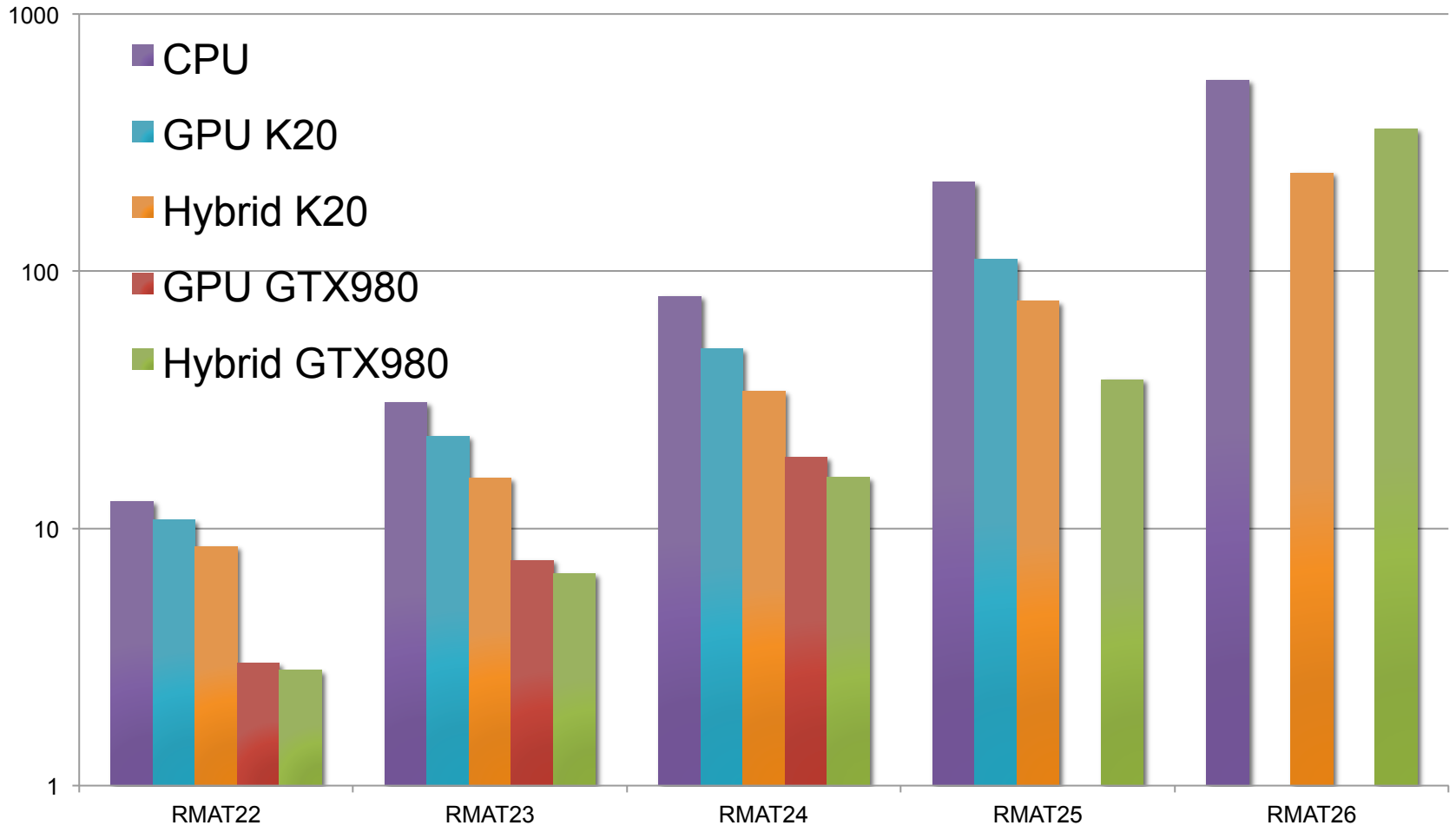
Performance evaluation

http://www.cs.vu.nl/das5/

- 2 GPUs presented
  - Kepler : K20
  - Maxwell : GTX980
- 1 CPU
  - Dual 8-core CPUs , 2.4GHz
- Three algorithms
  - **PageRank (500 iterations)**
- 10 datasets
  - 5 synthetic ones presented here
- PRELIMINARY results!

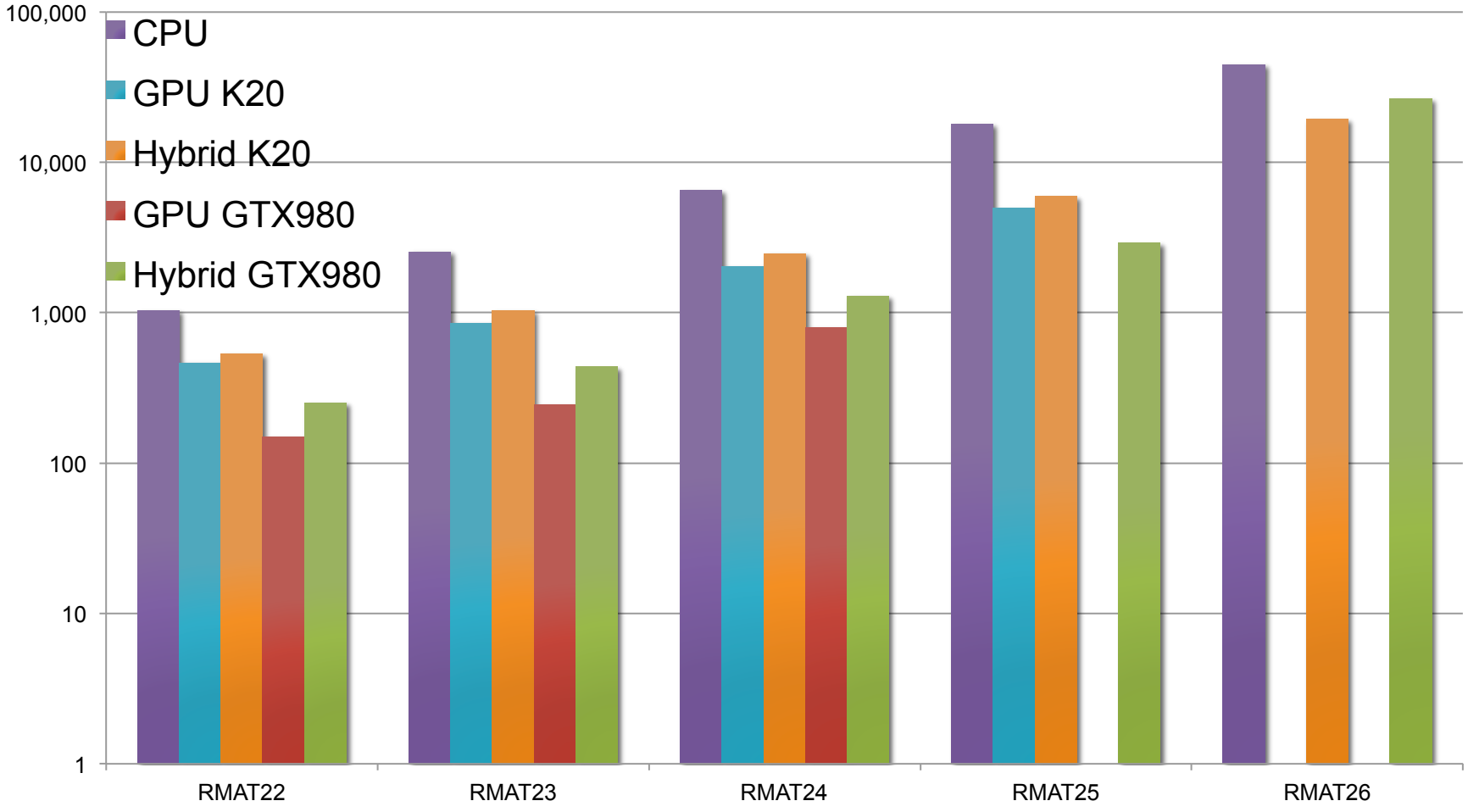
	|V|	|E|
RMAT22	2.4 M	64.2 M
RMAT23	4.6 M	129.3M
RMAT24	8.9 M	260.1M
RMAT25	17.1 M	523.6M
RMAT26	33.0 M	1051.9M

Preliminary results: execution time [s]



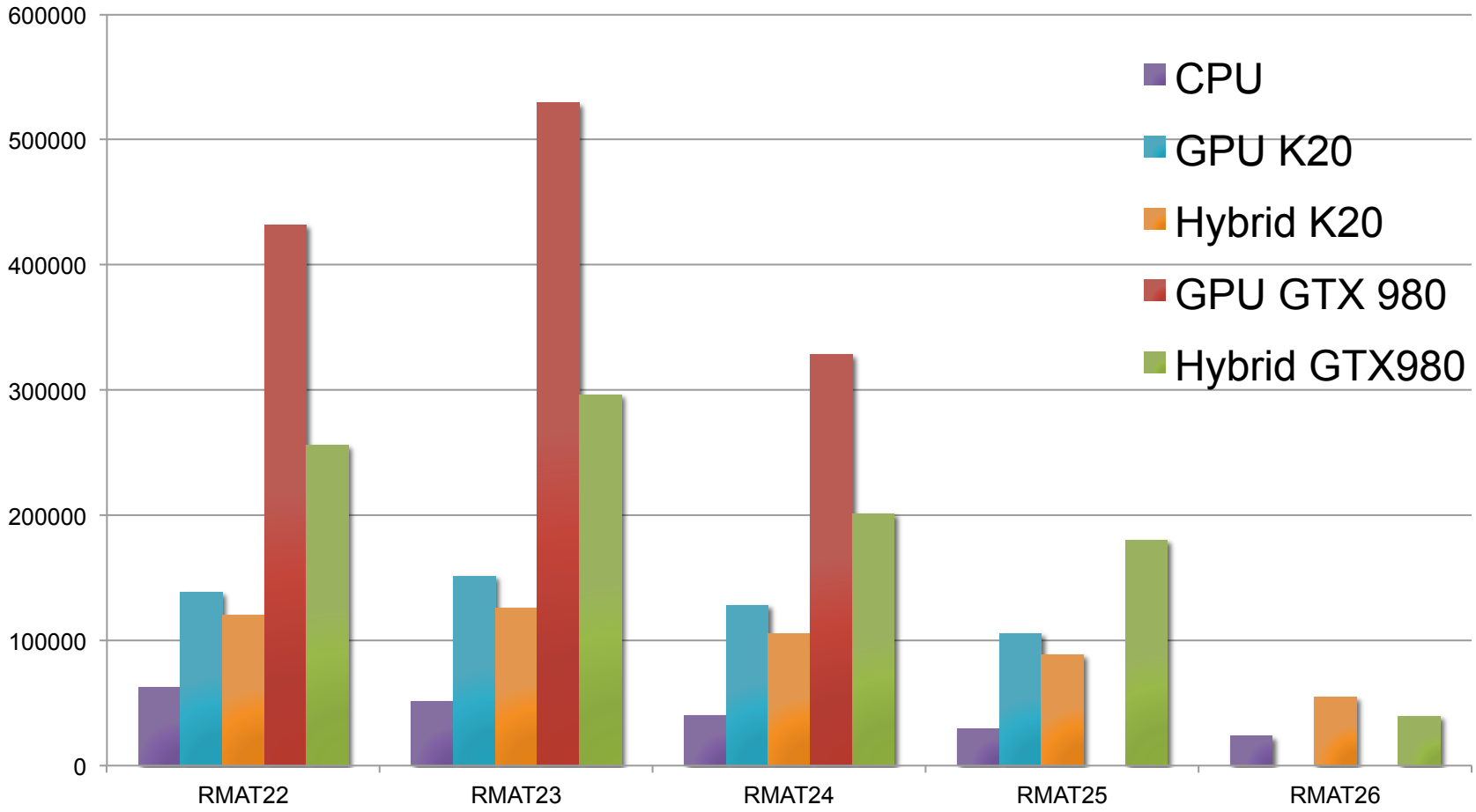
Lower is better

Preliminary results: Energy [Joule]



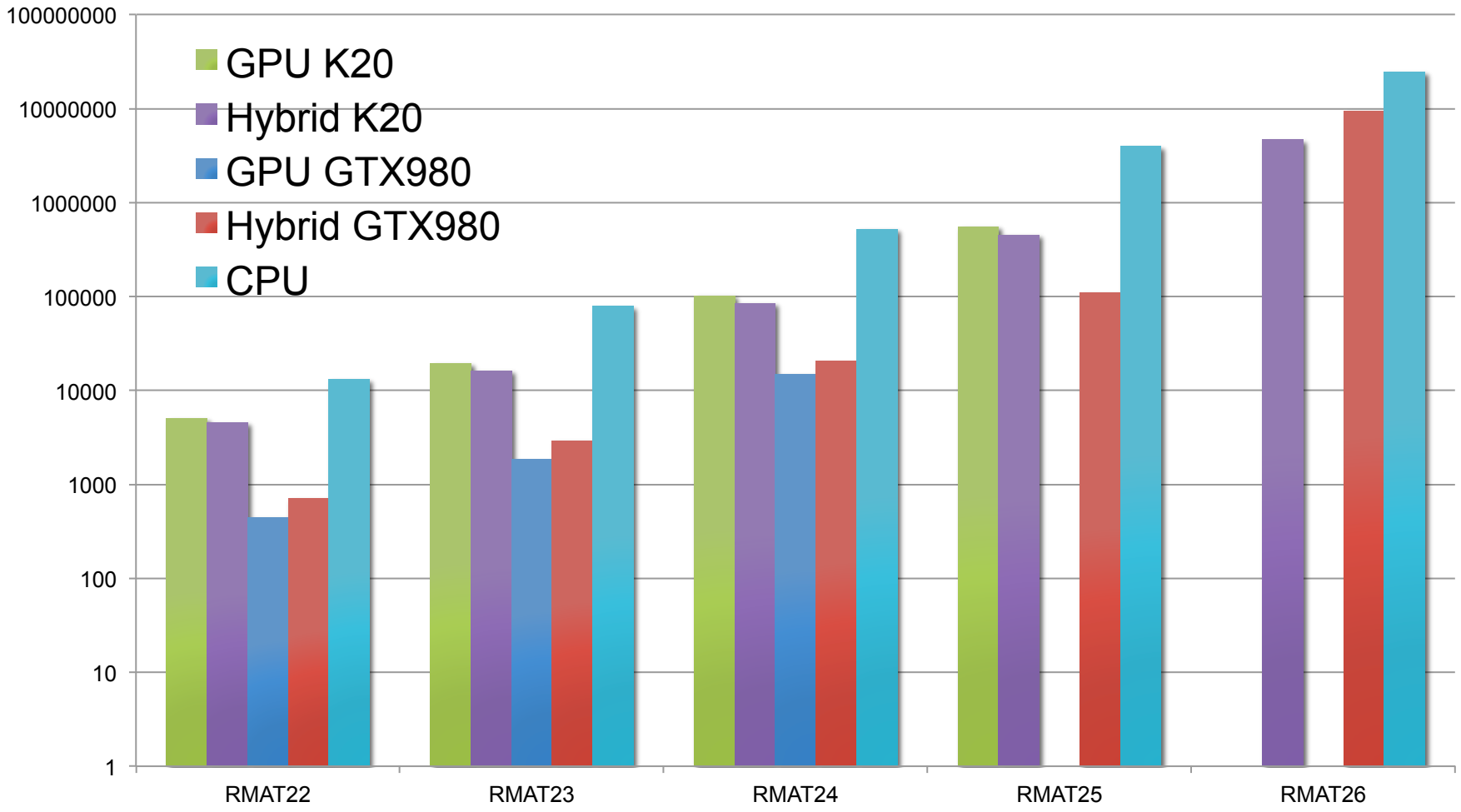
Lower is better

Results [3]: TEPS / Watt



Higher is better (TEPS = traversed edges per s)

Results [4]: EDP



Lessons learned

- For irregular applications…
  - Heterogeneous computing helps performance
  - Heterogeneous computing might help energy efficiency
    - For K20, it's pretty good
    - For GTX980 …. GPU wins
- Cannot generalize results to other types of graphs …
  - Both energy and performance are "unpredictable"
- Is hybrid dynamic processing feasible for energy efficiency ?!
  - Open question, still …

Instead of summary

Take home message [1]

- Heterogeneous computing works!
  - *More resources.*
  - *Specialized resources.*
- Most efforts in static partitioning and run-time systems
  - Glinda = static partitioning for single-kernel, data parallel applications
    - Now works for multi-kernel applications, too
  - StarPU, OmpSS = run-time based dynamic partitioning for multi-kernel, complex DAG applications
- Domain-specific efforts, too
  - HyGraph - graph processing
  - Cashmere – divide-and-conquer, distributed

Take home message [2]

to the office

- Choose a system based on your application scenario:
  - Single-kernel vs. multi-kernel
  - Massive parallel vs. Data-dependent
  - Single run vs. Multiple run
  - Programming model of choice
- There are models to cover combinations of these choices!
  - No framework to combine them all – food for thought?

Take home message [3]

- GPUs are energy efficient …
  - On their own
- Hybrid processing helps performance
  - … but for energy efficiency we are heavily application, workload, and hardware specific
- Irregular applications remain the cornerstone of *both* performance and energy …

Current/Future research directions

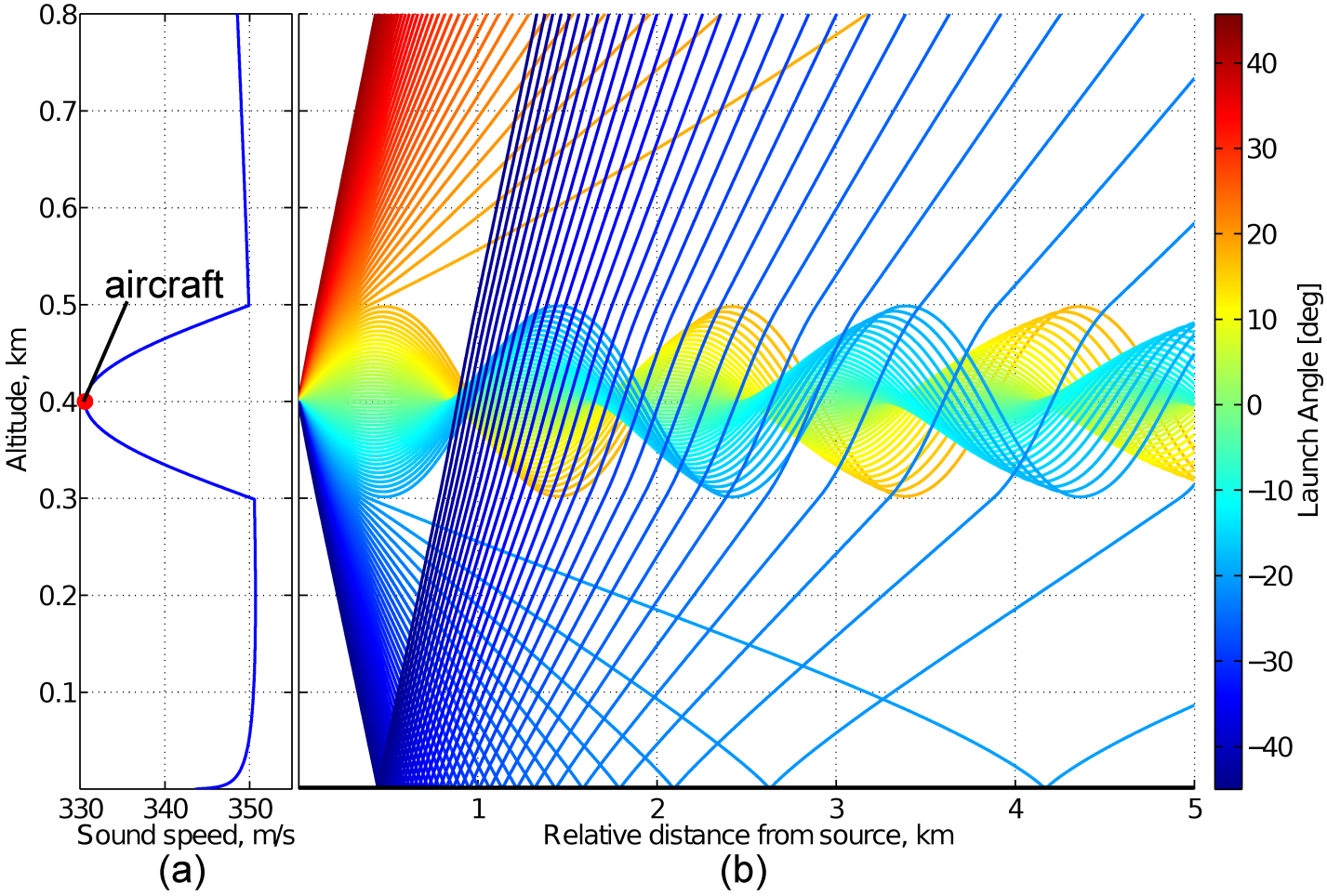
- More heterogeneous platforms
- Extension to more application classes
  - Multi-kernel with complex DAGs
  - (streaming applications, graph-processing applications)
- Integration with distributed systems
  - Intra-node workload partitioning + inter-node workload scheduling
- Extension of the partitioning model for energy

Backup slides

1 2

Imbalanced app: Sound ray tracing

Example 4: Sound ray tracing

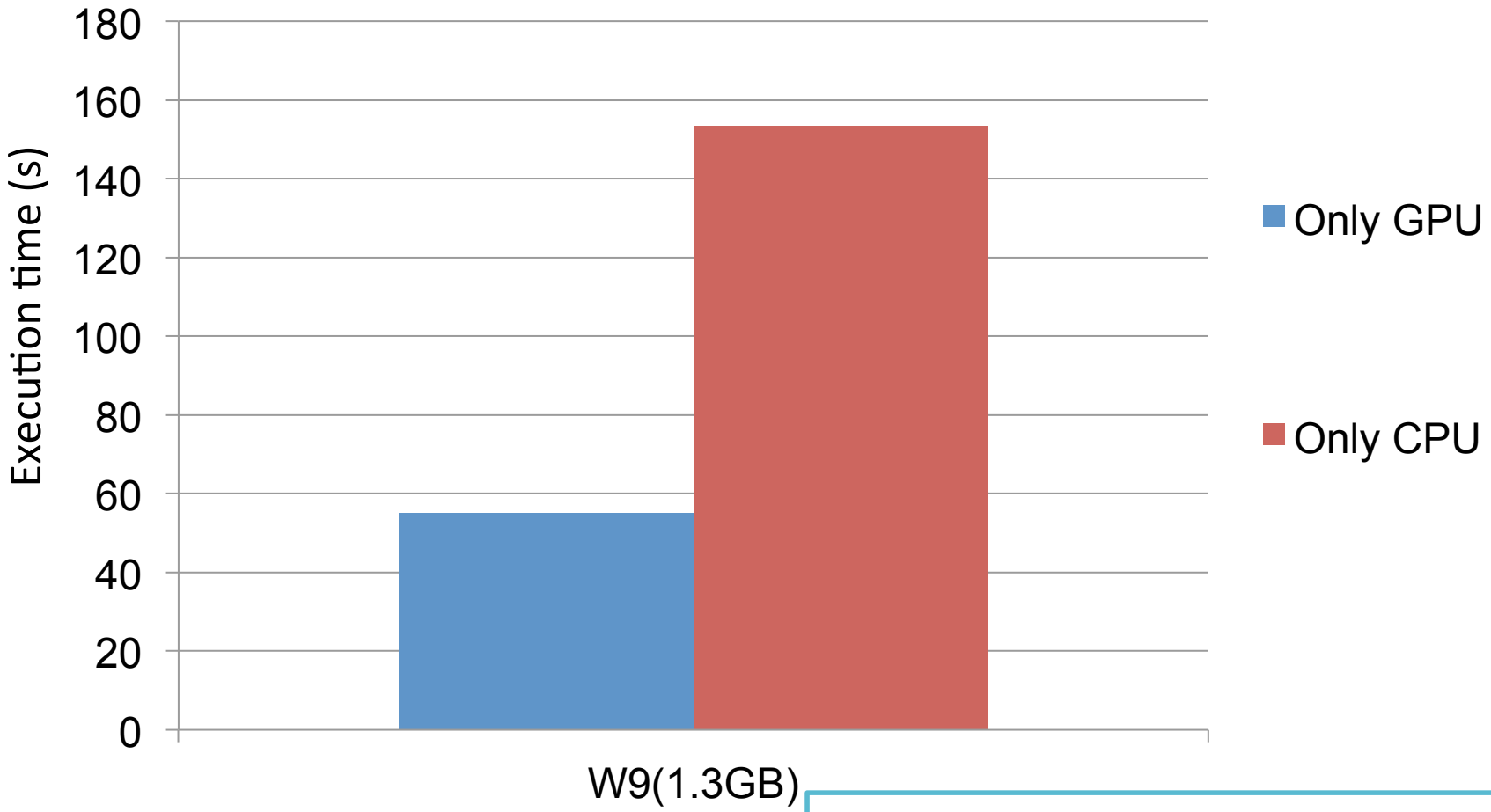


Which hardware?

- Our application has …
- Massive data-parallelism …
- No data dependency between rays …
- Compute-intensive per ray …

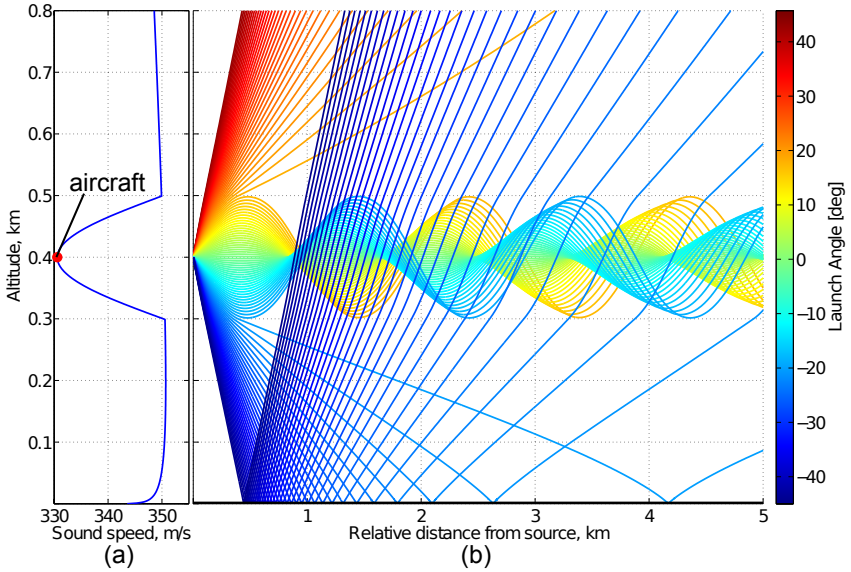
- … clearly, this is a perfect GPU workload !!!

Results [1]

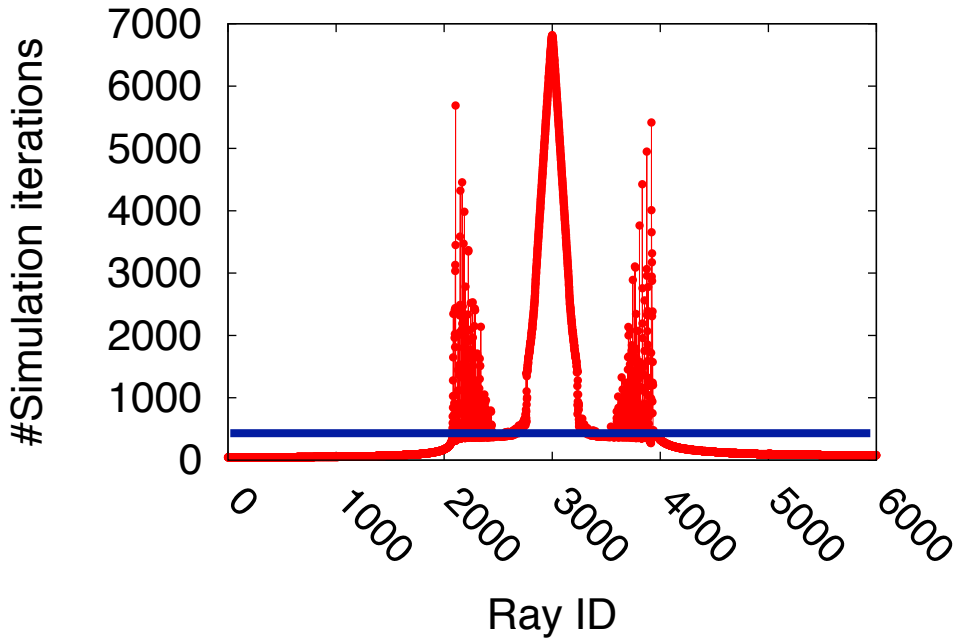


Only 2.2x performance improvement!
We expected 100x …

Workload profile

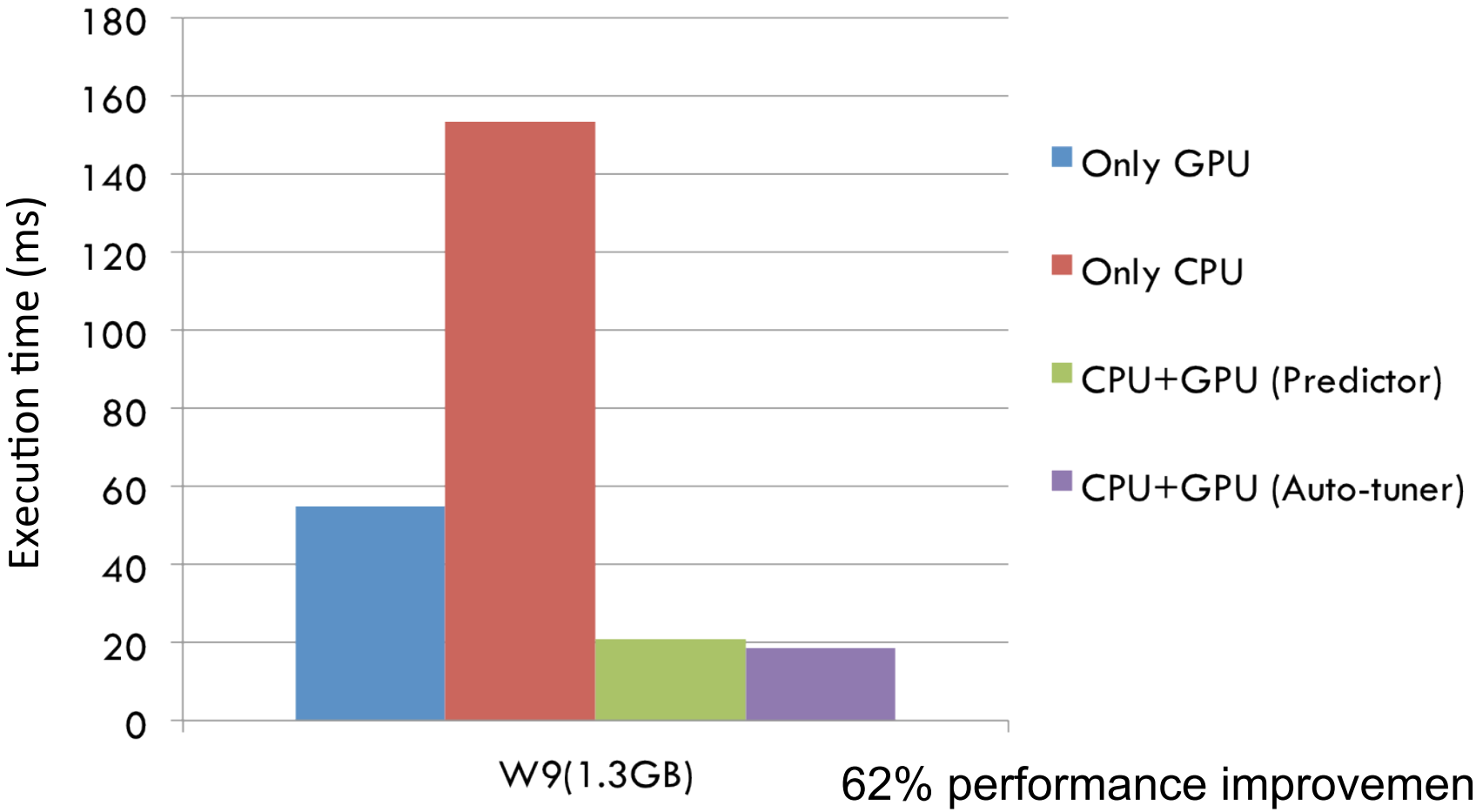


Peak
Processing iterations: ~7000



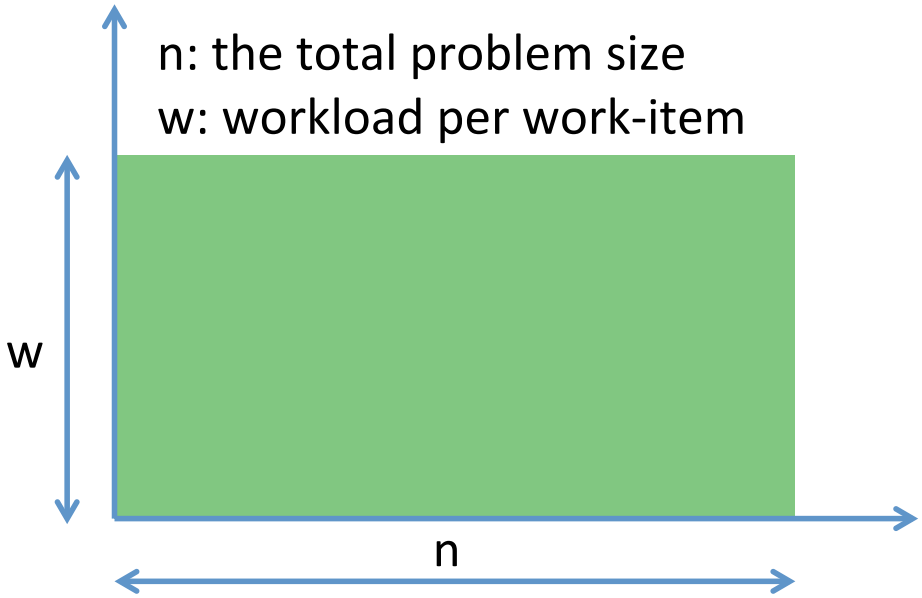
Bottom
Processing iterations: ~500

Results [2]



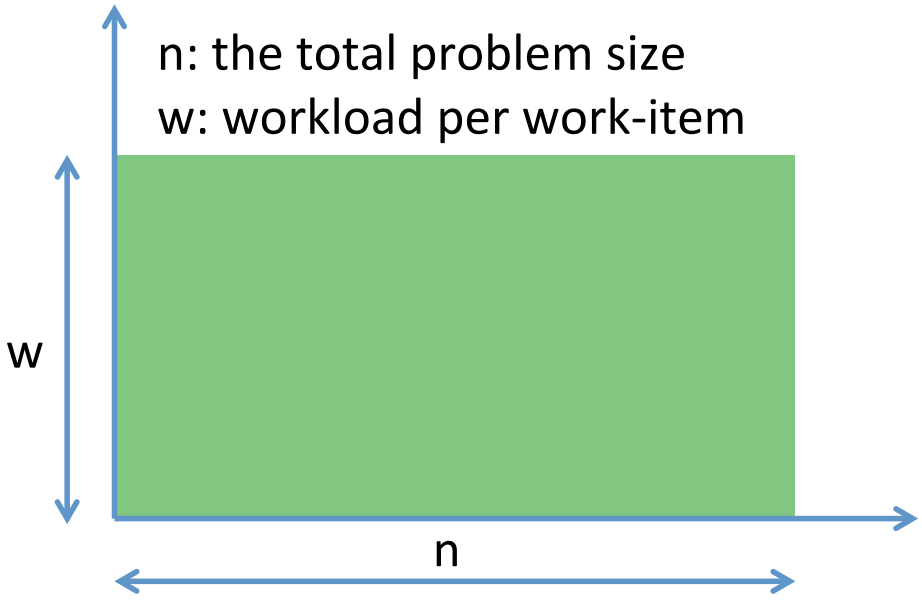
62% performance improvement compared to “Only-GPU”

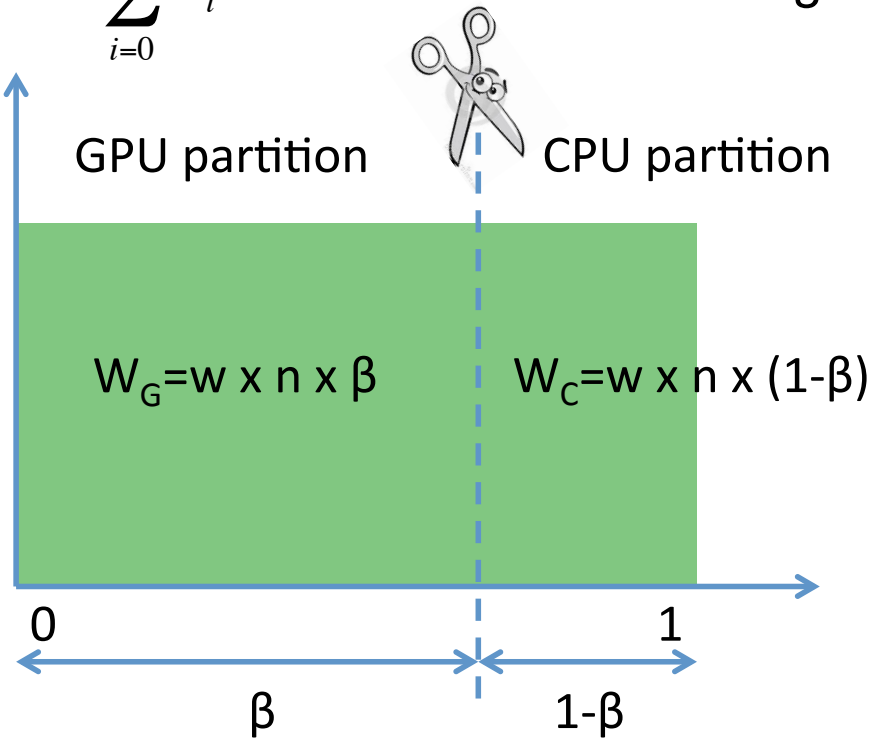
Model the app workload



Model the app workload

$$W = \sum_{i=0}^{n-1} w_i \approx \text{the area of the rectangle}$$





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

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: total workload size
W_C = (1-β) * *W* and W_G = β * W
P: processing throughput (*W*/second)

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

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

Model the HW capabilities

- Workload: $W_G + W_C = W$
- Execution time: T_G and T_C
- P (processing throughput)
  - Measured as workload processed per second
  - P evaluates the hardware capability of a processor

GPU kernel execution time: $T_G = W_G / P_G$
CPU kernel execution time: $T_C = W_C / P_C$

Model the data-transfer

- O (GPU data-transfer size)
  - Measured in bytes
- Q (GPU data-transfer bandwidth)
  - Measured in bytes per second

Data-transfer time: TD=O/Q + (Latency)
Latency < 0.1 ms, negligible impact

Predict the partitioning …

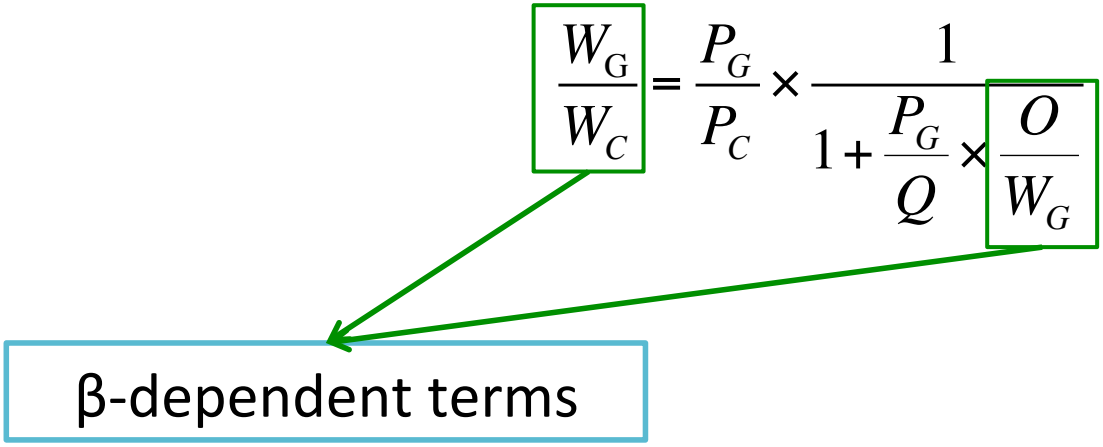
$$T_G = \frac{W_G}{P_G} \quad T_C = \frac{W_C}{P_C} \quad T_D = \frac{O}{Q}$$

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

… by solving this equation in β

Predict the partitioning

- Solve the equation



Expression

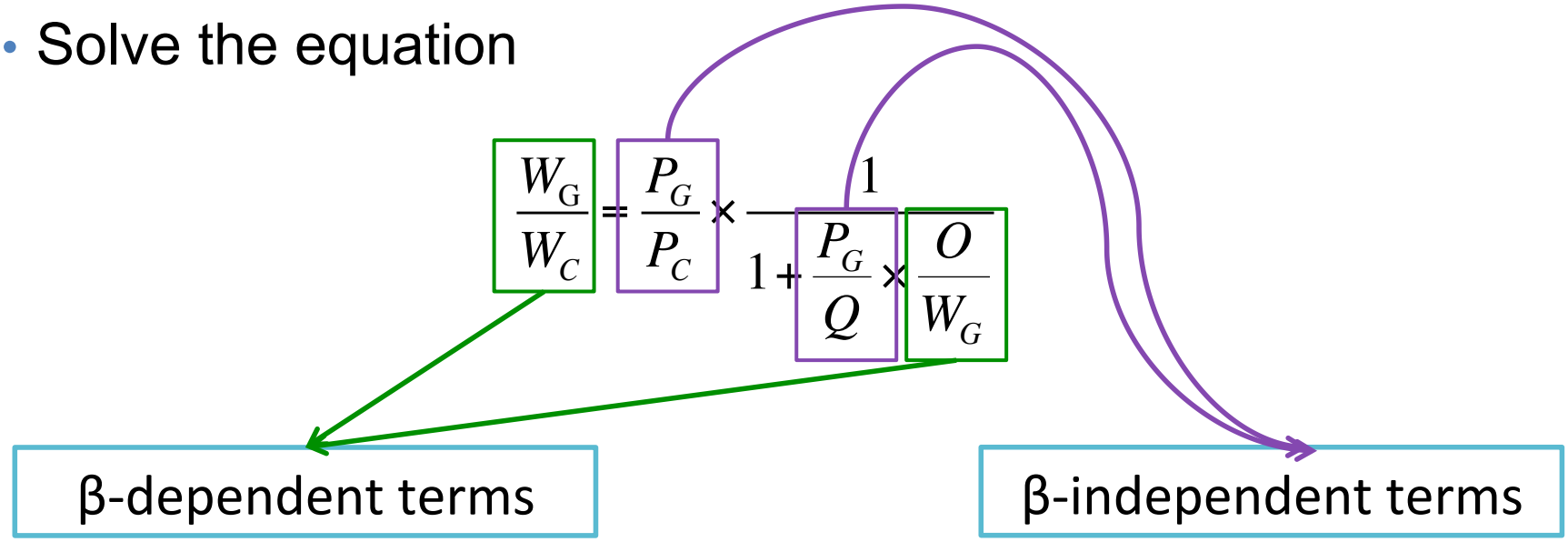
W_G=w x n x β

W_C=w x n x (1-β)

O=Full data transfer or
Full data transfer x β

Predict the partitioning

- Solve the equation



Expression

$W_G = w \times n \times \beta$

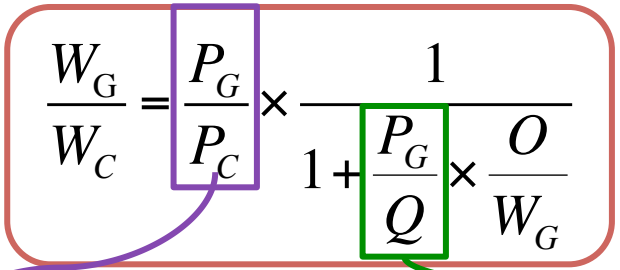
$W_C = w \times n \times (1-\beta)$

O=Full data transfer or
Full data transfer x β

Estimation

Modeling the partitioning

- Estimating the HW capability ratios by using profiling
  - The ratio of GPU throughput to CPU throughput
  - The ratio of GPU throughput to data transfer bandwidth



R_{GC}

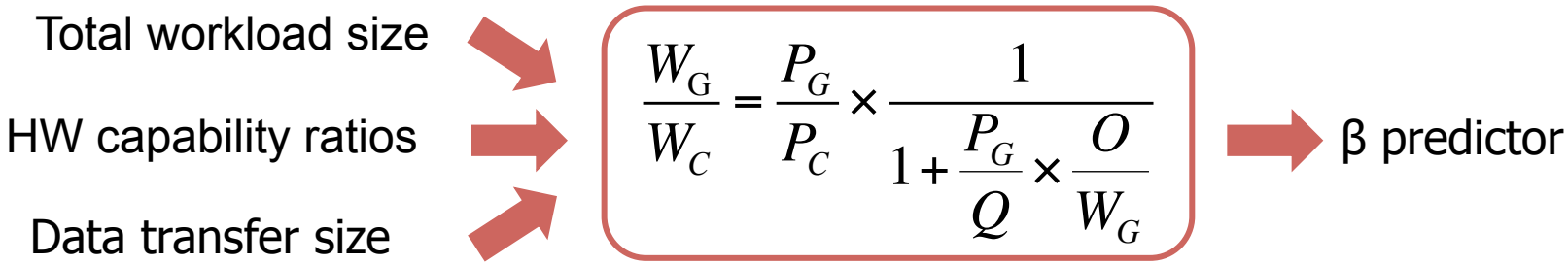
CPU kernel execution time vs.
GPU kernel execution time

R_{GD}

GPU data-transfer time vs.
GPU kernel execution time

Predicting the optimal partitioning

- Solving β from the equation



- There are three β predictors (by data transfer type)

$$\beta = \frac{R_{GC}}{1 + R_{GC}}$$

No data transfer

$$\beta = \frac{R_{GC}}{1 + \frac{v}{w} \times R_{GD} + R_{GC}}$$

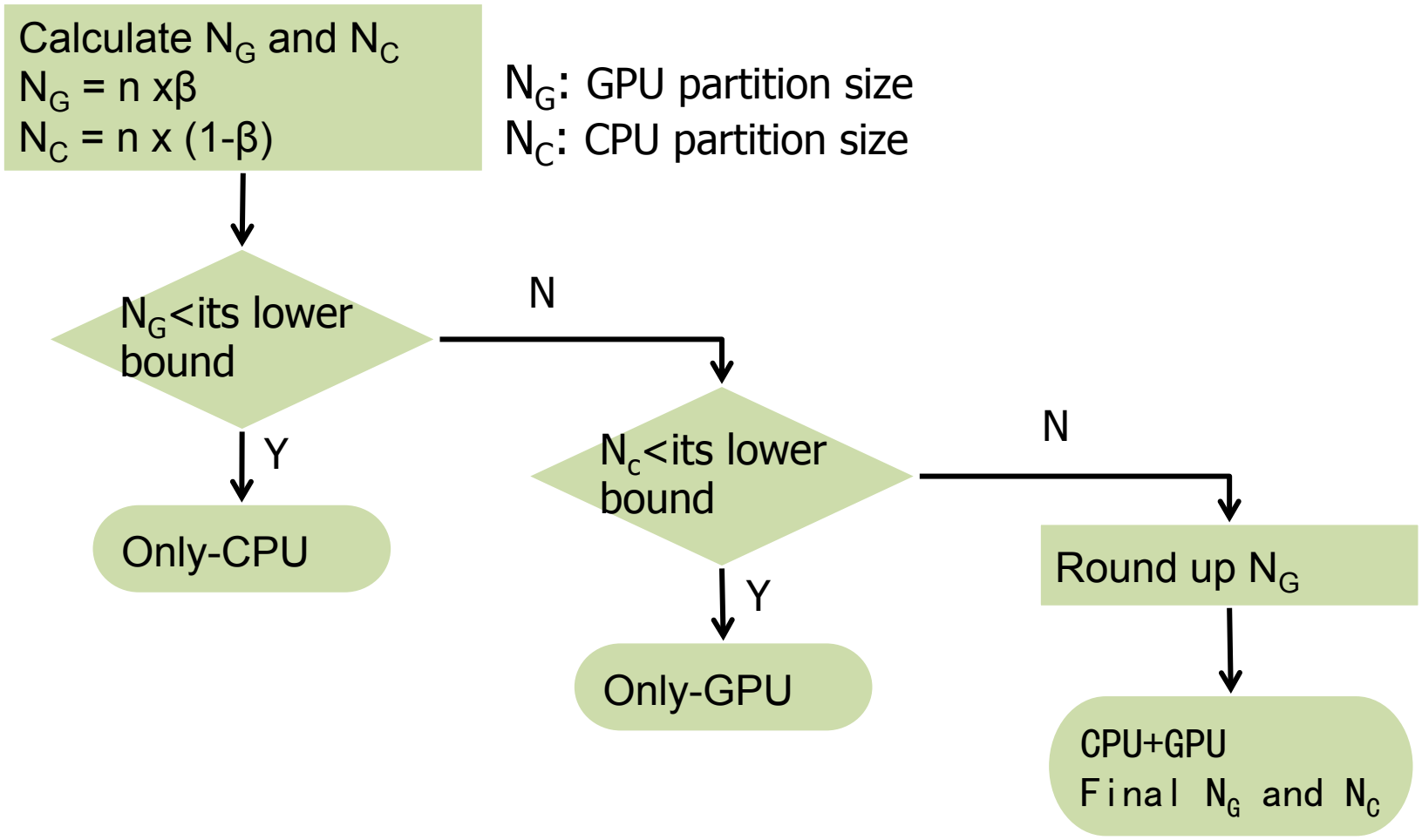
Partial data transfer

$$\beta = \frac{R_{GC} - \frac{v}{w} \times R_{GD}}{1 + R_{GC}}$$

Full data transfer

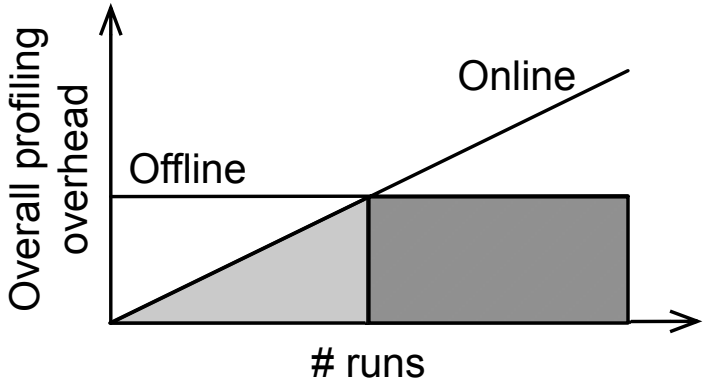
Making the decision in practice

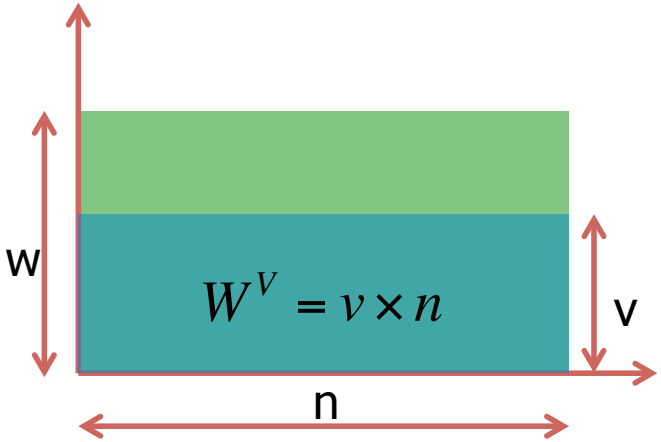
- From β to a practical HW configuration



Extensions

- Different profiling options
  - Online vs. Offline profiling
  - Partial vs. Full profiling





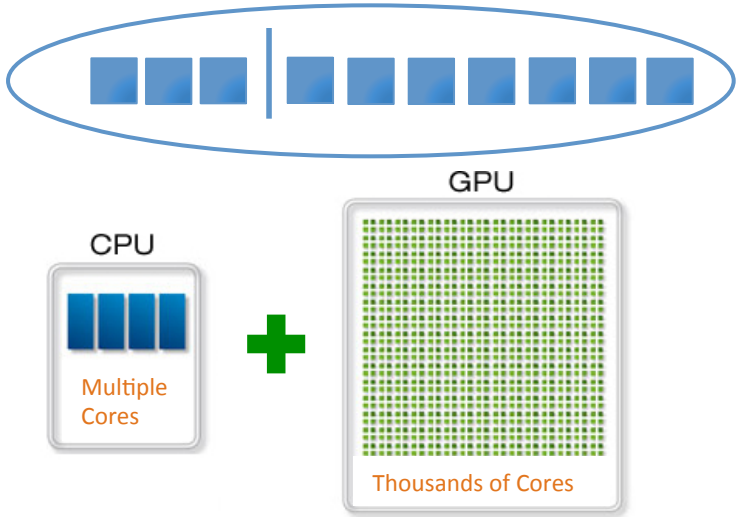
profile partial workload W^v

$$v_{\min} \le v \le w$$

- CPU+Multiple GPUs
  - Identical GPUs
  - Non-identical GPUs (may be suboptimal)

What if …

- … we have multiple kernels?
- … Can we still do static partitioning ?



Multi-kernel applications?

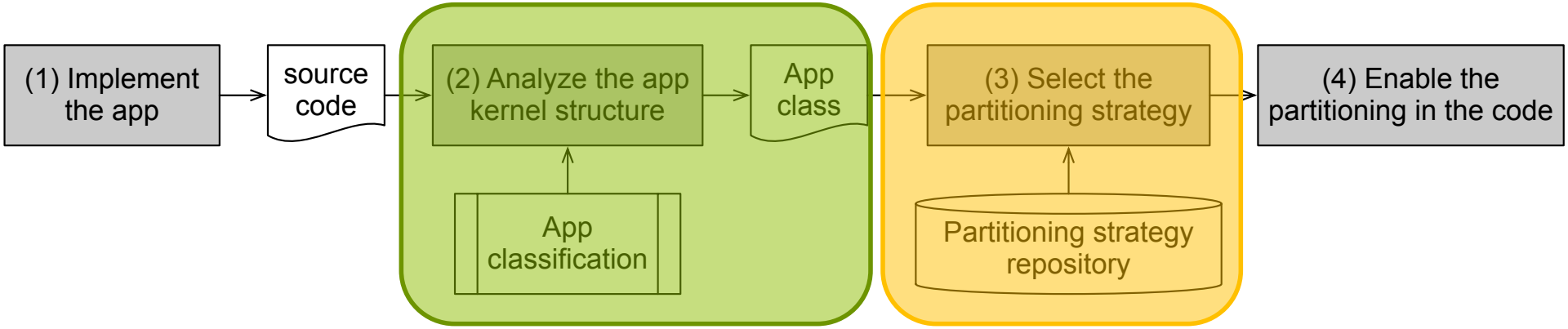
- Use dynamic partitioning: OmpSs, StarPU
  - Partition the kernels into chunks
  - Distribute chunks to *PUs
  - Keep data dependencies
  - (Often) lead to suboptimal performance
    - Scheduling policies and chunk size

Static partitioning: low applicability, high performance.
Dynamic partitioning: low performance, high applicability.

Can we get the best of both worlds?

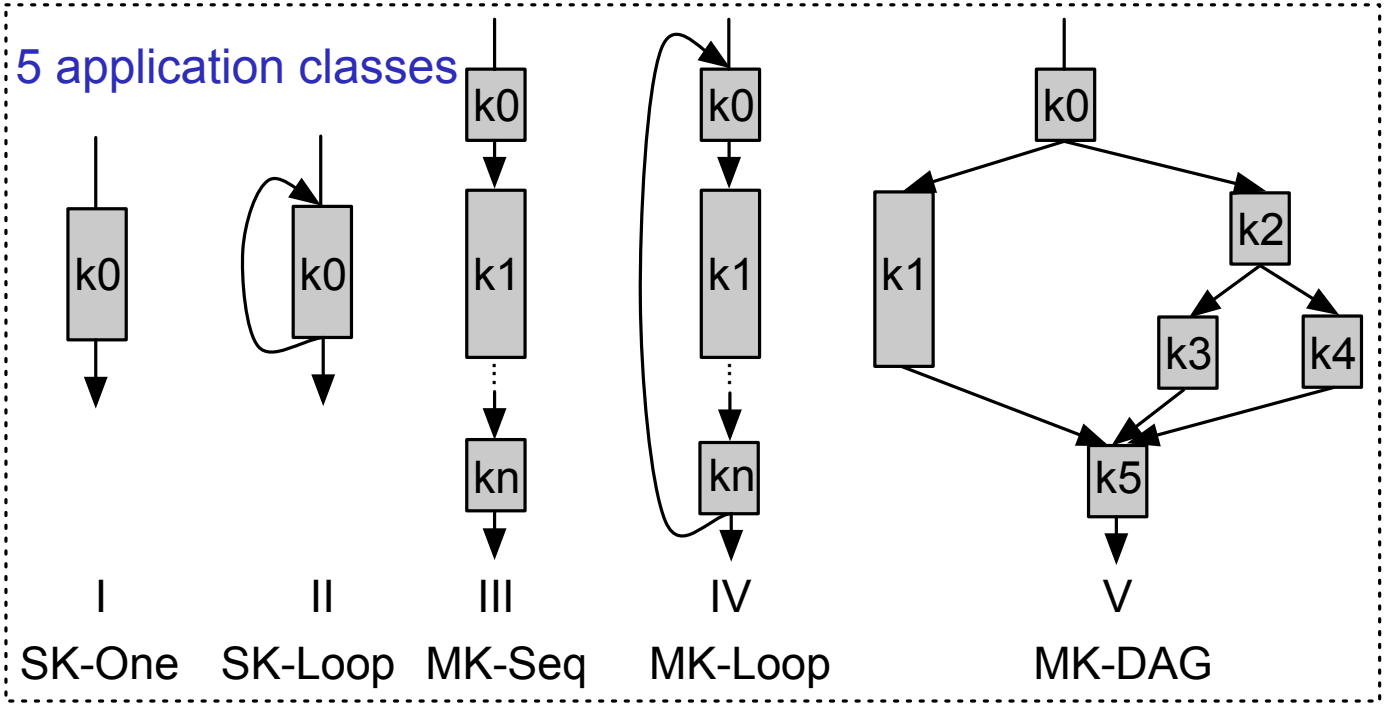
How to satisfy both requirements?

- We combine static and dynamic partitioning
  - We design an application analyzer that chooses the best performing partitioning strategy for any given application



Implement/get the source code → Analyze application's class → Select a partitioning strategy → Enable the partitioning

Application classification

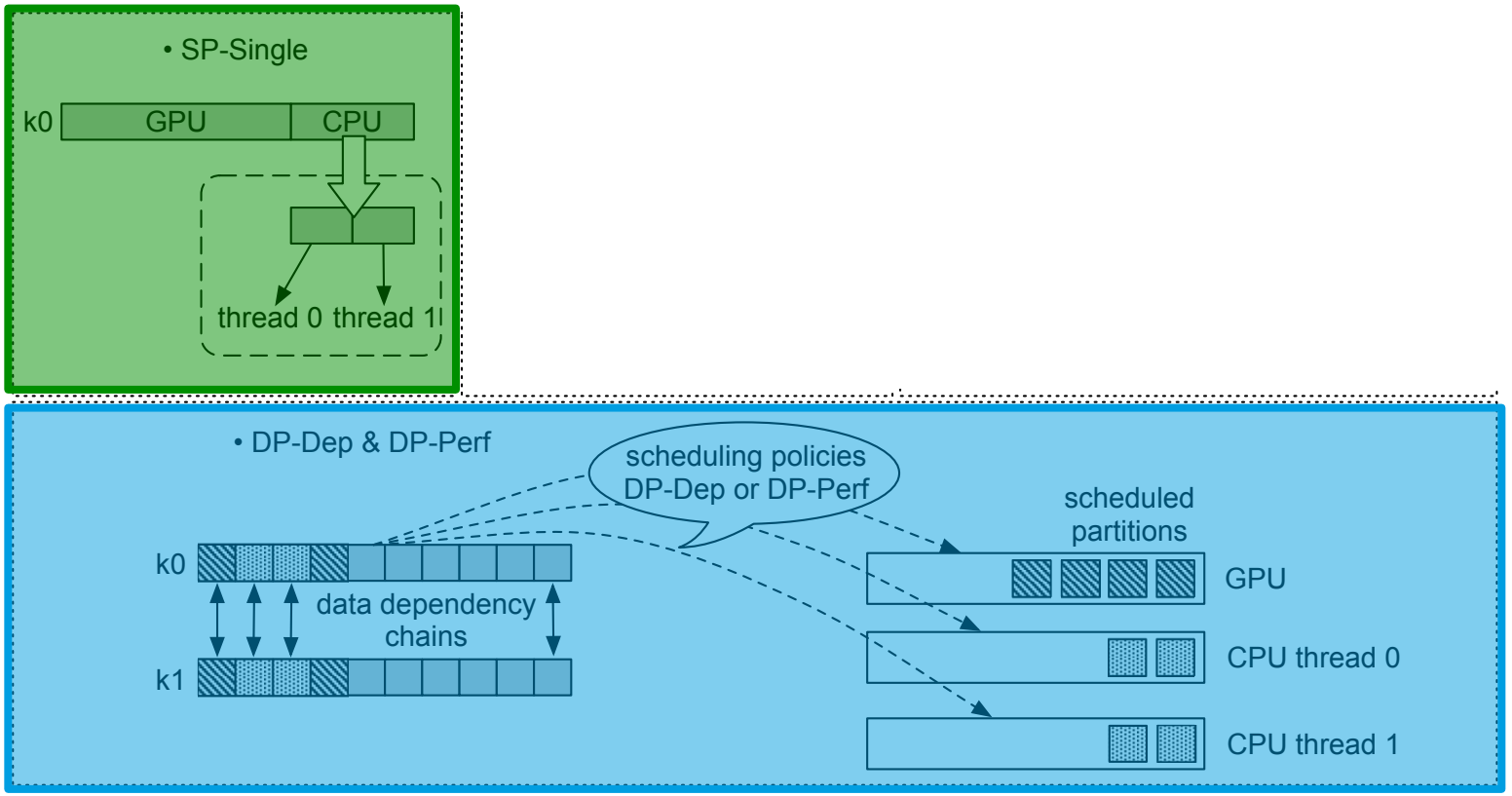


Implement/get the source code → Analyze application's class → Select a partitioning strategy → Enable the partitioning

Partitioning strategies

Static partitioning:
single-kernel applications

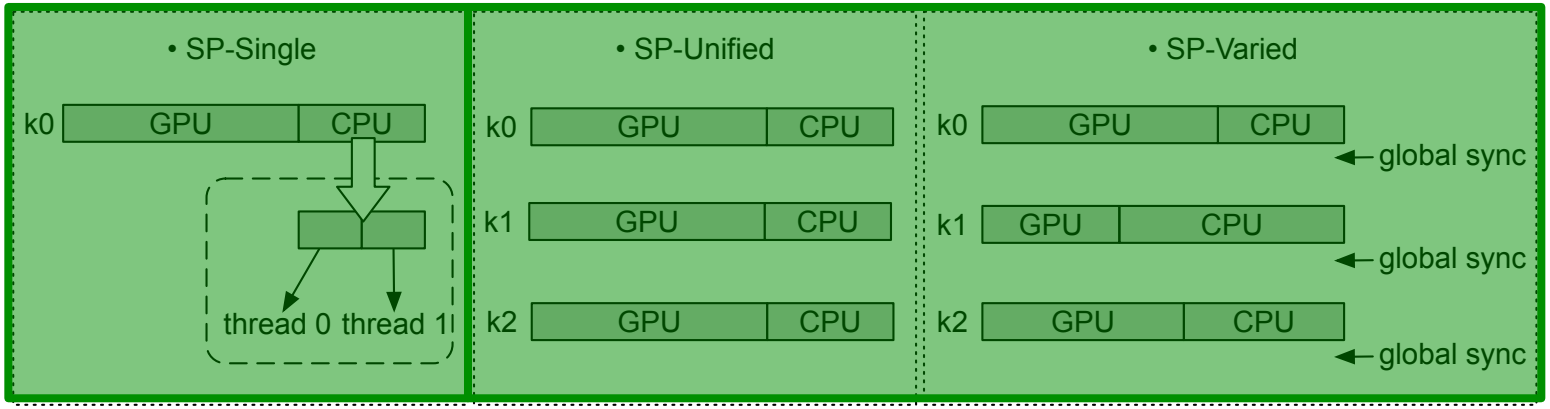
Dynamic partitioning:
multi-kernel applications

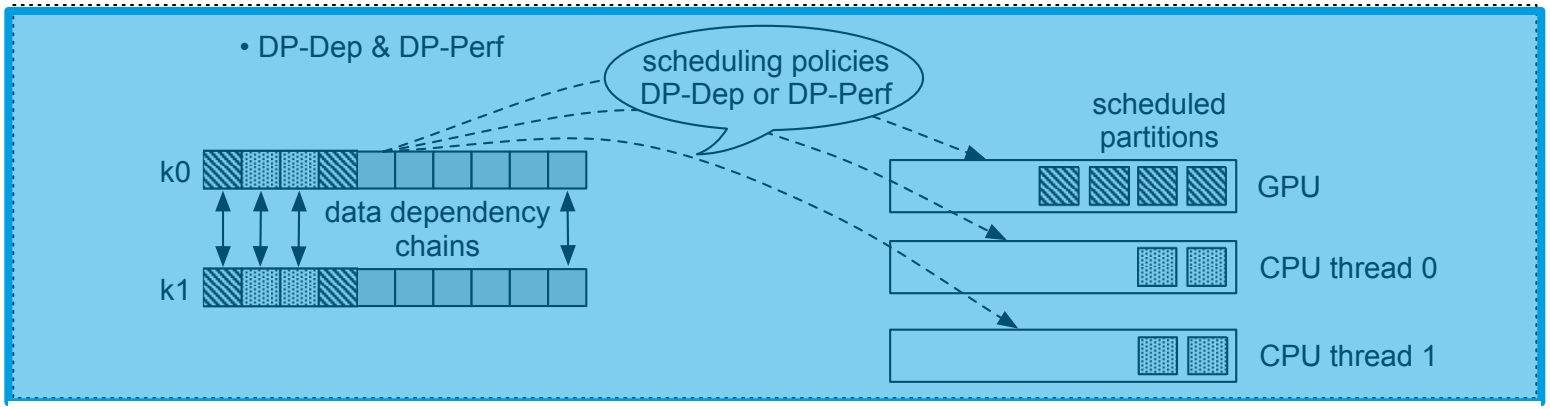


Implement/get the source code → Analyze application's class → Select a partitioning strategy → Enable the partitioning

Partitioning strategies

Static partitioning: in Glinda
single-kernel + multi-kernel applications





Dynamic partitioning: in OmpSs
multi-kernel applications (fall-back scenarios)

Implemen
the source code

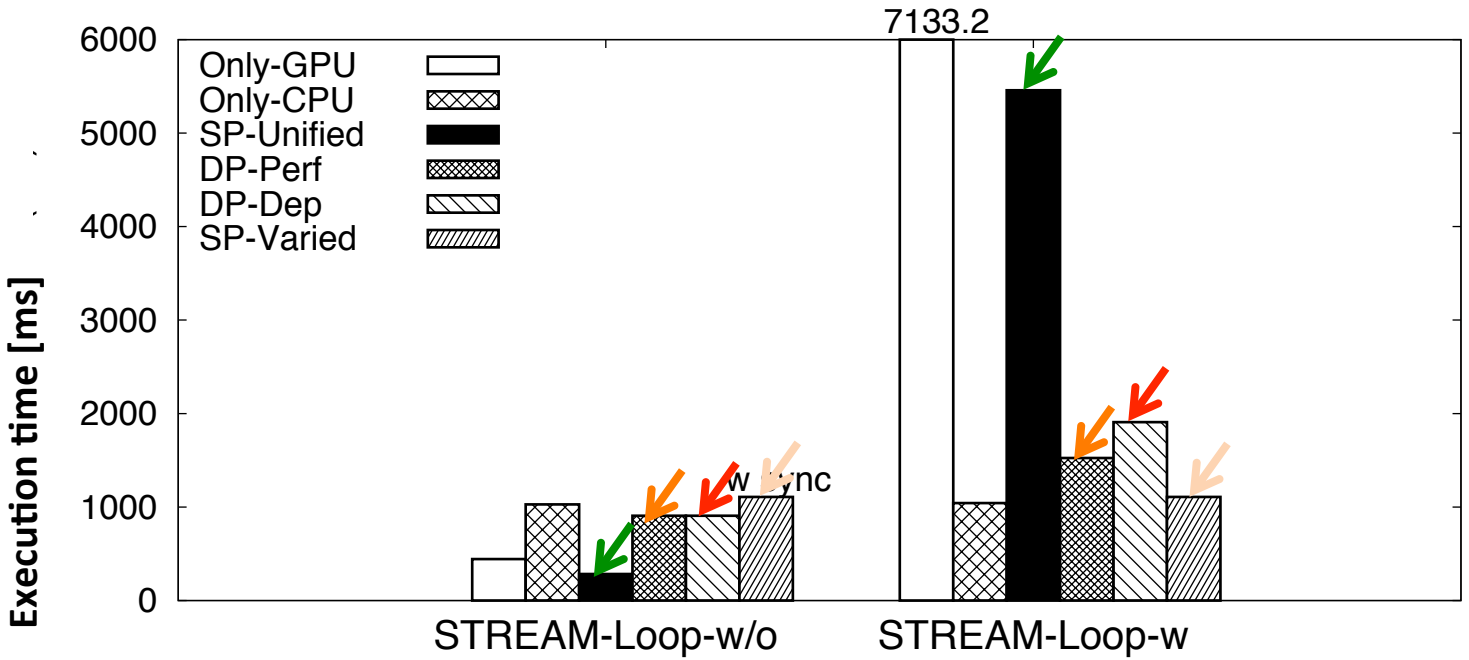
application's class

partitioning strategy

the partitioning

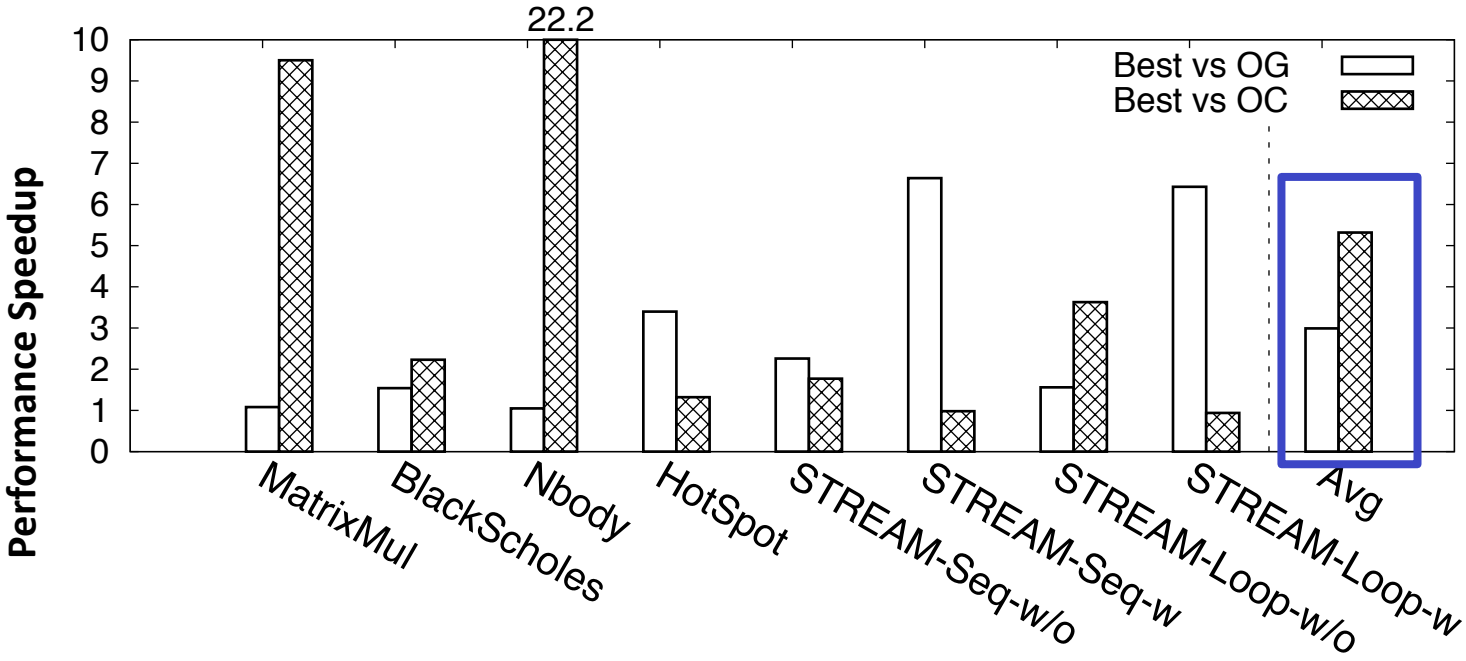
Results [4]

- MK-Loop (STREAM-Loop)
  - w/o sync: SP-Unified > DP-Perf >= DP-Dep > SP-Varied
  - with sync: SP-Varied > DP-Perf >= DP-Dep > SP-Unified



Results: performance gain

- Best partitioning strategy vs. Only-CPU or Only-GPU



Average:
3.0x (vs. Only-GPU)
5.3x (vs. Only-CPU)

Summary: Glinda

- Computes (close-to-)optimal partitioning
- Based on static partitioning
  - Single-kernel
  - Multi-kernel (with restrictions)

- No programming models attached
  - CUDA + OpenMP/TBB
  - OpenCL
  - … others => we propose HPL