# Workload Placement on Heterogeneous CPU-GPU Systems

**Marcos N. L. Carvalho**[1,2,3], **Alkis Simitsis**[2], **Anna Queralt**[1,4], **Oscar Romero**[1]

[1] Universitat Politècnica de Catalunya, Spain
[2] Athena Research Center, Greece
[3] National and Kapodistrian University of Athens, Greece
[4] Barcelona Supercomputing Center, Spain

VLDB 2024, August 29th – Tutorial

# CPU, GPU, and CPU-GPU architectures

- **CPU architecture**
  - **Pros**: few powerful cores suitable for latency-bound computations, large memory size
  - **Cons**: limited processing throughput, limited memory bandwidth

| | cores | memory size | memory bandwidth | processing throughput |
|---|---|---|---|---|
| **CPU** | - | ++ | + | - |
| **GPU** | + | -- | ++ | ++ |
| **CPU-GPU** | ++ | ++ | ? | ? |

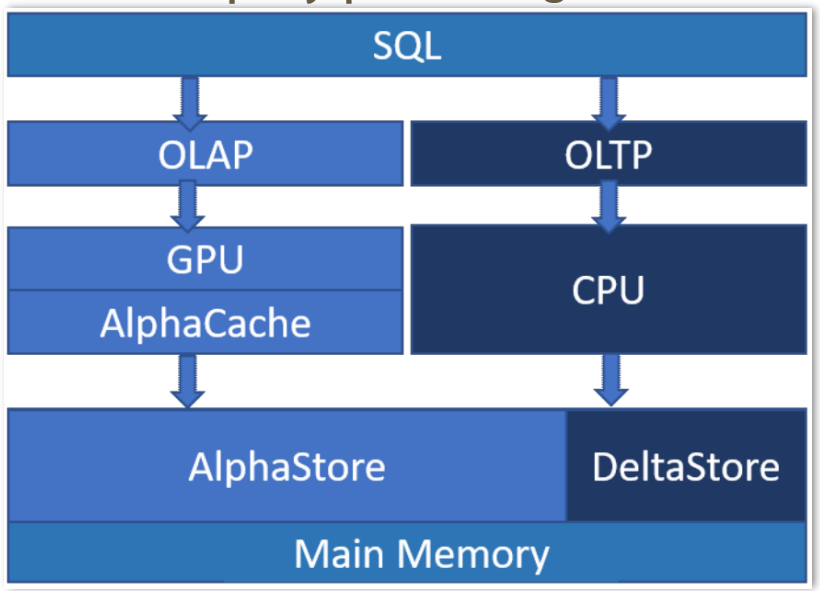
- **GPU architecture**
  - **Pros**: many simpler cores suitable for throughput-oriented computations, high processing throughput, high memory bandwidth
  - **Cons**: limited GPU memory size, data transfer bottleneck

- **Heterogeneous CPU-GPU architecture**
  - **Pros**: CPU-GPU co-processing, benefits both CPU and GPU
  - **Cons**: suboptimal workload placement decisions

# Example applications

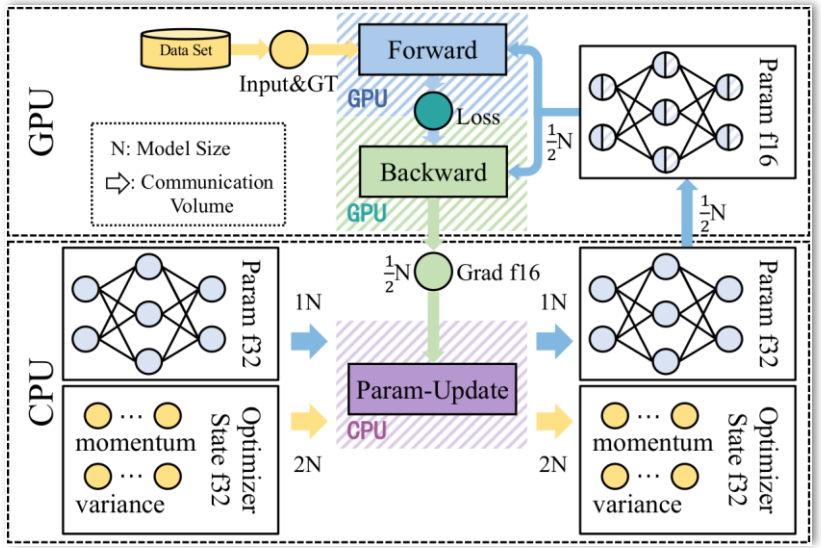
CPU-GPU architectures are employed in many domains and workloads*

## query processing



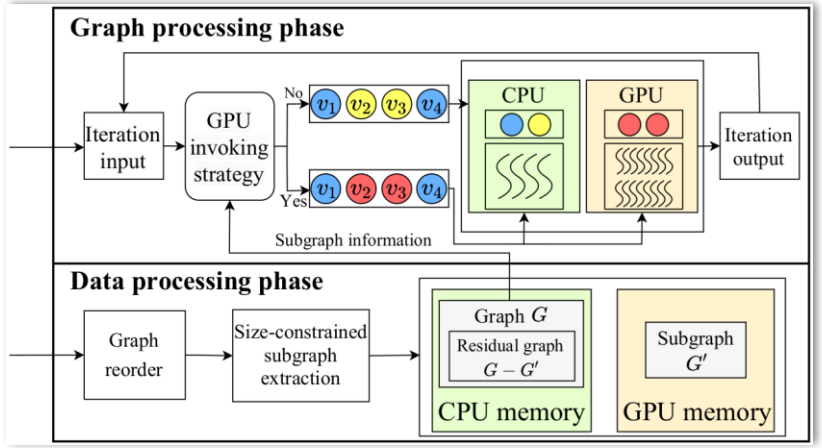

*RateupDB [PVLDB'21]*

## ML training




*CoTrain [ICPP'23]*

## graph processing

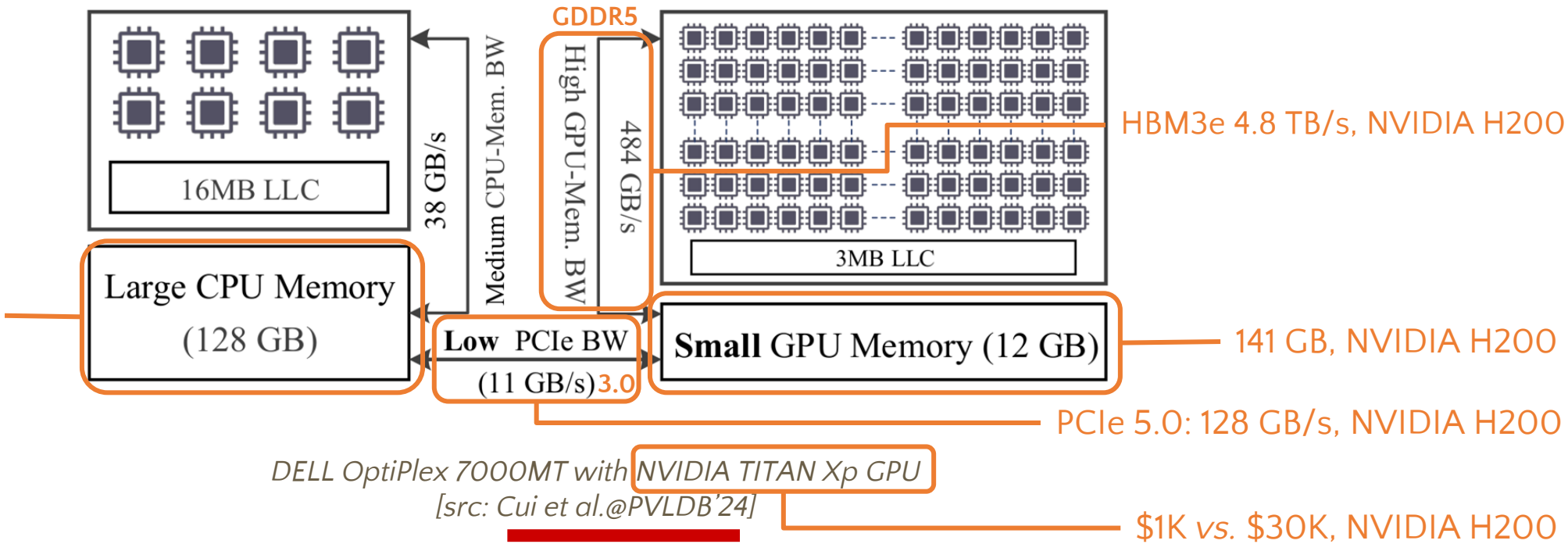



*CGgraph [PVLDB'24]*

*a variety of data processing functions: database queries, data flows/pipelines, or general apps and programs

# Heterogeneous CPU-GPU architectures

Commodity and modern setups:



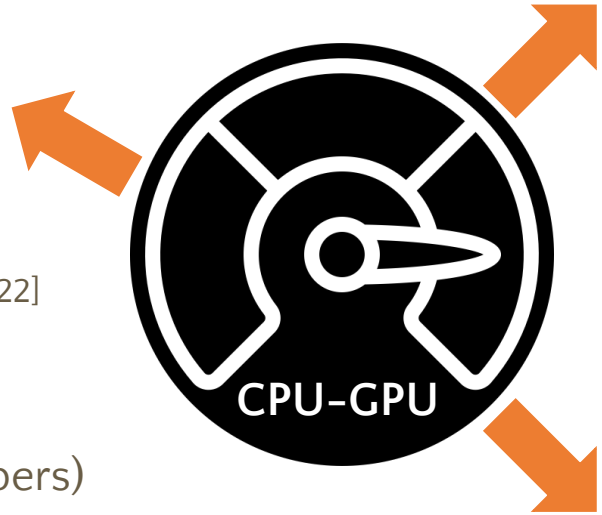
# Optimization challenges in heterogeneous CPU-GPU systems

**Challenge 2: Estimate performance metrics over different hardware architectures**

- Blend of techniques based on heuristics, cost models, and learn models

**Challenge 1: Find the optimal workload placement**

- Workload placement - where?
  - Scheduling - when? [not our focus today]
    e.g. [S. Mittal and J.S. Vetter@CSUR'15; Rosenfeld et al.@CSUR'22]
- Processor selection *vs.* data locality trade-off
  - Large search space of execution parameters
  - Lack of concrete guidelines (intuition of developers)
- Consequences of a bad placement strategy
  - Load imbalance
  - Waste of resources
  - Amplification of the data transfer bottleneck




Knobs:
- placement
- performance metrics
- multi-device ecosystem

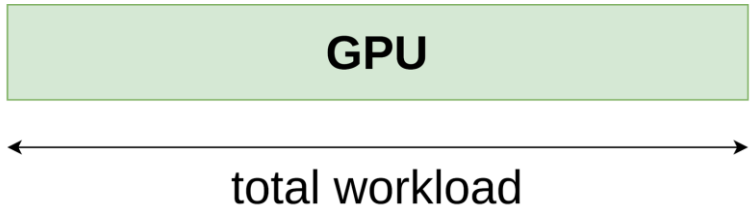
**Challenge 3: Manage a multi-device ecosystem**

- CPU code
- GPU code: kernel templates, compilers, and raw kernels
- GPU usage
- GPU integration

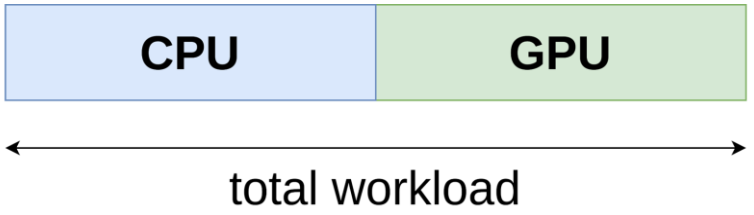
# The CPU-GPU Landscape — usage

**GPU usage and placement decisions** [Yogatama et al.@PVLDB'22]

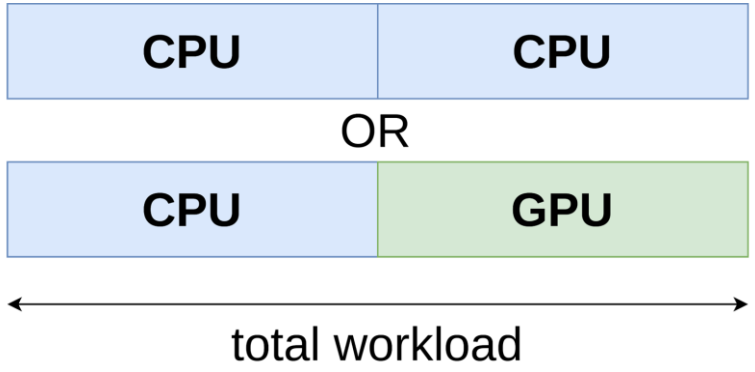
- **Primary processor**
  - All (or a significant portion) workload on GPU
  - **Pros**: low amount of CPU-GPU communication
  - **Cons**: limited GPU memory size, not fit for non-parallelizable workloads
  - **Placement**: static




- **Accelerator**
  - Specific portions of the workload only on GPU
  - **Pros**: robust to limited GPU memory size, intra-device parallelism
  - **Cons**: excessive CPU-GPU communication
  - **Placement**: static



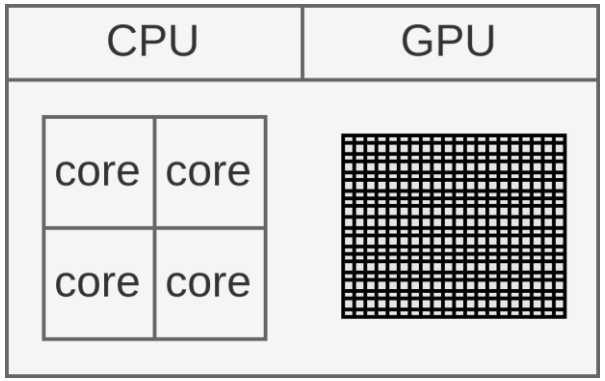

- **Heterogeneous CPU-GPU** [our focus today]
  - Specific portions of the workload on CPU, GPU or both
  - **Pros**: robust to limited GPU memory size, inter and intra-device parallelism
  - **Cons**: CPU-GPU communication
  - **Placement**: dynamic



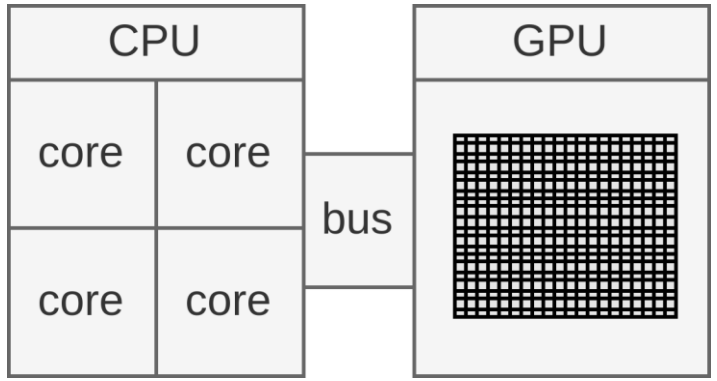
# The CPU-GPU Landscape — integration

## GPU integration and the data transfer bottleneck

- **Integrated GPU**
  - Characteristics
    - CPU and GPU cores on same die
    - Shared system main memory
  - **Pros**: no CPU-GPU communication, better power/performance ratio
  - **Cons**: limited memory bandwidth and processing throughput, resource contention




- **Dedicated GPU** [our focus today]
  - Characteristics
    - CPU and GPU cores on different dies
    - GPU memory decoupled from system main memory
  - **Pros**: high memory bandwidth and processing throughput
  - **Cons**: limited GPU memory size, worse power/performance ratio, limited interconnect bandwidth, **CPU-GPU communication**



# The CPU-GPU Landscape — data transfer/locality

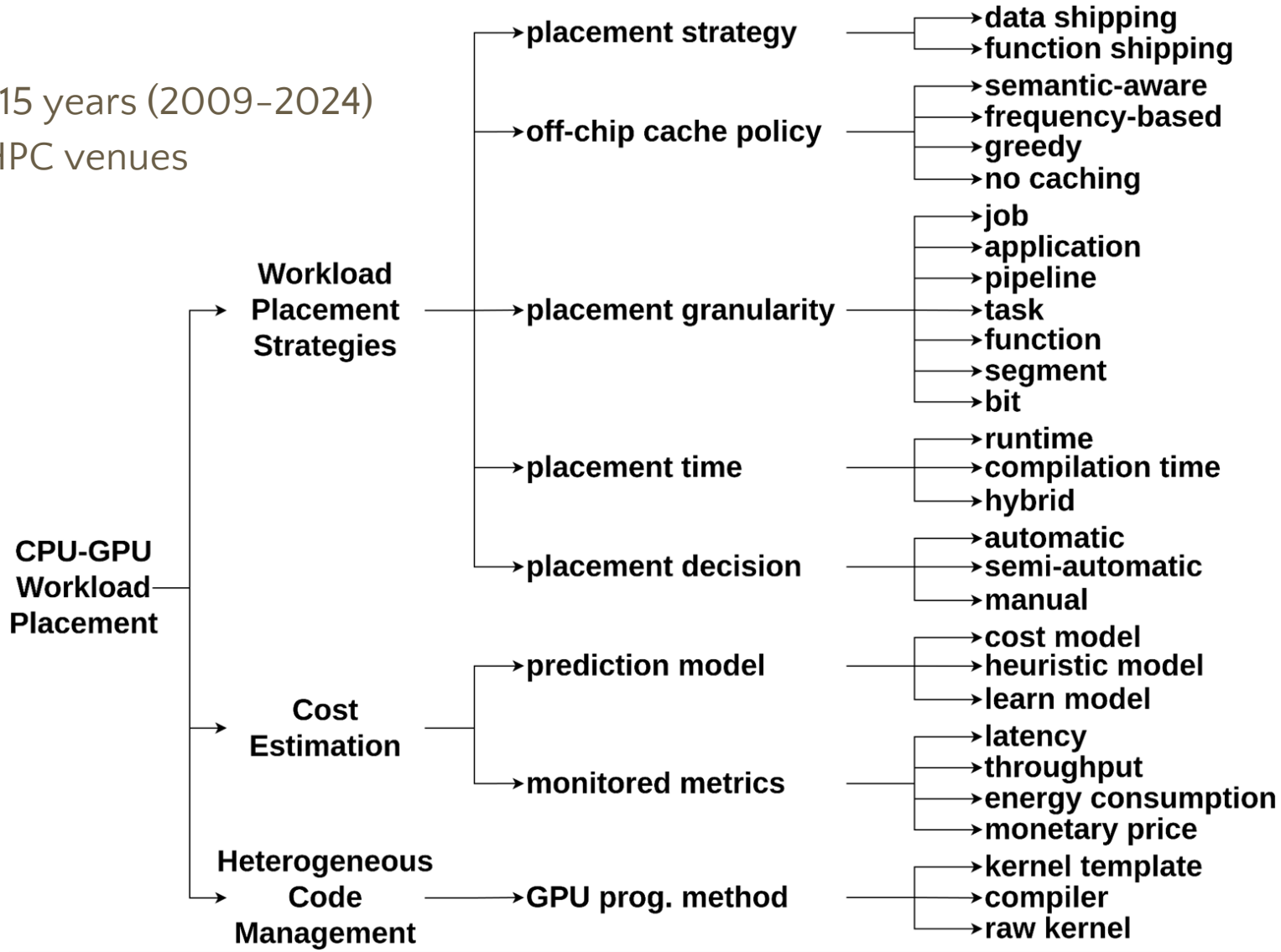
- **Data transfer techniques for dGPUs** (see also an excellent survey on the topic [Rosenfeld et al.@CSUR'22])
  - **Hardware-based techniques**
    - Fast bus interconnect
    - On-chip caching
  - **Software-based techniques**
    - GPU Storage RDMA
    - Overlapping
    - Compression
    - Single-pass algorithm
    - Heterogeneous execution
    - Data locality: avoid data movement by placing workloads on processors where data is already located
      - Common technique: **off-chip caching** (keep useful data in memory)
        - System main memory for CPU and GPU memory for GPU
- **Workload placement approaches**
  - **Ignore data locality:** prioritize the faster processor at the cost of expensive movement
  - **Consider data locality:** minimize data movement with off-chip cache (not always use the faster processor)

# A taxonomy for CPU-GPU workload placement

# A Taxonomy for CPU-GPU Workload Placement (abridged)

## Methodology

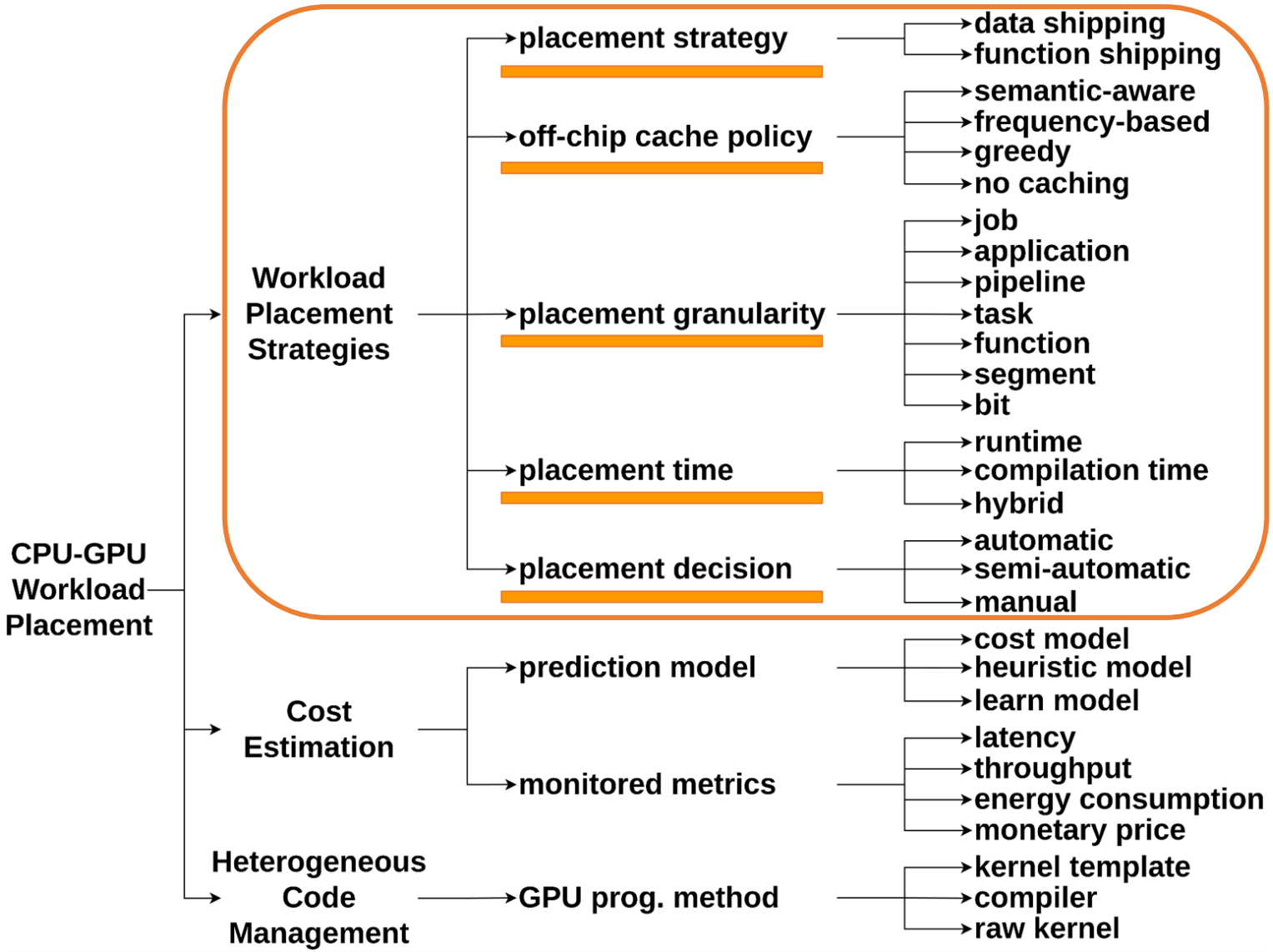
- 77 publications in 15 years (2009-2024)
- Top-tier DB and HPC venues

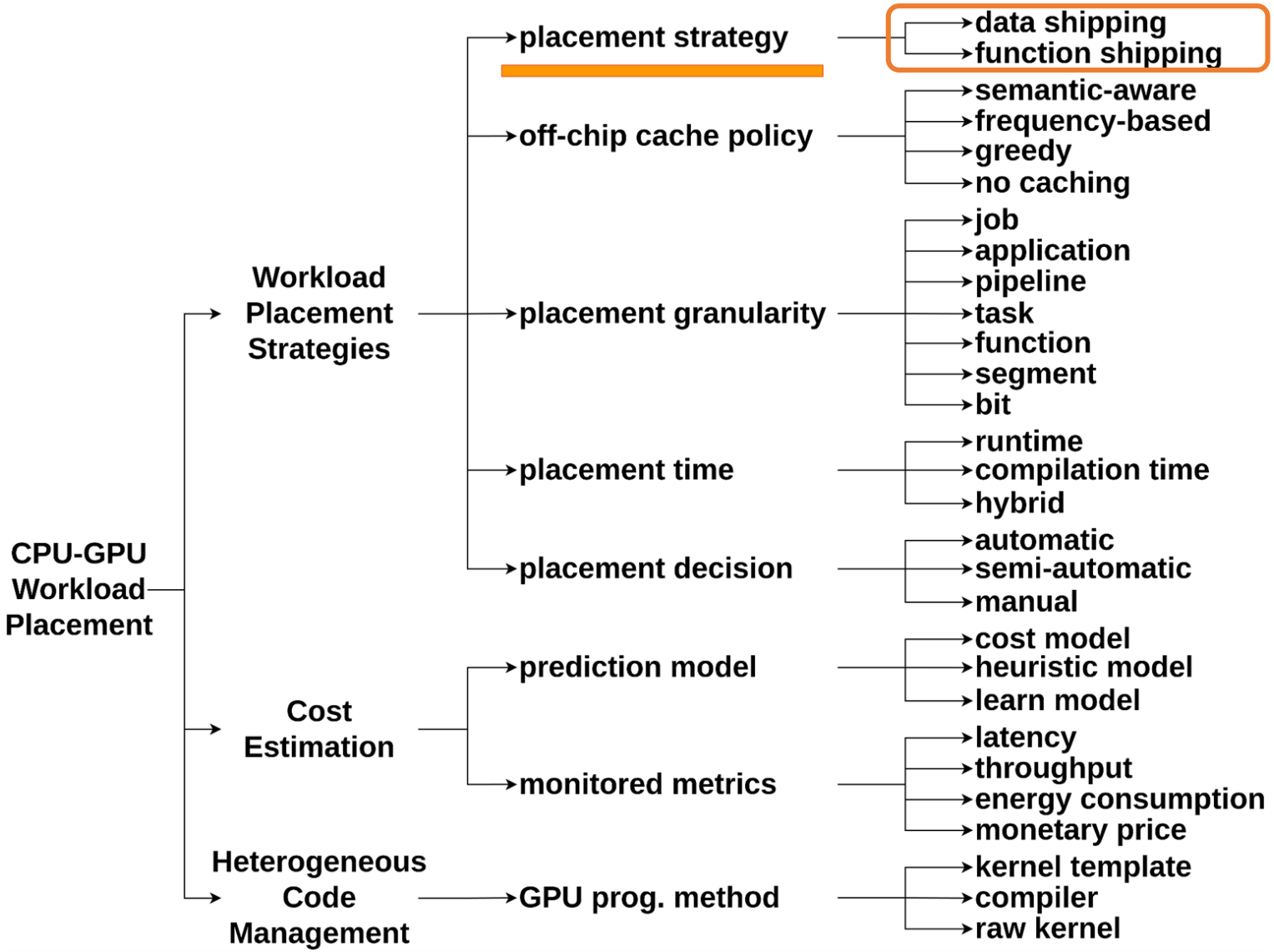


# A Taxonomy for CPU-GPU Work

| Dimension Group | Workload Placement Strategies | | | | | | | | | | | | | | Cost Estimation | | | | | | | | Heterog. Code Management | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dimensions | Plmt. Strategy | | Off-chip Cache Policy | | | | Plmt. Granularity | | | | | | | Plmt. Time | | | Plmt. Decision | | | Plmt. Pred. Model | | | Monitored Metrics | | | | GPU Prog. Method | | |
| Features | Data shipping | Function shipping | Semantic-aware | Frequency-based | Greedy | No caching | Job | Application | Pipeline | Task | Function | Segment | Bit | Runtime | Compilation time | Hybrid | Automatic | Semi-automatic | Manual | Cost model | Heuristic model | Learn model | Latency | Throughput | Energy consumption | Price | Kernel template | Compiler | Raw kernel |
| #papers | 68 | 9 | 5 | 9 | 13 | 50 | 3 | 9 | 7 | 49 | 2 | 5 | 2 | 26 | | | | | | | | | | | | | | | |
| Karnagel et al.@EDBTW'15 | X | | | | | X | | | | X | | | | | | | | | | | | | | | | | | | |
| Robust CoGaDB | | X | | X | | | | | | X | | | | X | | | | | | | | | | | | | | | |
| HetCache | | X | X | | | | | | X | | | | | X | | | | | | | | | | | | | | | |
| RateupDB | X | | | X | | | | X | | | | | | | | | | | | | | | | | | | | | |
| Parla | X | | | | X | | | | | X | | | | X | | | | | | | | | | | | | | | |
| Wen and O'Boyle@GPGPU-PPoPP'17 | X | | | | | X | | | | X | | | | | | | | | | | | | | | | | | | |
| Ravi et al.@CCGRID'12 | X | | | | | X | X | | | | | | | | | | | | | | | | | | | | | | |
| Compressed Crystal | X | | | | | X | | X | | | | | | | | | | | | | | | | | | | | | |
| HetExchange | X | | | | | X | | | X | | | | | | | | | | | | | | | | | | | | |
| Carvalho et al.@EDBT'24 | X | | | | | X | | | | X | | | | | | | | | | | | | | | | | | | |
| HERO | | X | | X | | | | | | | X | | | | | | | | | | | | | | | | | | |
| Mordred | | X | X | | | | | | | | | X | | | | | | | | | | | | | | | | | |
| Pirk@VLDBW'12 | X | | | | | X | | | | | | | X | | | | | | | | | | | | | | | | |
| Li et al.@J. Syst. Arch'2021 | X | | | | | X | | | | X | | | | X | | | | | | | | | | | | | | | |
| MCL Schedulers | | X | | | X | | | | | X | | | | X | | | | | | | | | | | | | | | |
| CGgraph | | X | X | | | | | | | X | | | | | | | | | | | | | | | | | | | |
| DBD | X | | | | | X | | | | X | | | | | | | | | | | | | | | | | | | |
| CoTrain | X | | | | X | | | | | X | | | | | | | | | | | | | | | | | | | |
| TensorFlow | X | | | | X | | | | | X | | | | | | | | | | | | | | | | | | | |
| Gowanlock et al.@DaMoN'19 | X | | | | | X | | | | X | | | | X | | | | | | | | | | | | | | | |
| GaccO | | X | X | | | | | X | | | | | | | | | | | | | | | | | | | | | |
| READYS | X | | | | | X | | | | X | | | | X | | | | | | | | | | | | | | | |
| Placeto | X | | | | | X | | | X | | | | | | | | | | | | | | | | | | | | |
| Lutz et al.@SIGMOD'20 | X | | | | X | | | | | X | | | | | | | | | | | | | | | | | | | |
| Sigmoid | X | | | | | X | | | | X | | | | X | | | | | | | | | | | | | | | |
| Crystal | X | | | | | X | | X | | | | | | | | | | | | | | | | | | | | | |
| Lee and Park@ICDEW'21 | X | | | | | X | | | | X | | | | | | | | | | | | | | | | | | | |
| Xekalaki et al.@VLDB'22 | X | | | | | X | | | | X | | | | | | | | | | | | | | | | | | | |
| GFlink | X | | | X | | | | | | X | | | | | | | | | | | | | | | | | | | |

# Workload placement strategies


CPU-GPU Workload Placement
Workload Placement Strategies
placement strategy
data shipping
function shipping
off-chip cache policy
semantic-aware
frequency-based
greedy
no caching
placement granularity
job
application
pipeline
task
function
segment
bit
placement time
runtime
compilation time
hybrid
placement decision
automatic
semi-automatic
manual
Cost Estimation
prediction model
cost model
heuristic model
learn model
monitored metrics
latency
throughput
energy consumption
monetary price
Heterogeneous Code Management
GPU prog. method
kernel template
compiler
raw kernel

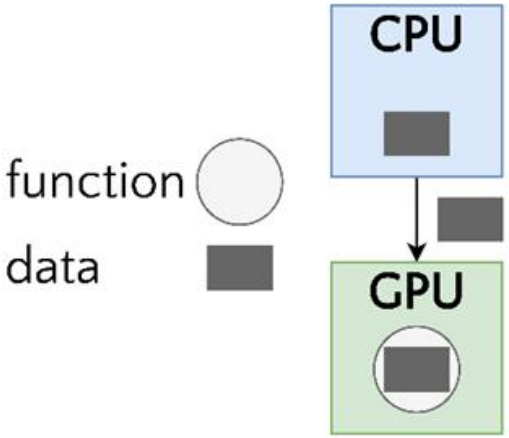

CPU-GPU Workload Placement
Workload Placement Strategies
placement strategy
data shipping
function shipping
off-chip cache policy
semantic-aware
frequency-based
greedy
no caching
placement granularity
job
application
pipeline
task
function
segment
bit
placement time
runtime
compilation time
hybrid
placement decision
automatic
semi-automatic
manual
Cost Estimation
prediction model
cost model
heuristic model
learn model
monitored metrics
latency
throughput
energy consumption
monetary price
Heterogeneous Code Management
GPU prog. method
kernel template
compiler
raw kernel

# Placement Strategy: overview

- Defines the placement order of a function and its input data
- Data shipping
  - Places functions first, then ships data (moves data to where functions are located)
  - Functions are placed on processors according to its relative performance
  - **Pros**: Prioritize the most fit processor
  - **Cons**: Data locality is not the priority, increased amount of data transfers
- Function shipping
  - Places data first, then ships functions (moves functions primarily to where data is located)
  - Data is placed and kept on the processor's off-chip memory considering a cache policy
  - **Pros**: Prioritize data locality, reduced amount of data transfers
  - **Cons**: Not always use the most fit processor

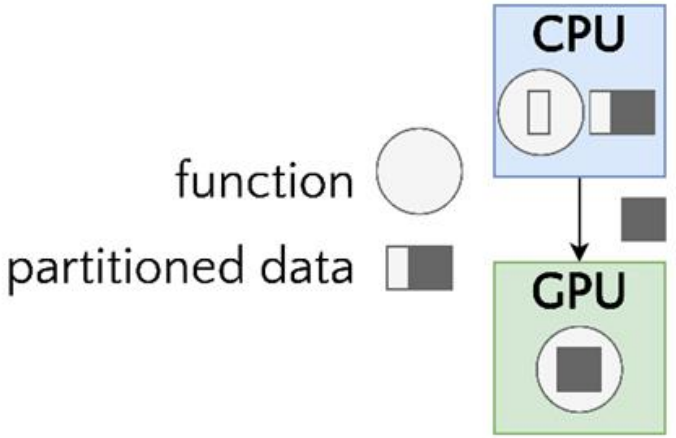
# Placement Strategy: data shipping (overview)

### Ship required input data from CPU to GPU




**Pros**: intuitive placement decision and implementation
**Cons**: load imbalance, large amount of data per data transfer, risk of GPU out-of-memory
**e.g.** Caldera, [Karnagel@EDBTW'15], Ocelot+HyPE


### Data partition on independent CPU-GPU functions

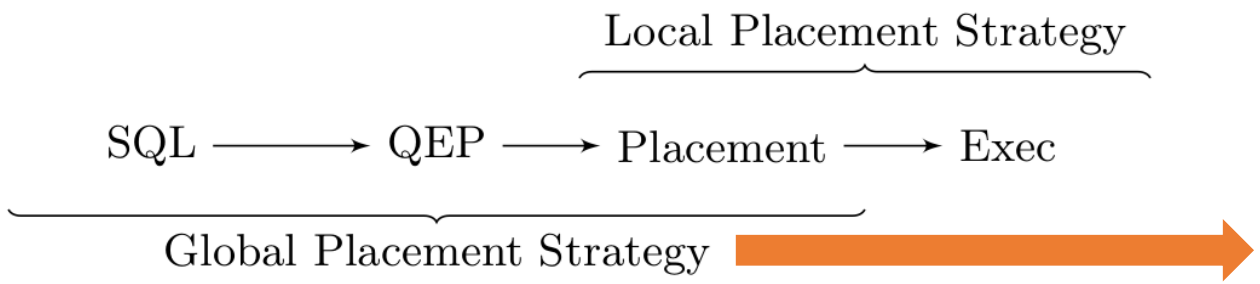



**Pros**: device parallelism, control of data granularity (load balance)
**Cons**: increased amount of data transfers, requires merging output data
**e.g.** [Kroviakov et al.@DAMON'24], CoTrain, HetExchange, [Gowanlock et al.@DaMoN'19]

# Placement Strategy: data shipping (example 1)

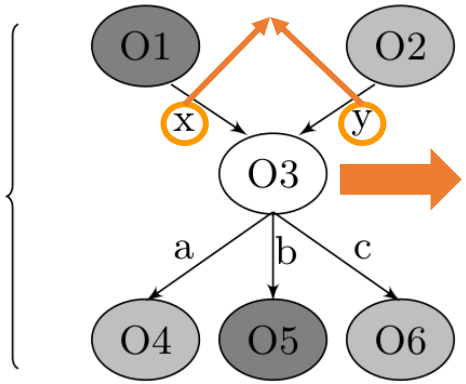
**e.g. ship data CPU-GPU (no partition)**: [Karnagel et al.@EDBTW'15]

- Key principles
  - Given a query execution plan, a placement decision is made for each operator
  - Once the operator is placed on a processor, move (if necessary) the required data to that processor
- Contributions
  - Greedy operator placement algorithm
    - Start with a pre-set placement decision for each operator (global placement)
      - More informed decision about data movements among operators
    - Then, iterate over each operator and evaluate if a local placement can reduce operator execution time

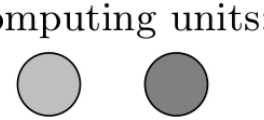



Information used for local cost estimation

data shipping
(table columns or intermediate results)




current operator

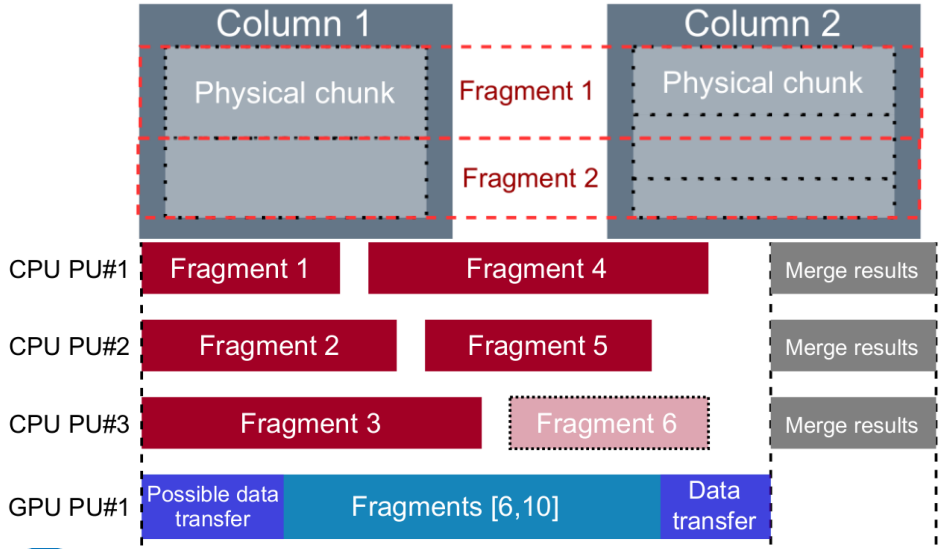


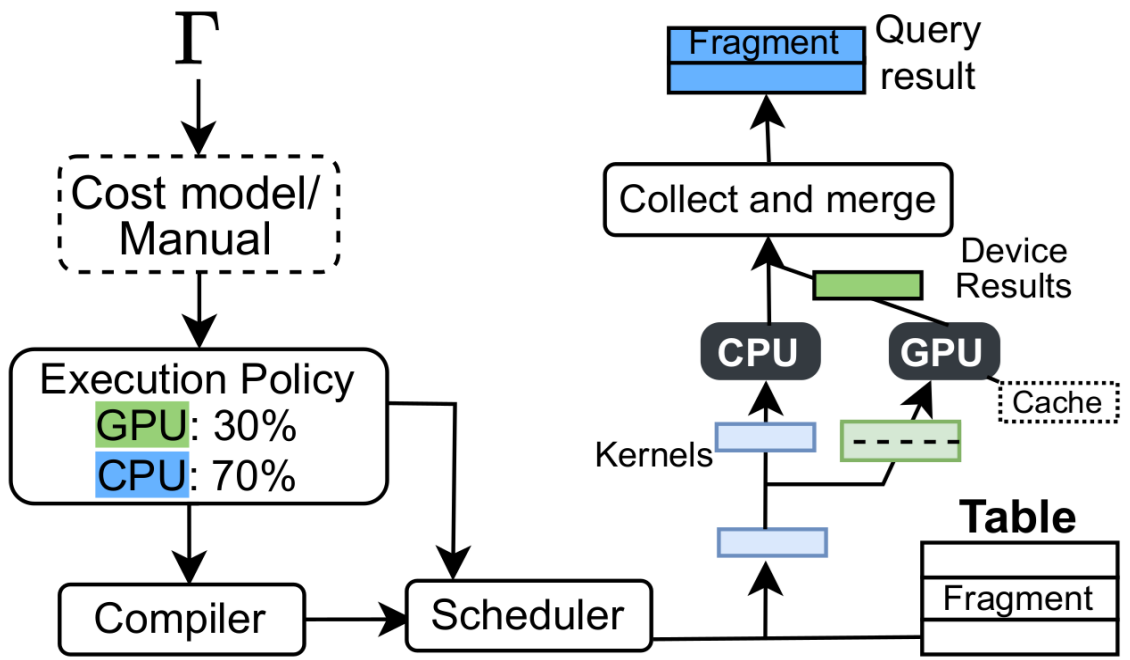

operator placement considering all dependencies in the QEP

# Placement Strategy: data shipping (example 2)

**e.g. data partition independent function**: [Kroviakov@DAMON'24]

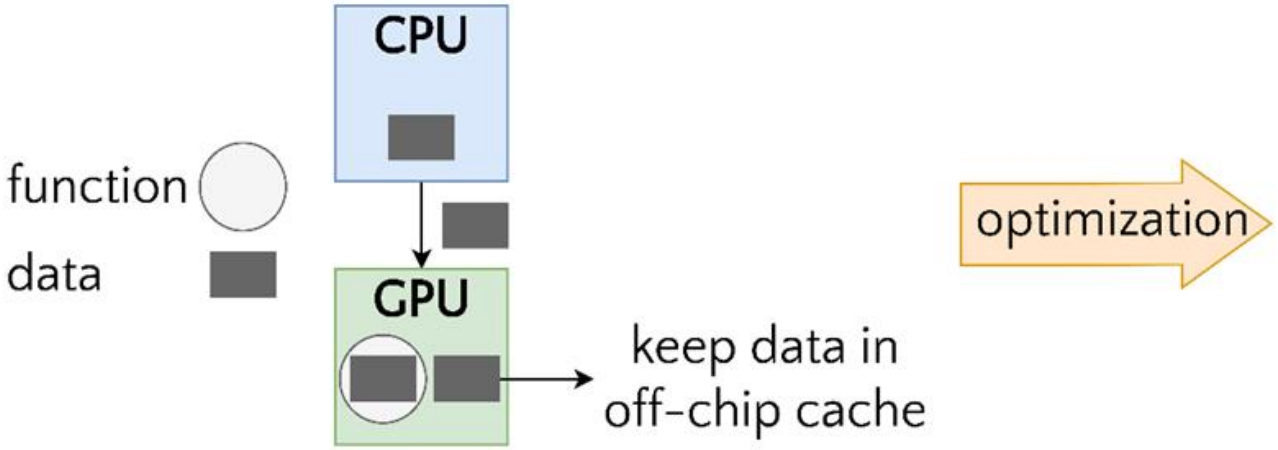
- Key principles
  - Supports cross-device horizontal morsel-driven parallelism in aggregation queries using fragments
  - Fragments are discrete horizontal partitions of a table with a fixed number of tuples
  - Selects CPU-GPU fragment proportions that minimize the execution time of aggregations
- Contributions
  - Supports either manual or automatic (cost model) fragment distribution between CPU and GPU
  - Data-driven model for data fragment distribution via numerical analysis over past data



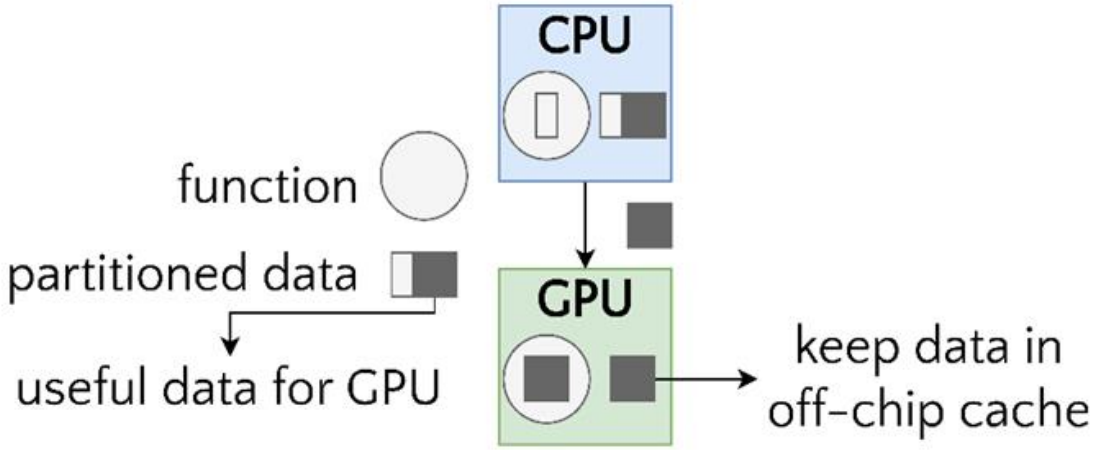
# Placement Strategy: function shipping (overview)

**Prioritize processors where entire data is local**




**Pros**: reduced amount of data transfers
**Cons**: high use of GPU memory, load imbalance
**e.g.** HetCache, GaccO, NeutronOrch, CGgraph, extended MCL, MCL schedulers, HERO, Robust CoGaDB

**Prioritize processors where fine-grained data is local**

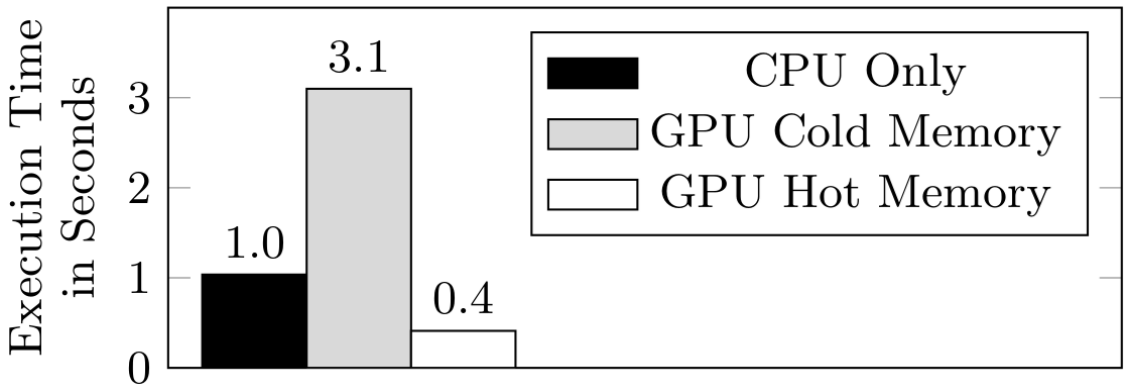


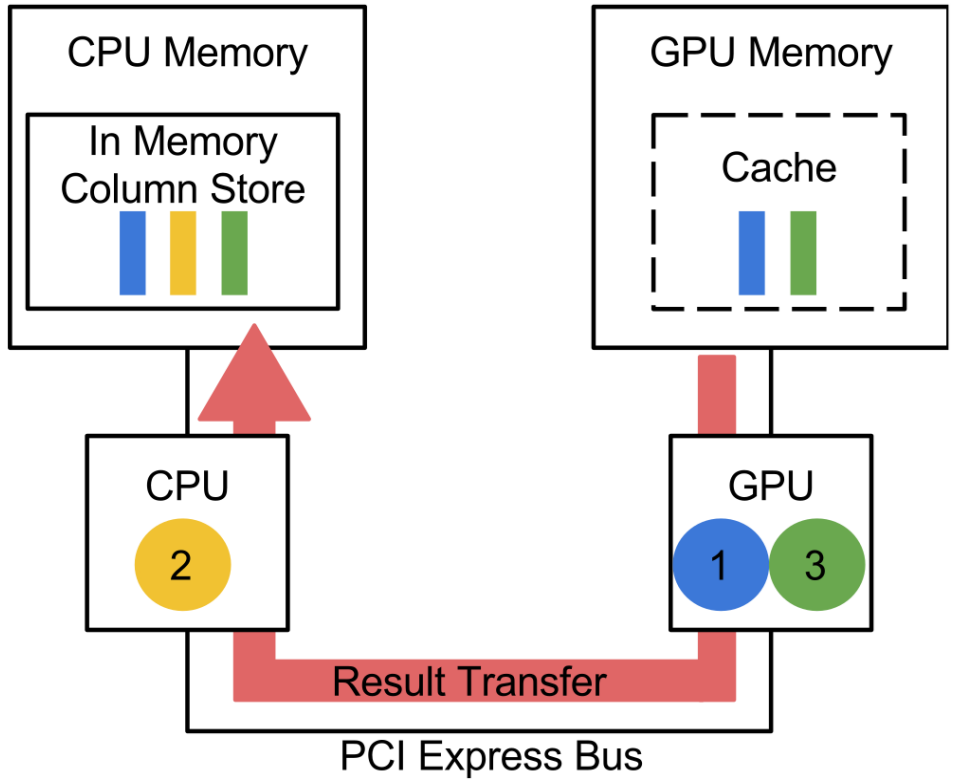

**Pros**: efficient use of GPU memory
**Cons**: increased amount of data transfers, complex implementation (e.g. intermediate materialization, merge outputs)
**e.g.** Mordred

# Placement Strategy: function shipping (example 1)

**e.g. prioritize processors where entire data is local**: Robust CoGaDB [Breß et al.@SIGMOD'16]

- Key principles
  - Shows that GPU performs worse than CPU because of slow PCIe data transfers for OLAP query processing
  - Places query operators on GPUs only if all data is in GPU memory. Otherwise, places operators on CPUs
    - Uses frequency-based cache policy to manage GPU cache
- Contributions
  - *Data-driven operator placement* to mitigate cache thrashing
  - *Query chopping* to mitigate GPU heap contention
    - Limited amount of operators in parallel on GPUs



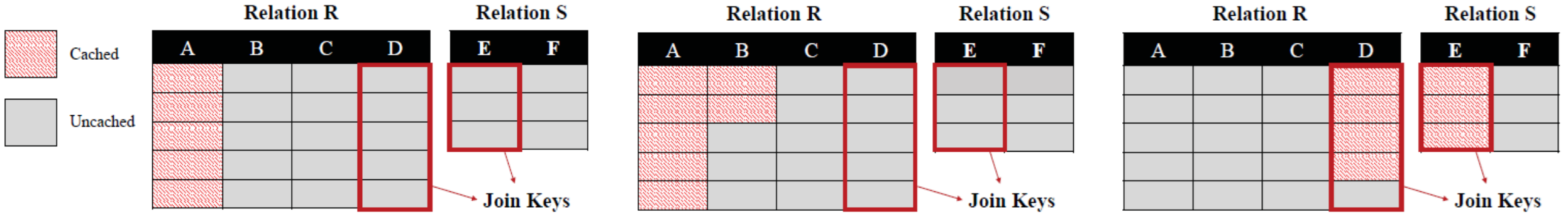
# Placement Strategy: function shipping (example 2)

**e.g. prioritize processors where fine-grained data is local**: Mordred [Yogatama et al.@PVLDB'22]

- Key principles
  - Places table column segments in GPU memory; keeps a copy of the entire database in CPU (main memory)
  - Ships query operators (selection, join, and group-by agg) on each processor based on segment locality
- Contributions
  - Fine-grained semantic-aware cache
  - Segment-level query plan (details in Placement Granularity->Segment section)

e.g. GPU cache size = 7 segments

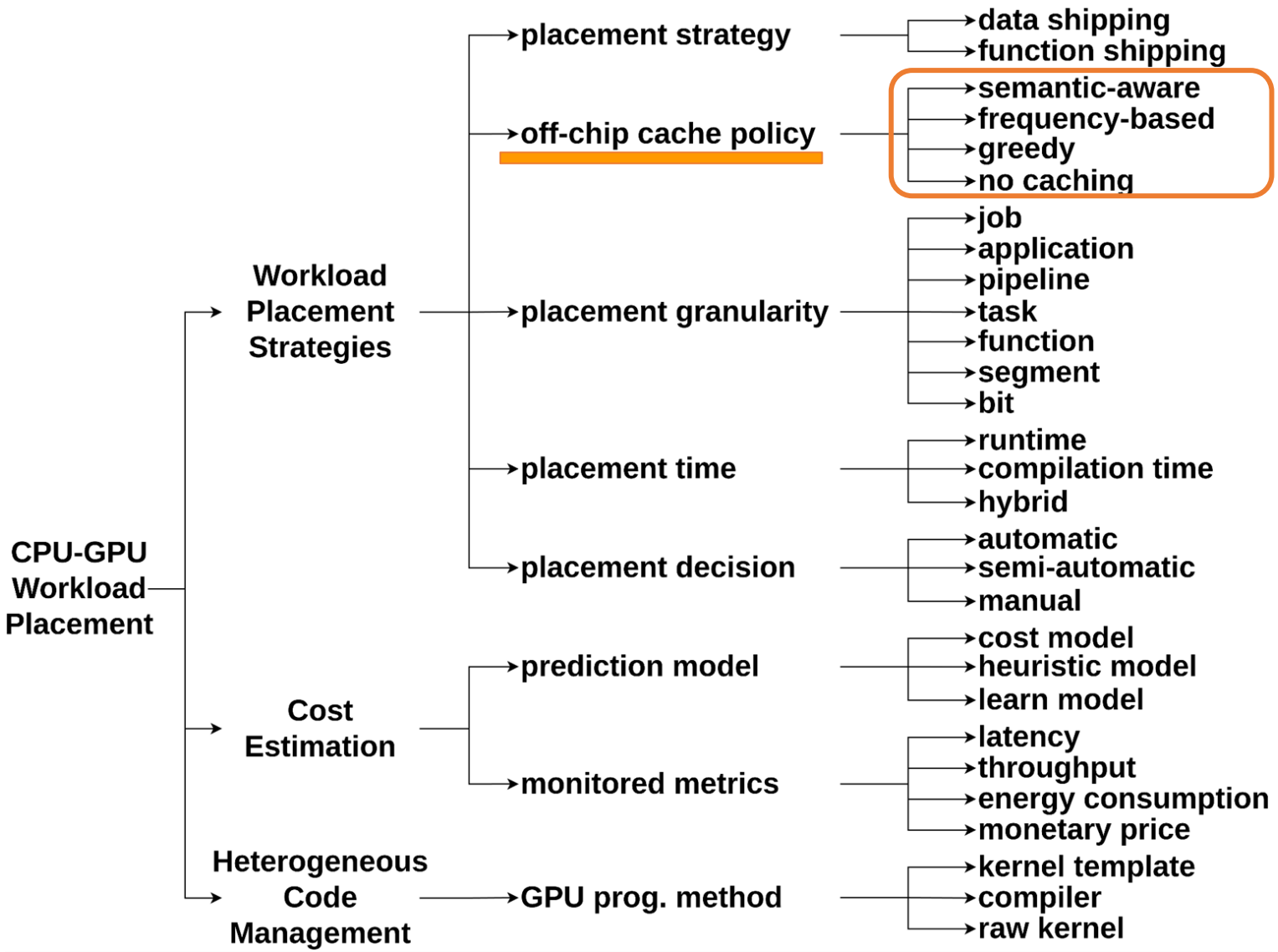
```
Qx: SELECT R.B, SUM(R.A) FROM R,S
    WHERE R.D = S.E

Qy: SELECT R.A AND R.B FROM R
```




Coarse-grained caching (e.g. LFU)
- partial use of cache size
- inefficient cache

Fine-grained caching (e.g. LFU)
+ full use of cache size
- inefficient cache

Fine-grained semantic-aware caching
+ full use of cache size
+ efficient cache


CPU-GPU Workload Placement
Workload Placement Strategies
placement strategy
data shipping
function shipping
off-chip cache policy
semantic-aware
frequency-based
greedy
no caching
placement granularity
job
application
pipeline
task
function
segment
bit
placement time
runtime
compilation time
hybrid
placement decision
automatic
semi-automatic
manual
Cost Estimation
prediction model
cost model
heuristic model
learn model
monitored metrics
latency
throughput
energy consumption
monetary price
Heterogeneous Code Management
GPU prog. method
kernel template
compiler
raw kernel

# Off-chip Cache Policy: overview

- No caching
  - Most papers (50) have not reported any cache strategy
  - Reasons for no caching in analytics [Nicholson et al.@CIDR'23]
    - Several complex challenges (e.g. defining the ideal cache size, implementing an efficient cache policy)
    - High bandwidth storage alternatives (e.g. NVMe)

**Cache-based approaches keep a subset of data in off-chip memory to reduce data movements**

- Greedy
  - Caches as many data as possible (assumes that cache benefits are linear to the cache size)
  - **Pros**: intuitive implementation
  - **Cons**: might cache more data than necessary
  - e.g. CoTrain, Parla, [Kroviakov et al.@DAMON'24], HyGraph
- Frequency-based
  - Caches data based on its access frequency and/or recency of use
  - **Pros**: popular algorithms, suitable when storage is a significant bottleneck
  - **Cons**: not suitable in high-bandwidth storage/interconnect or compute-bound workloads
- Semantic-aware
  - Caches data according to its use on a workload
  - **Pros**: workload and hardware heterogeneity-aware, efficient cache utilization
  - **Cons**: complex and ad-hoc implementation

# Off-chip Cache Policy: frequency-based (overview)

Aims to minimize storage accesses by maximizing the cache hit rate

- **LRU**: evicts least recently used data using a timestamp
  - RateupDB
  - Ray
  - HERO
  - Robust CoGaDB
  - Spark-GPU
  - Dask
  - Ocelot+HyPE
  - A&R

  **pros**: suitable for data frequently used during a period and then usage drops
  **cons**: memory overhead to update timestamps and maintain access order
- **LFU**: evicts least frequently used data using a frequency counter
  - Robust CoGaDB
  - PyCOMPSs (v. 3.2+)

  **pros**: reduced memory overhead (frequency counter updates)
  **cons**: inefficient for sudden drop on a frequent-accessed data
- **FIFO**: evicts the oldest data in the cache queue
  - GFlink

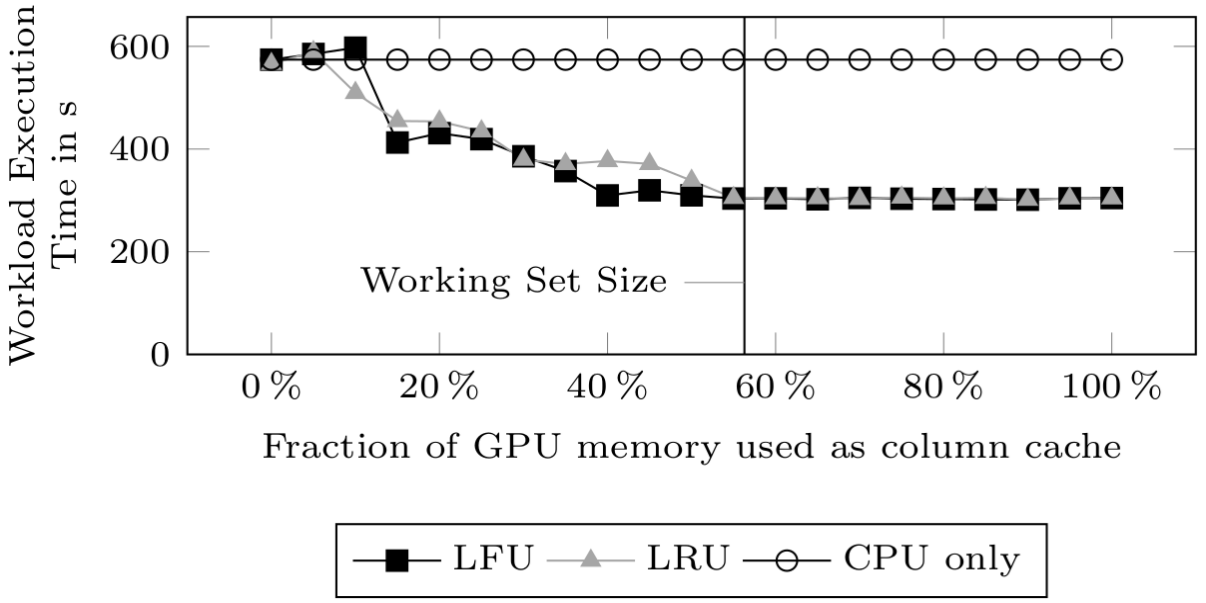
  **pros**: simple implementation, no extra memory required to monitor access statistics
  **cons**: inefficient when data access patterns do not match with the order of insertion

# Off-chip Cache Policy: frequency-based (example)

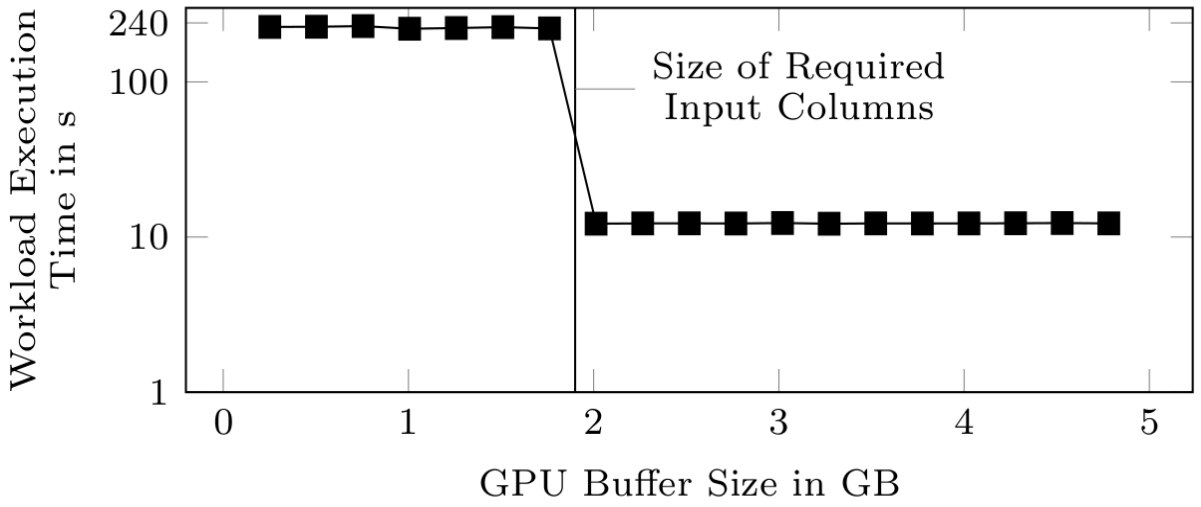
Robust CoGaDB [Breß@SIGMOD'16]

- Key principles
  - Uses LRU and LFU to keep in GPU memory table columns used in query operators
- Contributions
  - Compares LRU and LFU and concludes that both have similar performance
  - Introduces the *cache thrashing* issue and proposes *data-driven operator placement* to mitigate it

LRU *vs*. LFU on SSBM (SF=10)




The *caching thrashing* effect



# Off-chip Cache Policy: semantic-aware (overview)

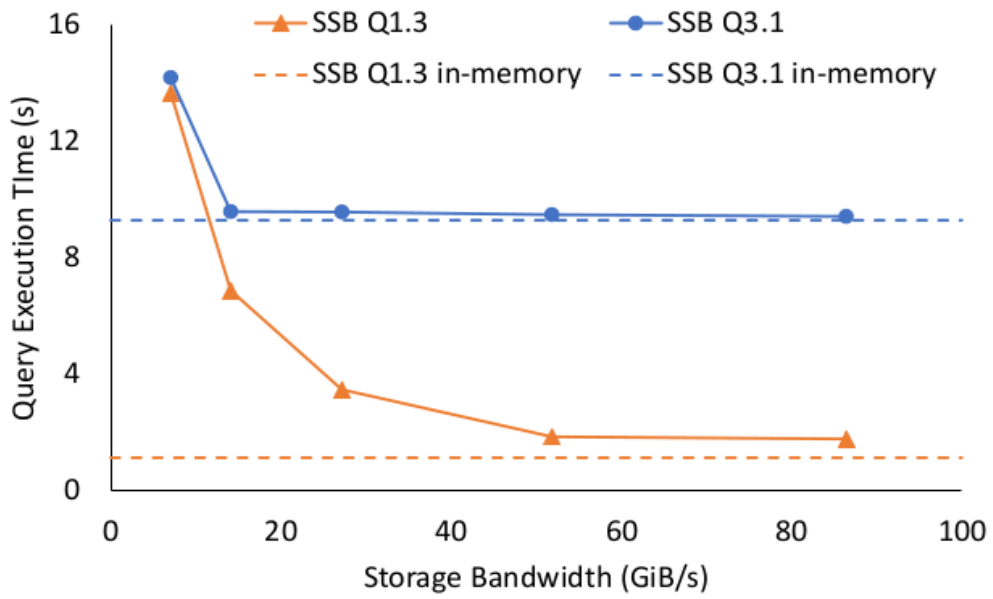
Caches the data with the most impact on workload execution (it depends on the application)

- **Database**
  - **OLAP query processing**
    - **HetCache**: Caches in GPU memory only table columns that accelerate queries the most w.r.t. query processing throughput
    - **Mordred**: LFU with weighted frequency counters representing the potential benefits of caching a segment
  - **OLTP query processing**
    - **GaccO**: Caches in GPU memory only tables used in dominant OLTP transactions (%total workload)
- **Machine learning training**
  - **NeutronOrch**: Frequency-based cache is used in CPU memory, but if GPU is idle, cache hot vertices in GPU memory and ship embedding computations from the bottom layer of GNN to GPU
- **Graph processing**
  - **CGgraph**: keeps only active edges and vertices of graphs (subgraph) in GPU memory

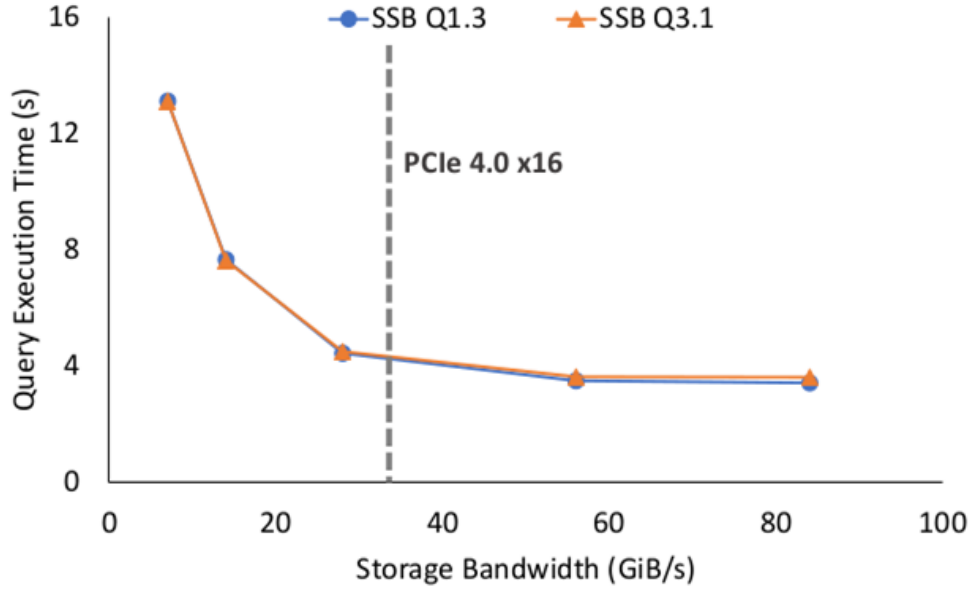
# Off-chip Cache Policy: semantic-aware (example)

HetCache [Nicholson et al.@CIDR'23]

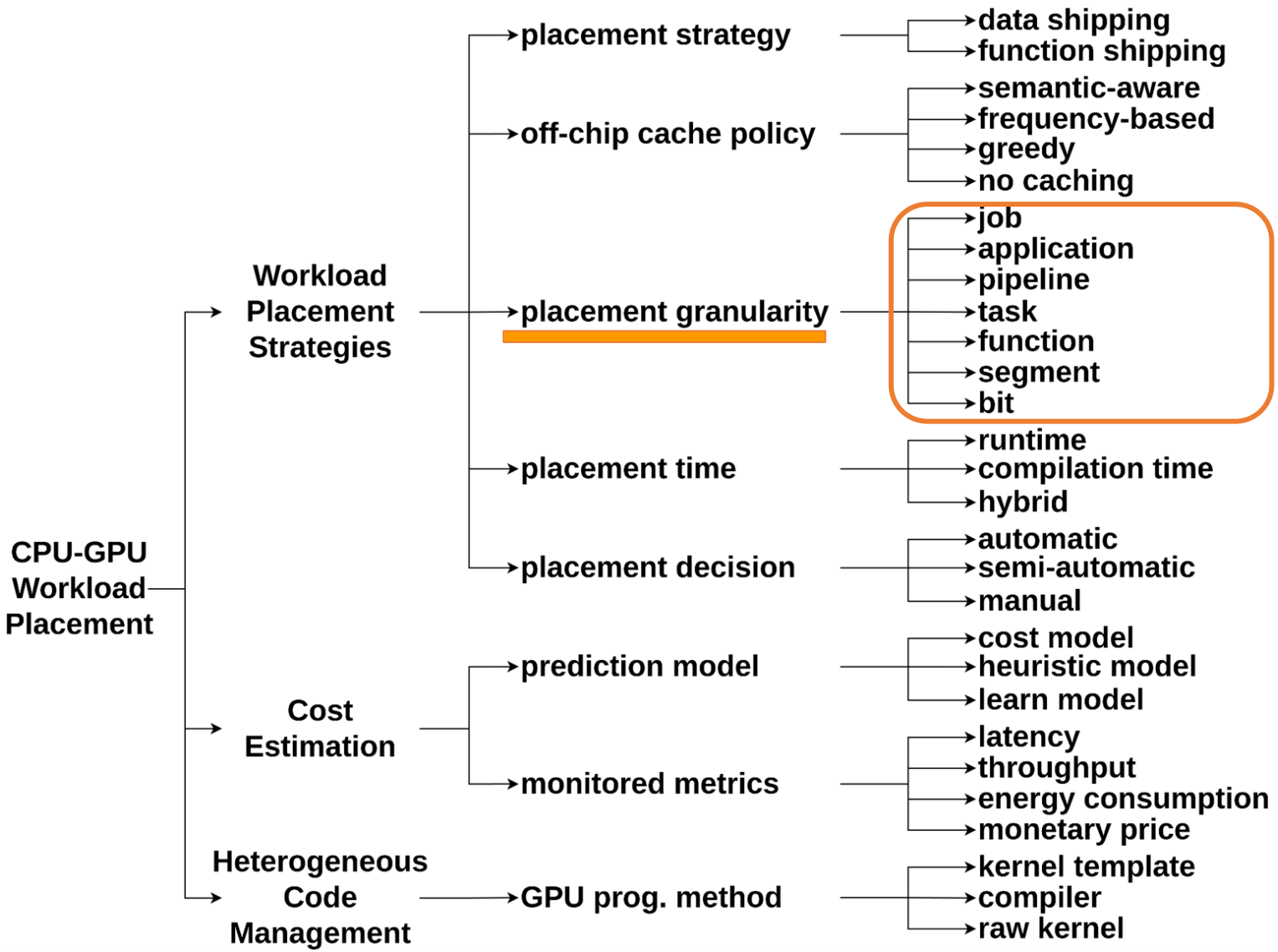
- Key principles
  - Wasteful hardware utilization results from caching decisions that have little impact on query execution
  - The benefit of caching a specific data depends on its use:
    - e.g. caching two data pages (A and B) with the same access frequency
      - A is used in IO-bound task (Q1.3) → Caching A will provide greater benefits than caching B
      - B is used in compute-bound task (Q3.1)



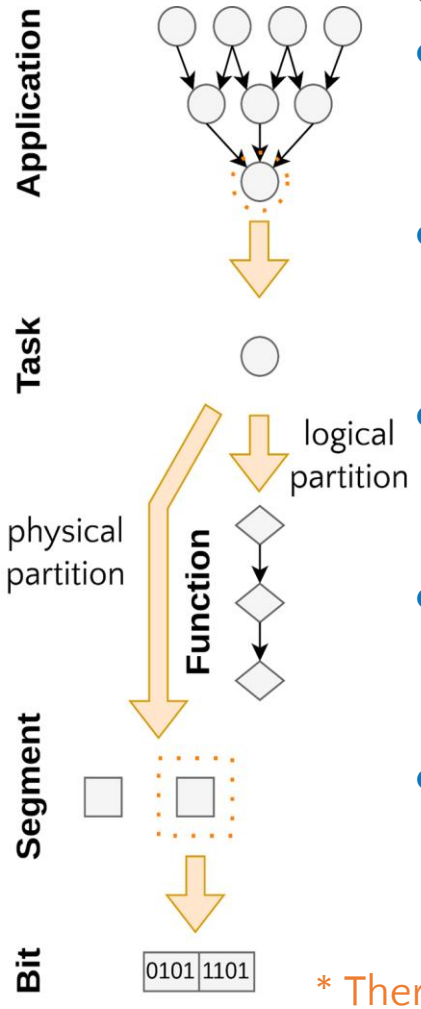

CPU execution




GPU execution


CPU-GPU Workload Placement
Workload Placement Strategies
placement strategy
data shipping
function shipping
off-chip cache policy
semantic-aware
frequency-based
greedy
no caching
placement granularity
job
application
pipeline
task
function
segment
bit
placement time
runtime
compilation time
hybrid
placement decision
automatic
semi-automatic
manual
Cost Estimation
prediction model
cost model
heuristic model
learn model
monitored metrics
latency
throughput
energy consumption
monetary price
Heterogeneous Code Management
GPU prog. method
kernel template
compiler
raw kernel

# Placement Granularity: overview




- Defines the level of abstraction/size of the functions and/or data to place
- Application*
  - **Pros**: reduced amount of data transfers
  - **Cons**: high memory footprint, limited parallelism opportunities, load imbalance
- Task**
  - **Pros**: many parallelism opportunities, load balance
  - **Cons**: increased amount of data transfers
- Function (logical partition)
  - **Pros**: high degree of code specialization, reduced code complexity (less runtime variation)
  - **Cons**: increased amount of data transfers, load imbalance
- Segment (physical partition)
  - **Pros**: efficient memory use, load balance
  - **Cons**: increased amount of data transfers, merge/synchronization overhead
- Bit (physical partition)
  - **Pros**: reduced volume of data per transfer
  - **Cons**: complex implementation

* There are works scheduling jobs (multiple concurrent applications), but the placement is still at application level

** Tasks can also be grouped or pipelined and placed together at a coarser granularity to reduce data transfers

# Placement Granularity: application (overview)

- **Implements specific kernel(s) per application**
  - **Pros**: fine-tune optimization
  - **Cons**: low kernel reuse, complex and ad-hoc implementation
  - **e.g.** GaccO, [Cumming@SC'14], [Kerr et al.@ASPLOS'10]

- **Applications implemented via block-wide function kernels**
  - **Pros**: great opportunities for kernel reuse, simplification of code implementation
  - **Cons**: potential increase on the amount of data transfers and kernel warm-up overheads
  - **e.g.** GAP, RateupDB, Compressed Crystal, Crystal, Caldera, SABER

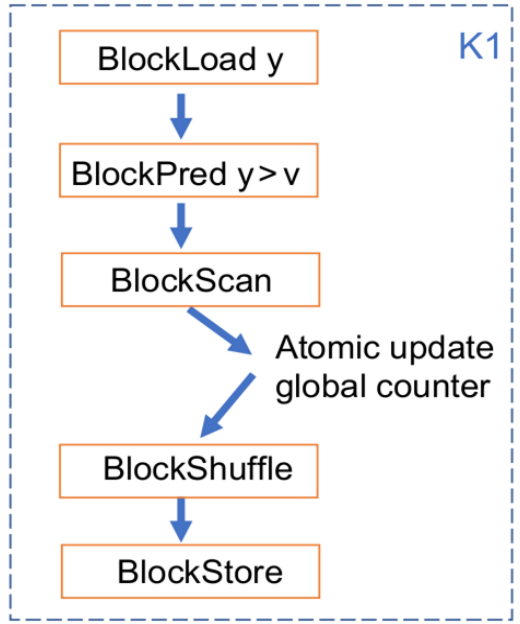
# Placement Granularity: application (example)

**e.g. block-wide function**: Crystal, Compressed Crystal [Shanbhag et al.@SIGMOD'20, SIGMOD'22]

- Key principle
  - Accelerates analytical queries (SSB benchmark) focusing on selection, projection, and join operators
- Contribution
  - Proposes a library of query primitives in CUDA that are combined to run entire queries on GPUs
    - Primitives are implemented in *block-wide functions* providing:
      - Code modularity
      - Code extensibility

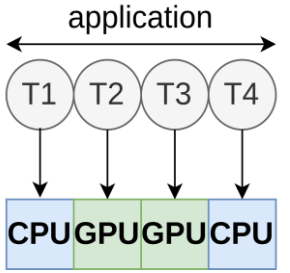
| Primitive | Description |
|---|---|
| `BlockLoad` | Copies a tile of items from global memory to shared memory. Uses vector instructions to load full tiles. |
| `BlockLoadSel` | Selectively load a tile of items from global memory to shared memory based on a bitmap. |
| `BlockStore` | Copies a tile of items in shared memory to device memory. |
| `BlockPred` | Applies a predicate to a tile of items and stores the result in a bitmap array. |
| `BlockScan` | Co-operatively computes prefix sum across the block. Also returns sum of all entries. |
| `BlockShuffle` | Uses the thread offsets along with a bitmap to locally rearrange a tile to create a contiguous array of matched entries. |
| `BlockLookup` | Returns matching entries from a hash table for a tile of keys. |
| `BlockAggregate` | Uses hierarchical reduction to compute local aggregate for a tile of items. |

SELECT y FROM R WHERE y > v

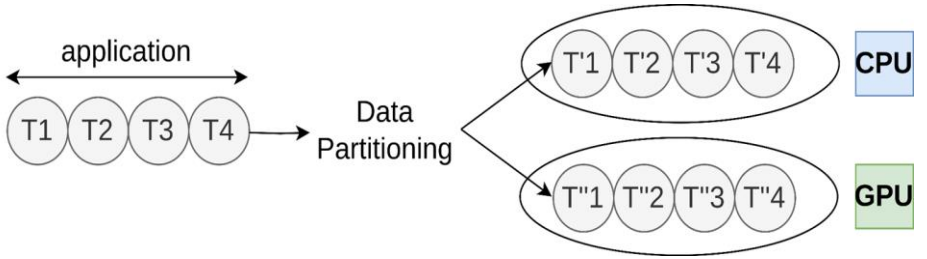


# Placement Granularity: task (overview)

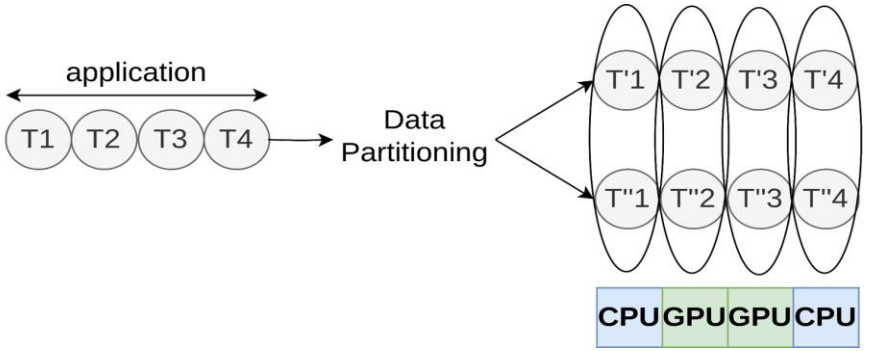
- **Task placement without data partitioning**
  - **Pros**: reduced amount of data transfers
  - **Cons**: high memory footprint, load imbalance
  - **e.g.** MCL Schedulers, [Karnagel et al.@EDBTW'15], Symphony, [Wen and O'Boyle@GPGPU-PPoPP'17]




- **Data partitioning with horizontal device parallelism**
  - **Pros**: low memory footprint, load balance
  - **Cons**: increased amount of data transfers, low data locality, CPU-GPU synchronization
  - **e.g.** [Kroviakov et al.@DAMON'24], NeutronOrch, FusionFlow, CGgraph, CoTrain, Parla, HetExchange, Qilin



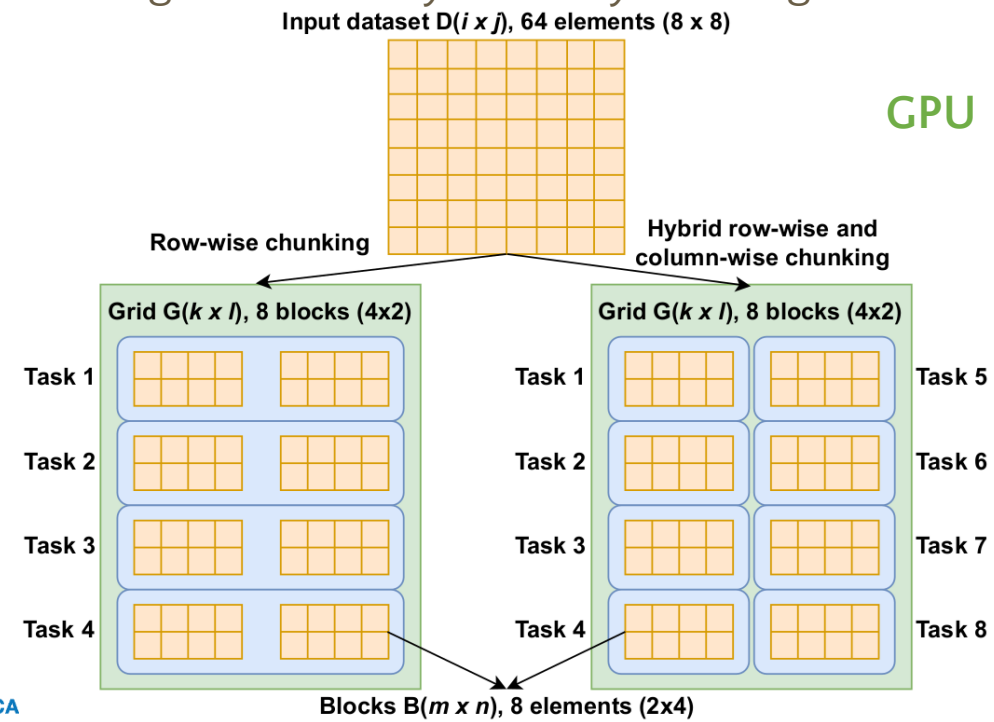

- **Data partitioning with vertical device parallelism**
  - **Pros**: low memory footprint, high data locality
  - **Cons**: increased amount of data transfers, load imbalance
  - **e.g.** [Carvalho et al.@EDBT'24], [Lee and Park@ICDEW'21], Spark-GPU, Dask, PyCOMPSs

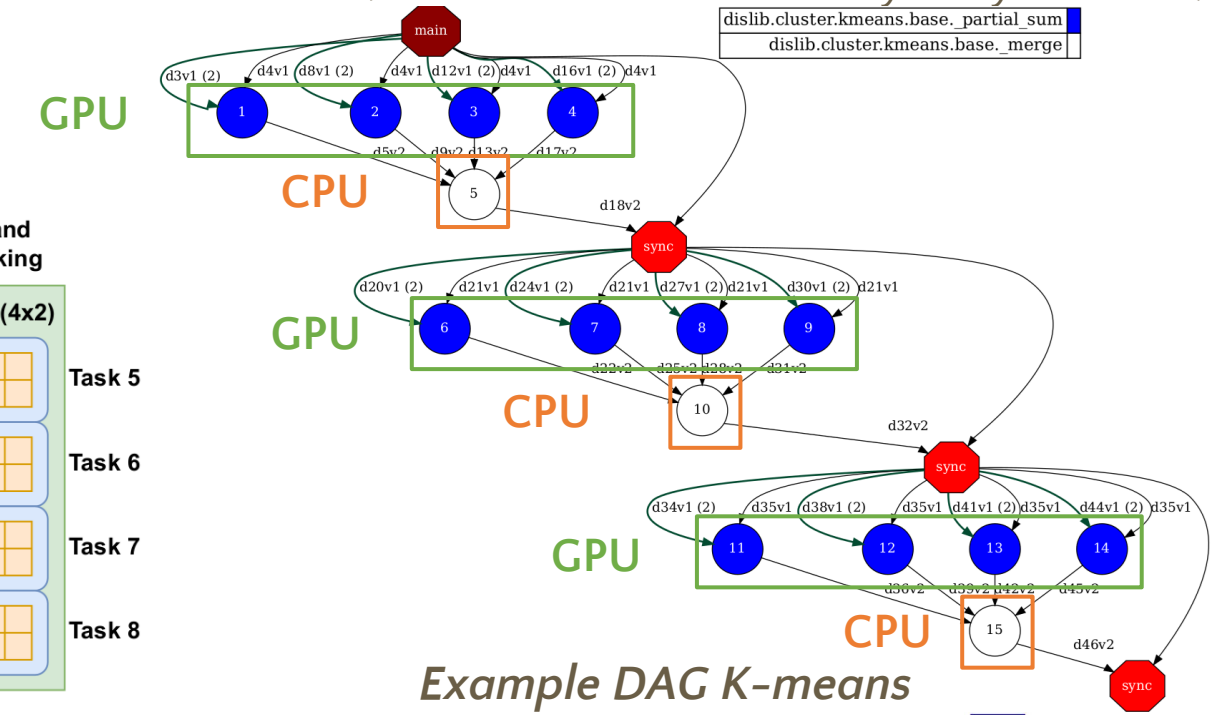


# Placement Granularity: task (example)

**e.g. data partitioning with vertical device parallelism**: PyCOMPSs [Amela et al. @OGTS'18]

- Key principles
  - Tasks are annotated functions in Python applications
  - Task dependency graph represented as a DAG (Directed Acyclic Graph)
- Contributions
  - A GPU-accelerated distributed processing system in Python
  - Data partitioning is defined by users by defining block size in *dislib* (distributed *sklearn* library in PyCOMPSs)



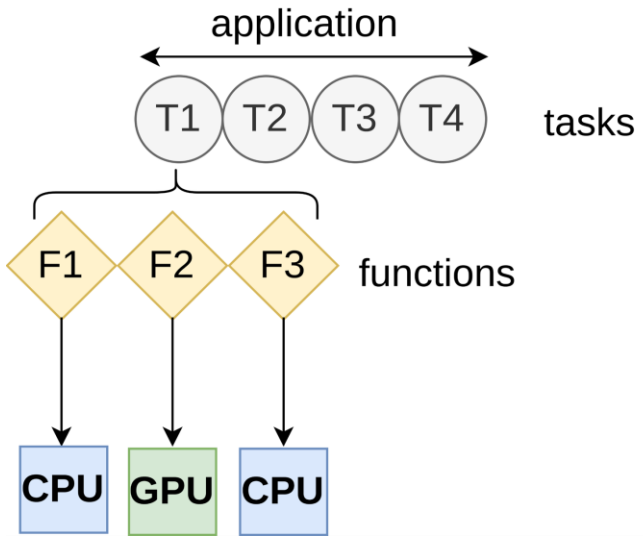




*Example DAG K-means*

# Placement Granularity: function (overview)

Function-based approaches depend on the application

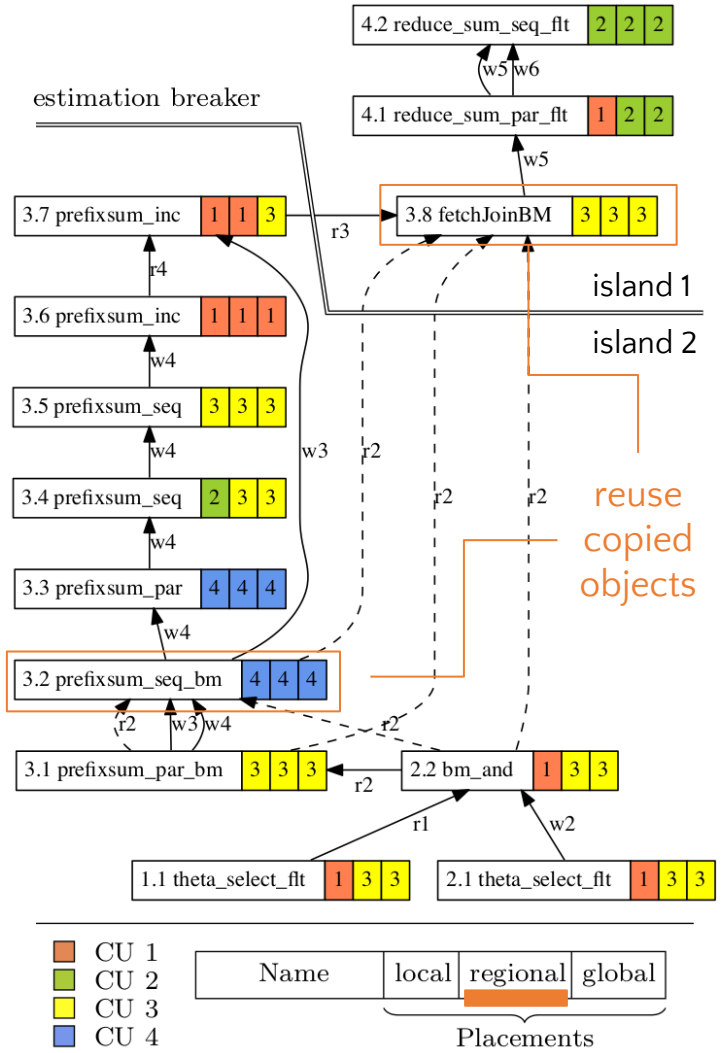
- **Function placement in streaming processing**
  - **LDWP-IAWP**: places functions (subtasks in streaming tasks) with smallest predicted GPU/CPU speedup on CPU
- **Function placement in OLAP query processing**
  - **HERO**: function (query sub-operator) placement to improve execution time predictions



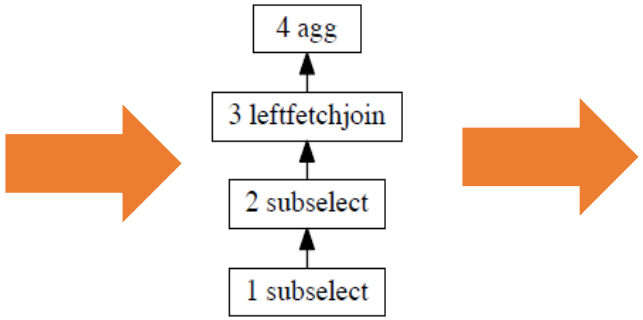
# Placement Granularity: function (example)

**function placement in OLAP query processing**: HERO [Karnagel et al.@PVLDB'17]

- Key principle
  - Places sub-operators (recurring functions executed in query operators) used in OLAP queries
- Contribution
  - Query plan division scheme into execution islands (regional placement)
    - Intermediate cardinality is precisely known (no cardinality estimation)
      - Precise estimation of execution time and data transfer costs
    - Keeps copies of intermediate results on multiple processors
      - Locality-aware approach:
        - Places a sub-operator on the CU where an object copy resides



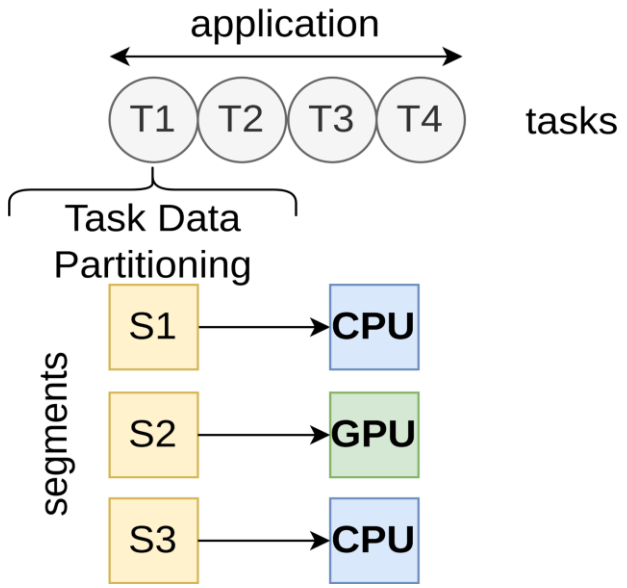

$$SELECT\ sum(l\_quantity)\ FROM\ lineitem\ WHERE\ l\_discount < 20\ and\ l\_quantity < 24$$



# Placement Granularity: segment (overview)

Segment-based approaches depend on the application

- **Segment placement in graph processing**
  - **gSWORD**: places sampling tasks on GPU and enumeration tasks on CPU to balance efficiency (execution time) and accuracy (amount of valid samples) in subgraph counting
  - **HyGraph**: places states update tasks over vertice blocks (consecutive fixed-sized segments) of a graph
- **Segment placement in streaming processing**
  - **Rectangle method**: places streams partitions ensuring both deadline constraint and throughput requirements
- **Segment placement in OLAP query processing**
  - **[Kroviakov et al.@DAMON'24]**: allows aggregations to execute different fragments (horizontal partitions of a data table with a fixed number of tuples)
  - **Mordred**: allows different query plans to execute different segments of a column based on segments' locality

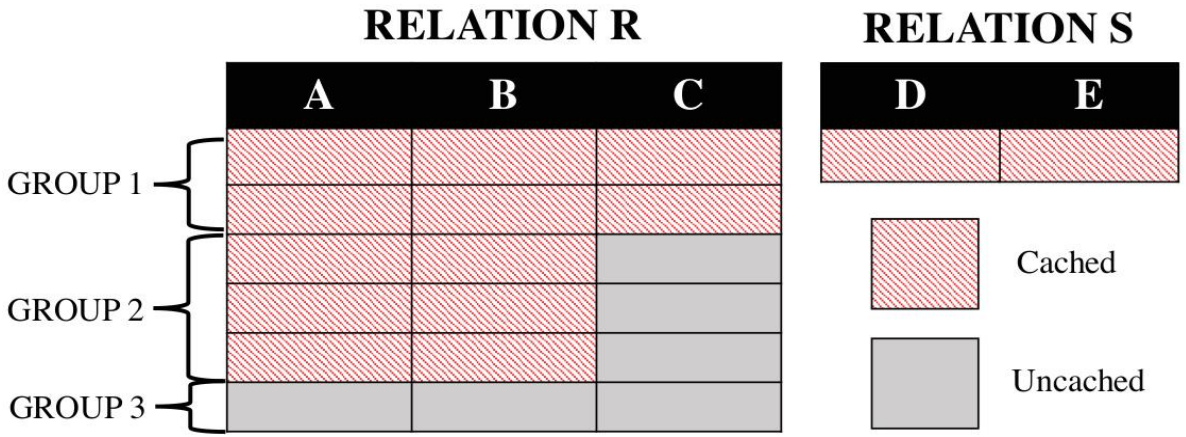


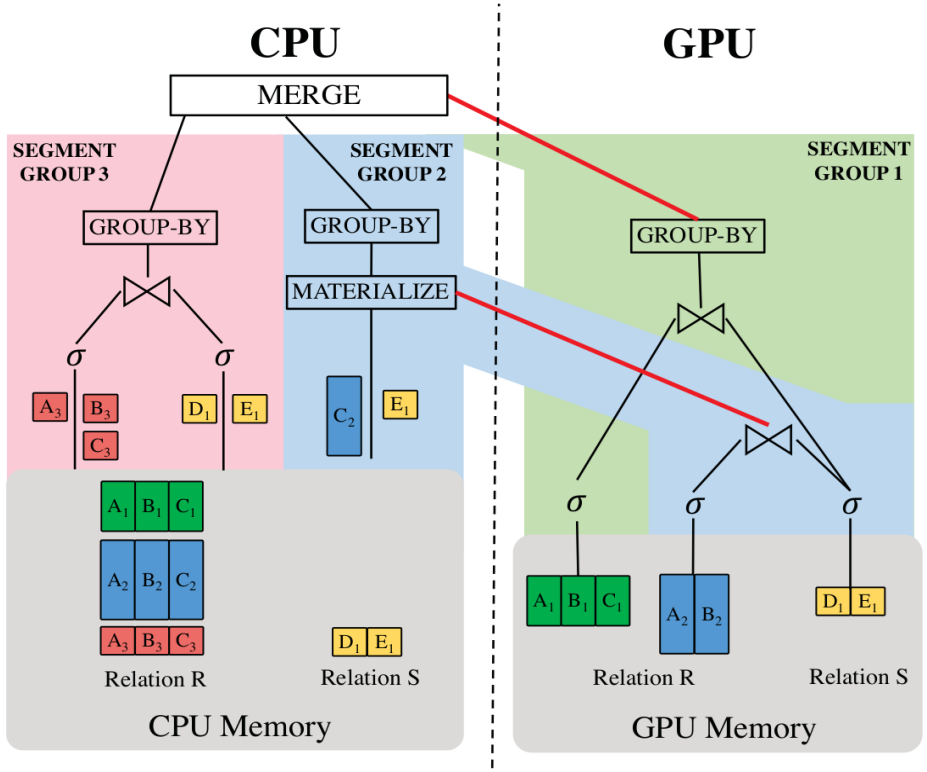
# Placement Granularity: segment (example)

Mordred [Yogatama et al.@PVLDB'22]

- Key principles
  - Selects segments (part of table's columns) to cache and group them accordingly
  - Places query operators according to the segments' locality
- Contributions
  - Semantic-aware caching
  - Segment-level query plan

```
SELECT S.D, SUM(R.C) FROM R,S
WHERE R.B = S.D AND R.A > 10 AND S.E > 20
GROUP BY S.E
```



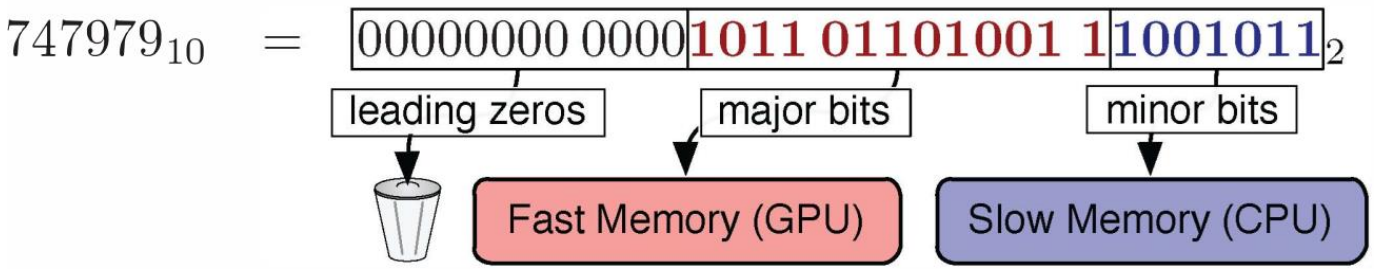



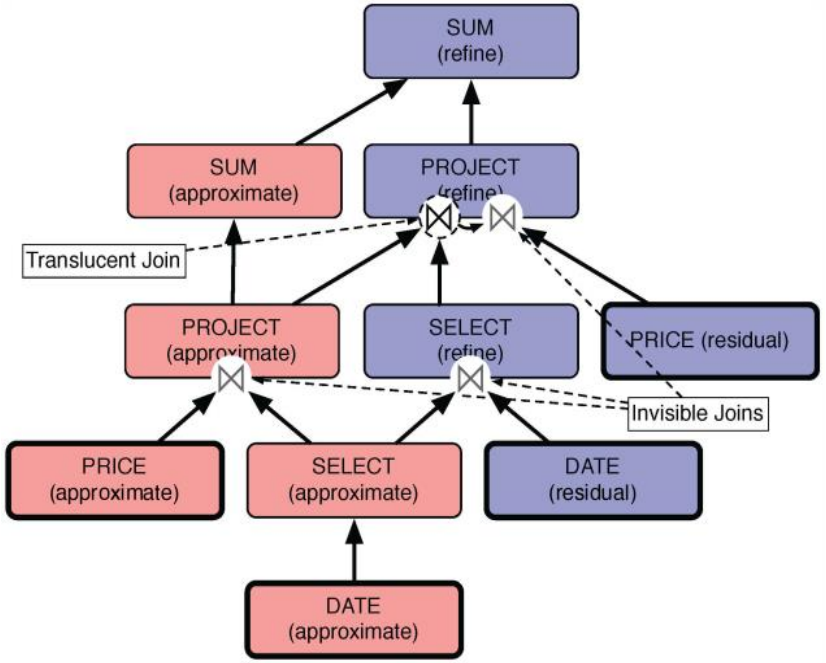
# Placement Granularity: bit (overview and example)

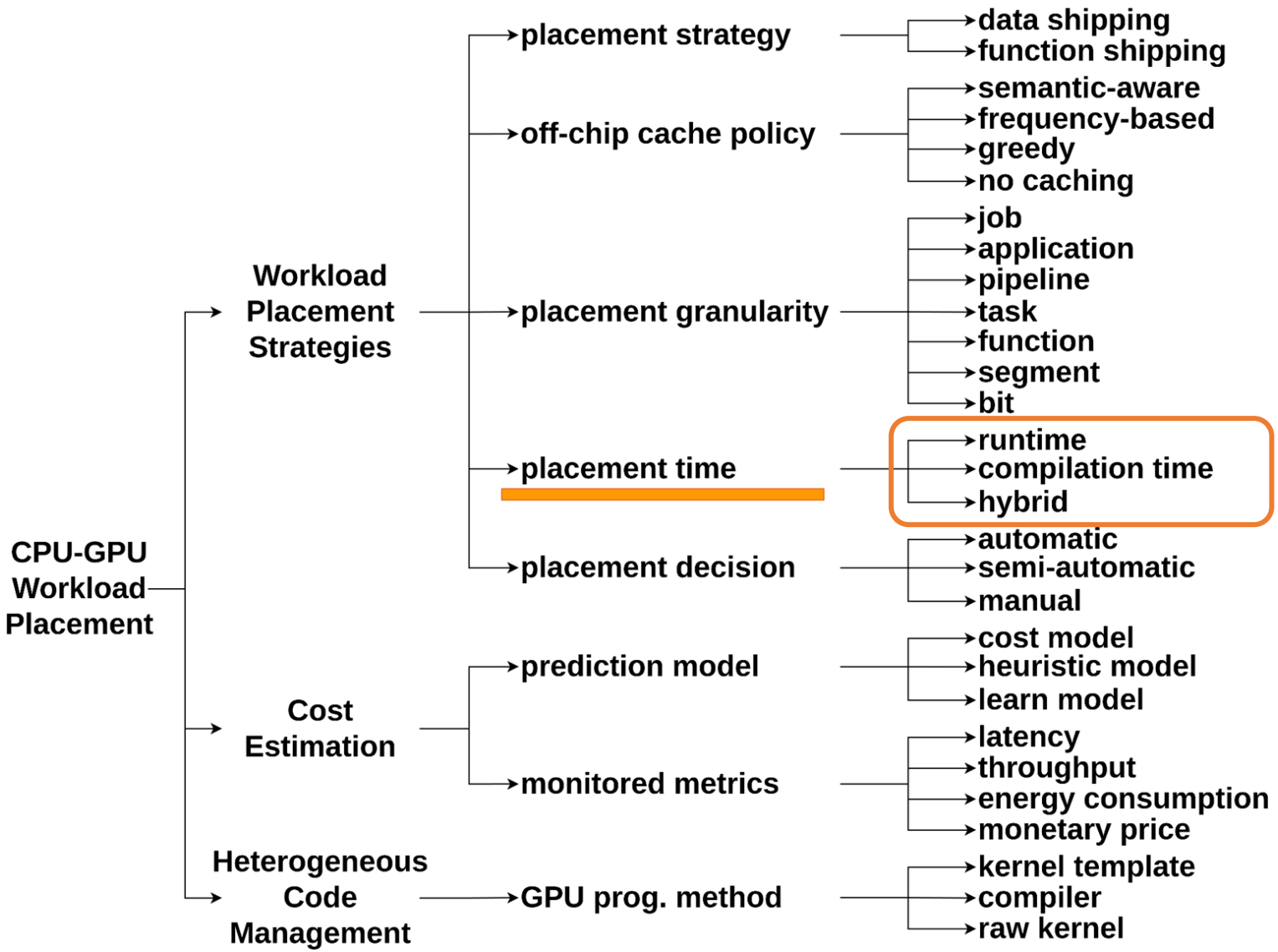
A&R[Pirk et al.@ICDE'14], [Pirk@PVLDB'12]

- Key principles
  - Bitwise decomposition and partition of data for query processing
  - Place higher/lower bits on GPU/CPU
- Contributions
  - Approximate & Refine (A&R) query plan model
    - GPUs compute approximate results (lossy compression)
    - CPUs refine GPUs approximations into exact results
  - First approach to enable query acceleration on data greater than GPU memory and minimize CPU-GPU communication

Data values of a table column




CPU-GPU Workload Placement
Workload Placement Strategies
placement strategy
data shipping
function shipping
off-chip cache policy
semantic-aware
frequency-based
greedy
no caching
placement granularity
job
application
pipeline
task
function
segment
bit
placement time
runtime
compilation time
hybrid
placement decision
automatic
semi-automatic
manual
Cost Estimation
prediction model
cost model
heuristic model
learn model
monitored metrics
latency
throughput
energy consumption
monetary price
Heterogeneous Code Management
GPU prog. method
kernel template
compiler
raw kernel

# Placement Time: overview

- Defines when the placement decisions are made
- Runtime (local, dynamic)
  - Placement is decided right before the workload execution
  - **Pros**: small search space, simple implementation, little computational overhead (but at runtime), input data (e.g. cardinality) precisely known, flexible to react to unforeseen events
  - **Cons**: not fully informed (e.g. DAG structure), worst-case placement worse than single processor execution
- Compilation time (global, static)
  - Placement is decided in advance during compilation phase
  - **Pros**: fully informed (e.g. DAG structure), worst-case placement better than single processor execution
  - **Cons**: huge search space, complex implementation, high computational overhead (but not at runtime), need to estimate input data, inflexible to react to unforeseen events
- Hybrid
  - An initial placement is defined at compilation time and updated at runtime
  - Same **pros** and **cons** of runtime and compilation time approaches

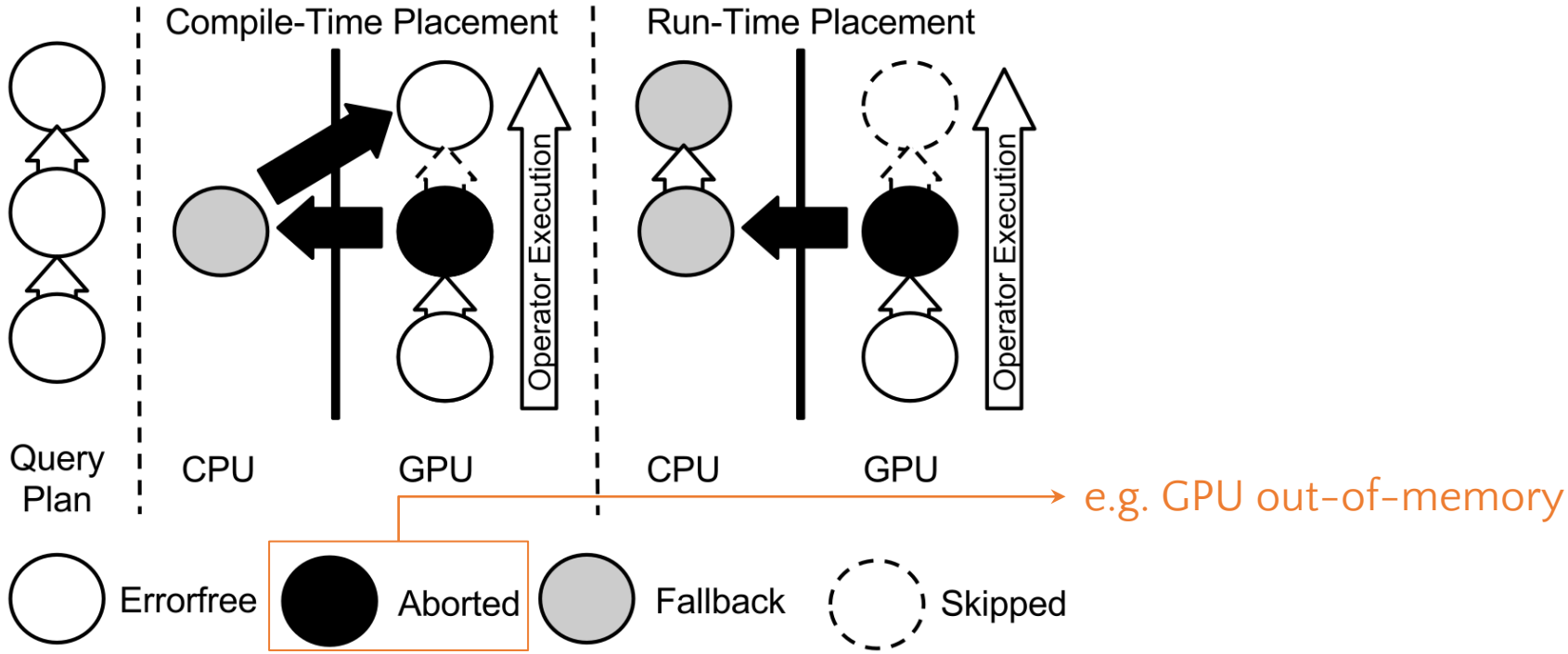
# Placement Time: runtime and compilation time (overview)

- Runtime
  - **Placement based on load balance** (requires runtime resources statistics, typical in schedulers)
    - **Pros**: better utilization of resources, high throughput (CPU-GPU parallelism)
    - **Cons**: not always use the most fit processor, increased amount of data transfers
    - **e.g.** NeutronOrch, FusionFlow, Parla, MCL Schedulers, [Gowanlock et al.@DaMoN'19], SABER, HyGraph
  - **Placement based on precise data size information**
    - **Pros**: reduced amount of data transfers (knowledge of data transfer costs)
    - **Cons**: not always use the most fit processor, load imbalance
    - **e.g.** NeutronOrch, HetCache, Parla, MCL Schedulers, Robust CoGaDB, [Li et al.@J. Syst. Arch'2021]
  - **Placement based on GPU memory size limitation**
    - **Pros**: reacts to GPU out-of-memory errors
    - **Cons**: may not use GPU again to avoid extra data transfers
    - **e.g.** Qilin, Heterogeneous Linpack, Robust CoGaDB, Hierarchical Planner
- Compilation time
  - **Placement based on precise data locality information**
  - **Pros**: reduced amount of data transfers (knowledge of data dependencies)
  - **Cons**: huge search space of possible placements
  - **e.g.** ADTS, TensorFlow, Spark-GPU

# Placement Time: runtime (example)

**e.g. data size info + GPU mem. limitation**: Robust CoGaDB [Breß et al.@SIGMOD'16]

- **Precise data size information**
  - Places tasks (query operators) on GPU if data is local and fits in GPU memory
- **GPU memory limitation awareness**
  - If an operator execution aborts, creates a fallback operator and runs it (and future operators) on CPU
  - Avoids additional data transfers at the cost of not having the acceleration of GPUs in the next operators

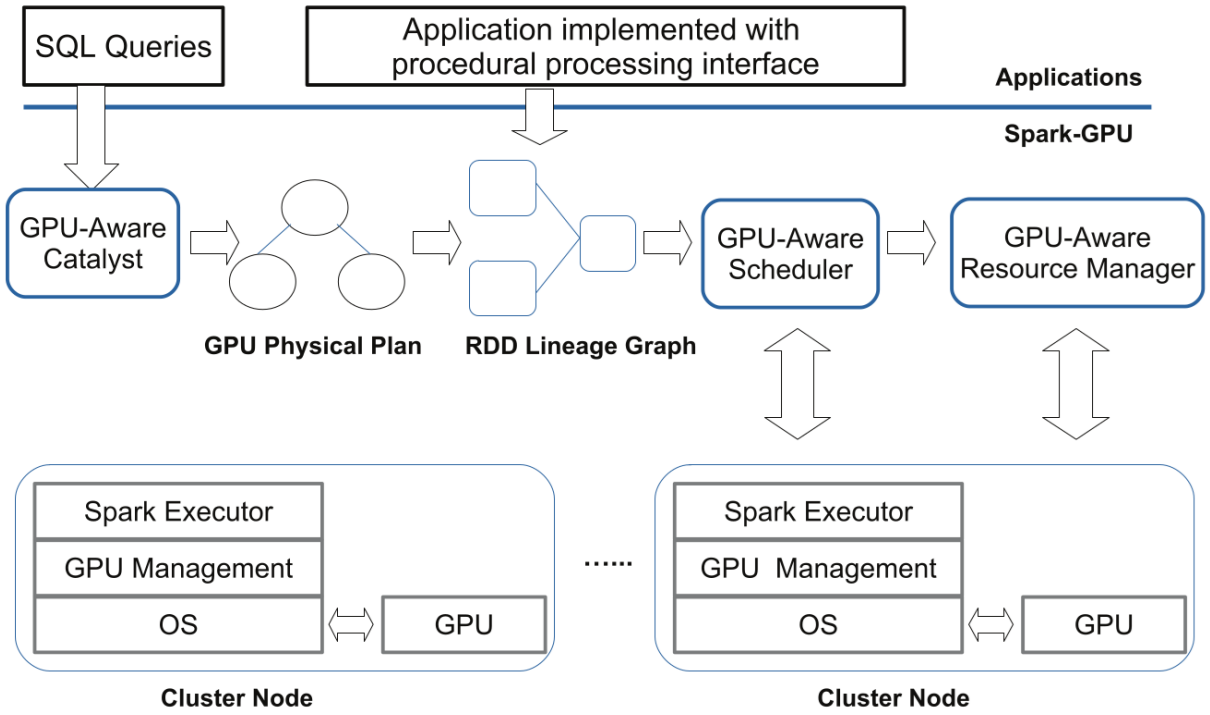


# Placement Time: compilation time (example)

**e.g. data locality info**: Spark-GPU [Yuan et al.@Big Data'16]

- Key principles
  - Implements Spark in GPU (incl. 5 key Spark operators and a row/column GPU-RDD data format)
  - Spark-GPU builds a heterogeneous CPU-GPU physical plan
- Contributions → **execution flow structure (DAG)**
  - Given a logical plan, use GPU operators if:
    - the operator is compute-intensive
    - the operator accesses the same data multiple times
    - there are multiple consecutive GPU operators on the data

→ **data locality information**



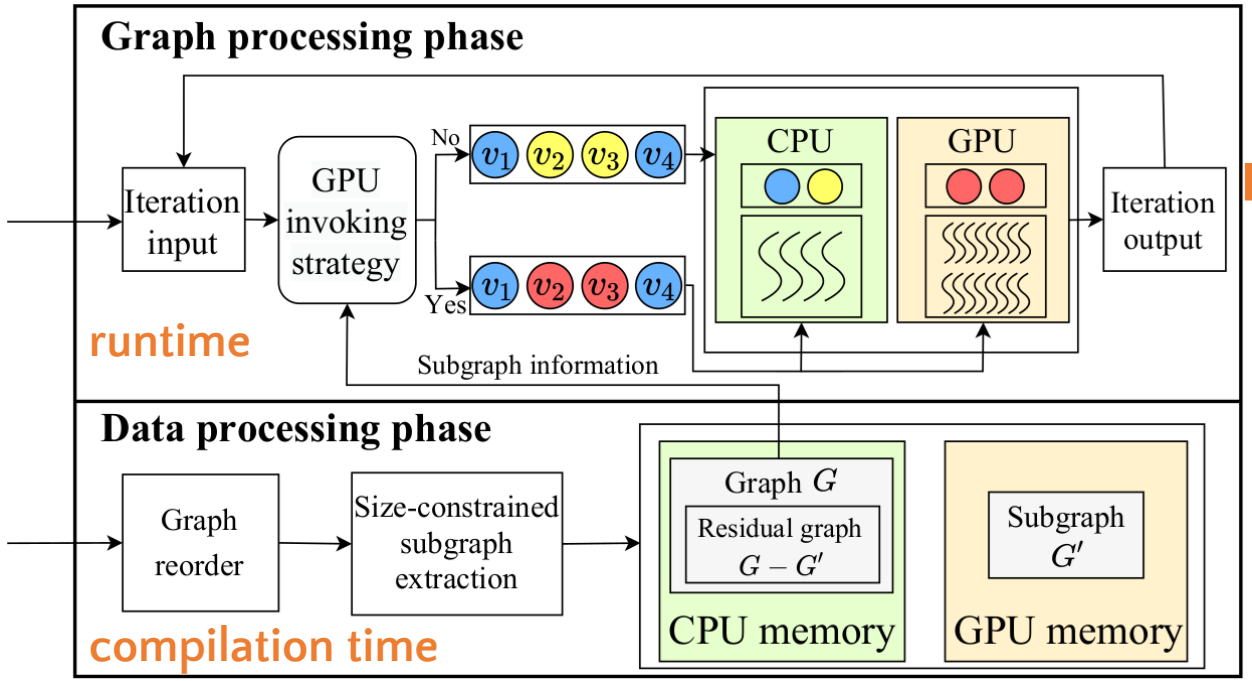
# Placement Time: hybrid (overview)

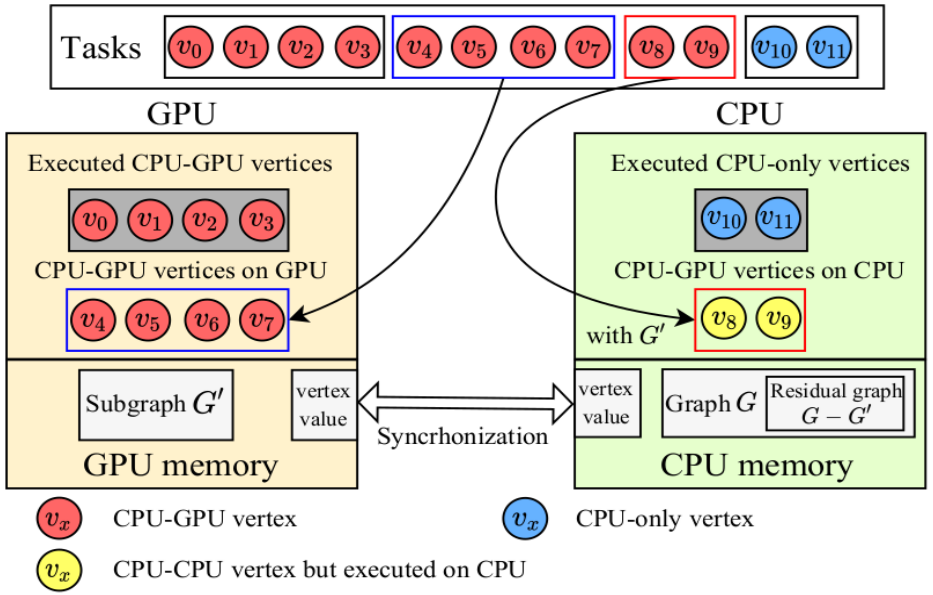
- **Placement based on load balance (same as in runtime approach)**
  - **e.g.** CGgraph, CoTrain, [Lutz@SIGMOD'20], HetExchange, PLB-HAC, [Kofler et al.@ICS'13], SCCG, AHP
- **Placement based on GPU memory size limitation (same as in runtime approach)**
  - **e.g.** CoTrain, GaccO, RateupDB, Mordred, [Lutz@SIGMOD'20], HetExchange
- **Placement based on data locality (same as compilation time approaches)**
  - **e.g.** CGgraph, CoTrain, Mordred, HetExchange, HERO, [Karnagel@EDBTW'15], AHP

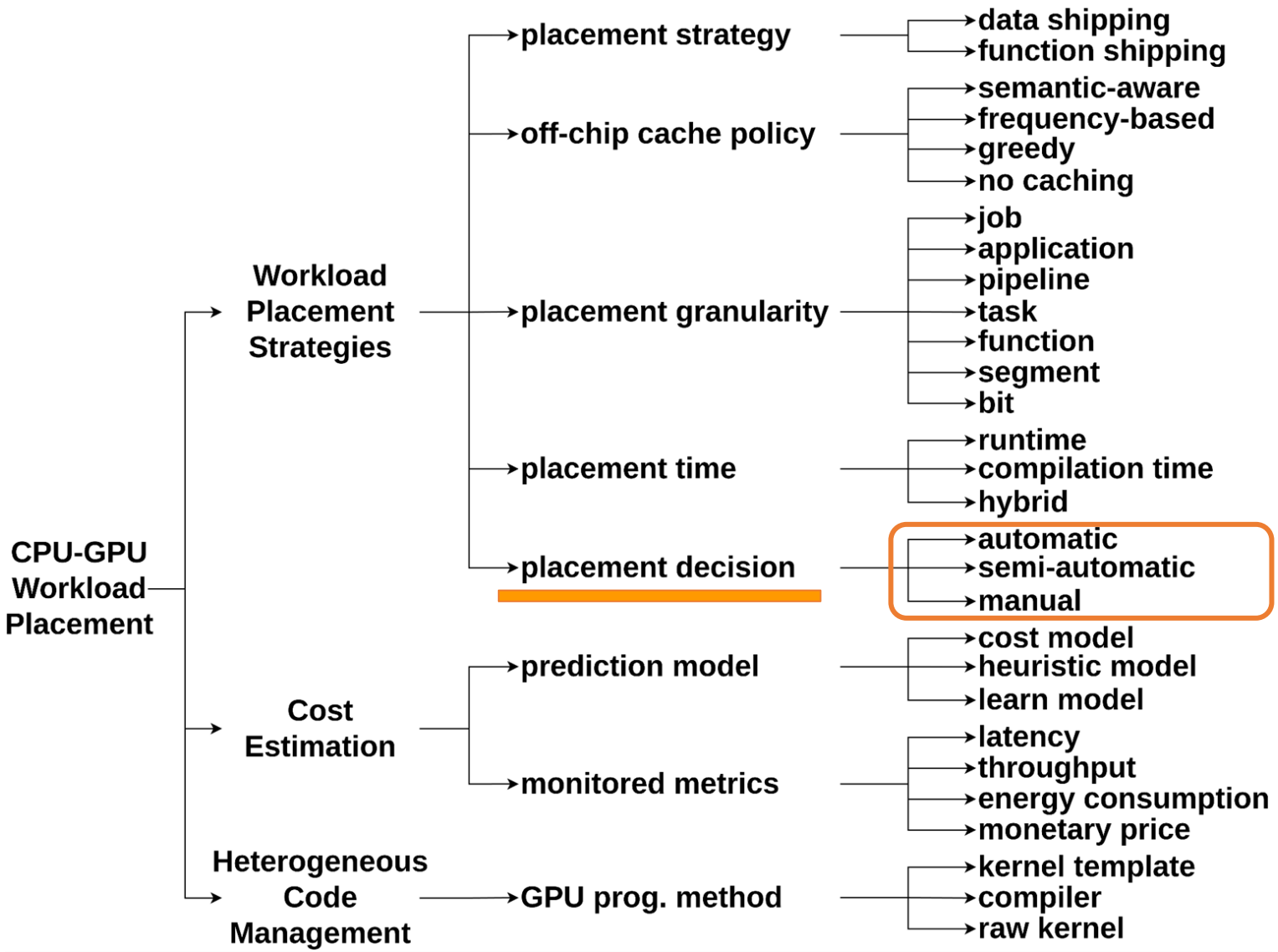
# Placement Time: hybrid (example)

**e.g. data locality+load balance**: CGgraph [Cui et al.@PVLDB'24]

- Key principles
  - CPU-GPU tasks are filter-expand operations of vertices in graph processing
  - **At compilation time:** Load the subgraph with active edges and vertices into GPU memory once and reuse it in multiple GPU tasks to improve data locality
  - **At runtime:** On-demand task allocation on CPU for CPU-GPU load balance




CPU-GPU Workload Placement
Workload Placement Strategies
placement strategy
data shipping
function shipping
off-chip cache policy
semantic-aware
frequency-based
greedy
no caching
placement granularity
job
application
pipeline
task
function
segment
bit
placement time
runtime
compilation time
hybrid
placement decision
automatic
semi-automatic
manual
Cost Estimation
prediction model
cost model
heuristic model
learn model
monitored metrics
latency
throughput
energy consumption
monetary price
Heterogeneous Code Management
GPU prog. method
kernel template
compiler
raw kernel

# Placement Decision: overview

- Defines the degree of automation on placement decisions
- Automatic
  - Fully automatic placement decisions
  - **Pros**: no human interaction, near-optimal placement decisions
  - **Cons**: complex implementation, ad-hoc design
- Semi-automatic
  - Automatic placement decision, but human interaction for tuning is required
  - **Pros**: flexible design, near-optimal placement decisions
  - **Cons**: partially dependent on human interactions
- Manual
  - Fully dependent on human interactions for placement decisions
  - **Pros**: no implementation effort required for placement
  - **Cons**: extensive manual profiling, static placement decisions, not scalable

# Placement Decision: automatic (overview)

- **Automatic data partition**
  - **Pros**: load balance, high utilization of resources (reduced idleness)
  - **Cons**: extra implementation complexity to estimate the most efficient data partition
  - **e.g.** FusionFlow, Kroviakov@DAMON'24, Sigmoid, [Lutz@SIGMOD'20], HyGraph, [Clarke@Euro-Par'12]

- **Automatic processor affinity classification**
  - **Pros**: use the most fit processor for each function/data class
  - **Cons**: load imbalance
  - **e.g.** RateupDB, DBD

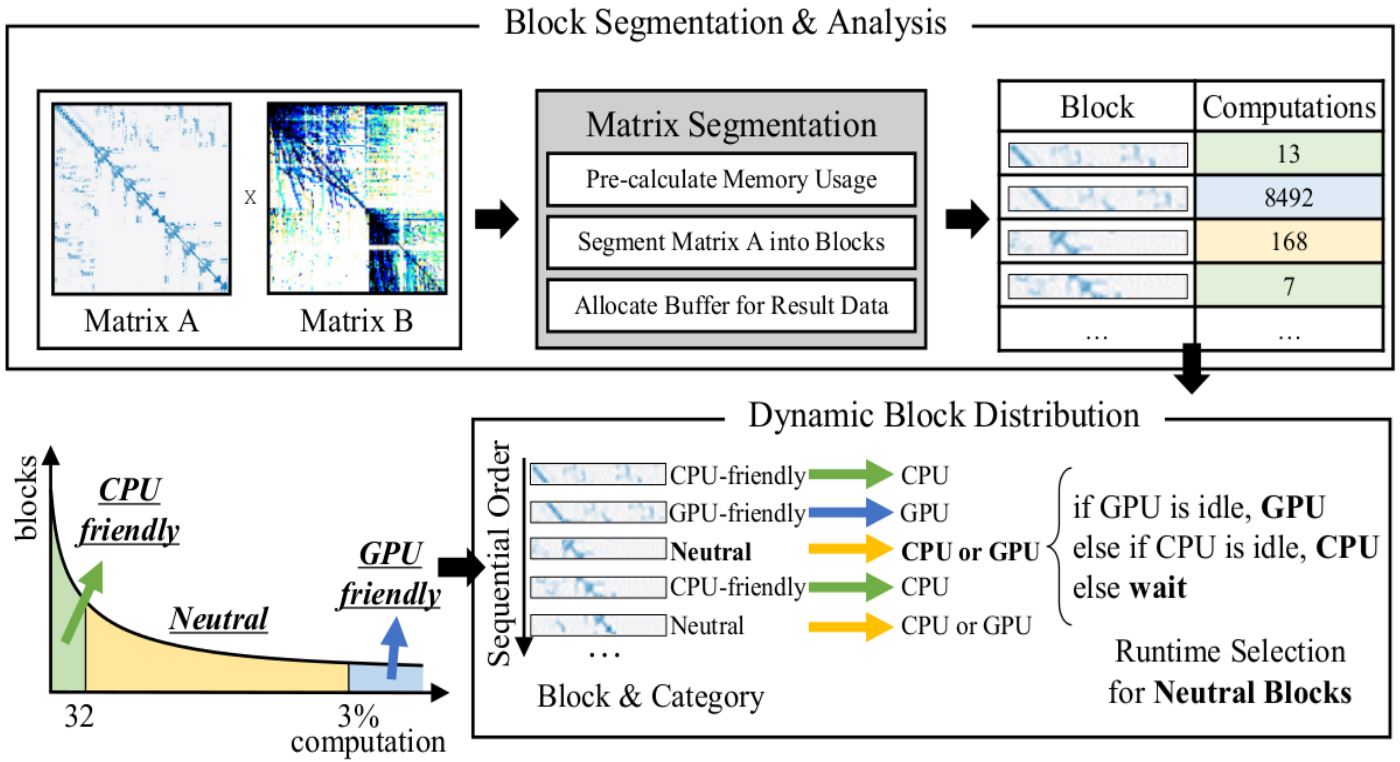
- **Automatic identification of GPU kernel implementation availability**
  - **Pros**: code portability and transparency
  - **Cons**: might not use the most fit processor
  - [Xekalaki@VLDB'22]

# Placement Decision: automatic (example)

**e.g. automatic processor affinity classification**: DBD [Park et al.@ICDE'23]

- Key principles
  - Dynamic Block Distributor (DBD) partition and distribution of SpGEMM on CPU-GPU systems
  - Considers block classification and the status of resources (load balance)

- Contributions
  - Automatic classification of blocks at compile time:
    - **CPU-friendly**: blocks with fewer computations than the warp size
    - **GPU-friendly**: Blocks with most amount of computation (3% of all blocks)
    - **Neutral**: uncategorized blocks can be placed on either CPU or GPU based on load balance



# Placement Decision: semi-automatic (overview)

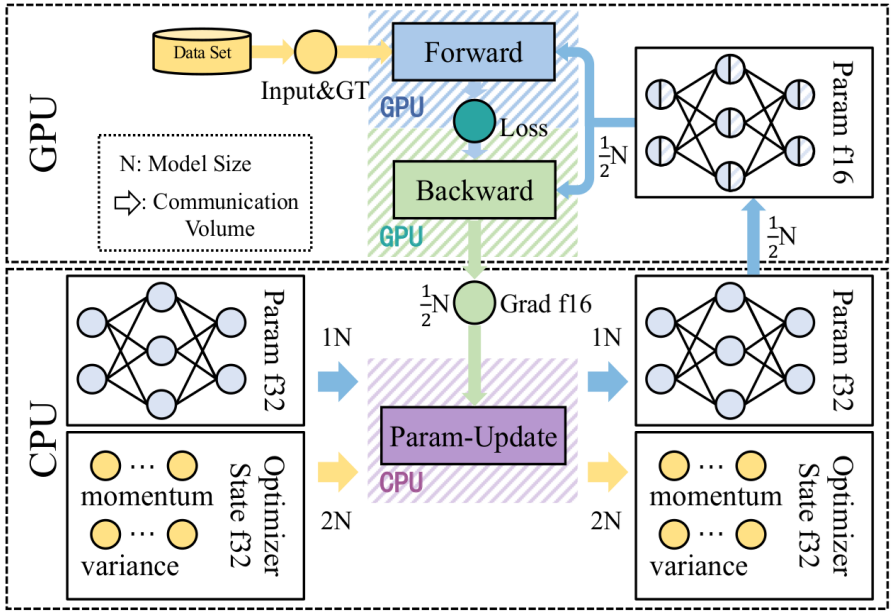
- **Placement considering the specification of**:
  - **Function parameters**
    - **CoTrain**: tasks in Deep Learning (DL) training
    - **GaccO**: dominant applications (OLTP transactions)
    - **[Gowanlock et al.@DaMoN'19]**: task computational complexity
    - **[Ravi@CCGRID'12]**: CPU-GPU job execution time
    - **AHP**: compiler directives to extract tasks to be placed
    - **DAGuE**: GPU kernel implementation for tasks
    - **StarPU, Rectangle method**: scheduling hints (priority tasks, performance model, schedulability criterion)
  - **Data parameters**
    - **Mordred**: heuristics to determine table column segment correlations
    - **PyTorch, Dask**: GPU-based data type to trigger GPU
    - **A&R, [Pirk et al.@VLDBW'12]**: amount of data (bits) to be placed on GPU
    - **SnuCL**: CPU-GPU distance metric to guide data transfers
- **Placement considering workload profiling**
  - **e.g.** Troodon, Caldera, AHP, GreenGPU, [Gowanlock et al.@DaMoN'19], Spark-GPU, MEGHA, Qilin
- **Placement considering training phase**
  - **e.g.** HERO, GHive, READYS, Troodon, Placeto, SKMD, [Kofler et al.@ICS'13], Ocelot+HyPE

# Placement Decision: semi-automatic (example)

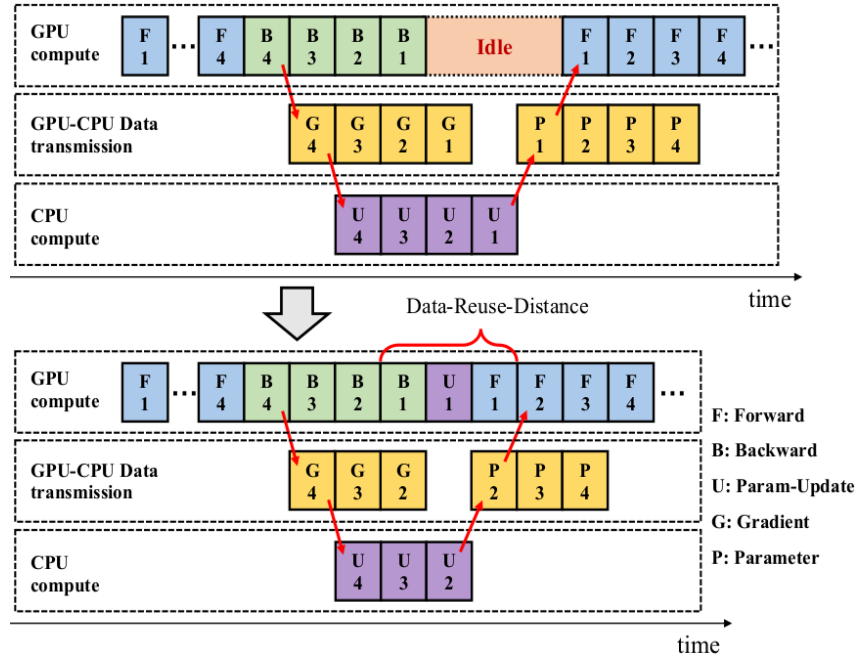
**e.g. specification of function parameters**: CoTrain [Li et al.@ICPP'23]

- Key principle
  - CPU-GPU task scheduling framework for Deep Learning training
- Contribution
  - Manual placement of *Forward* and *Backward* tasks on GPU and automatic placement of *Parameter-Update* tasks on both CPU and GPU based on data locality and load balance

Manual task placement




Automatic task placement



# Placement Decision: manual (overview/example)

- Placement considering the manual specification of:
  - Data partition and distribution
    - **e.g.** [Kroviakov et al.@DAMON'24], HyGraph, [Cheng et al.@DaMoN'15], Extended TOTEM
  - Tasks to be placed on each processor
    - **e.g.** Parla, MCL Schedulers, Crystal, PyTorch, GFlink, Ray, PyCOMPSs*, Dask, TensorFlow, Legion

**e.g. manual specification of task placement and cache parameters in PyCOMPSs**

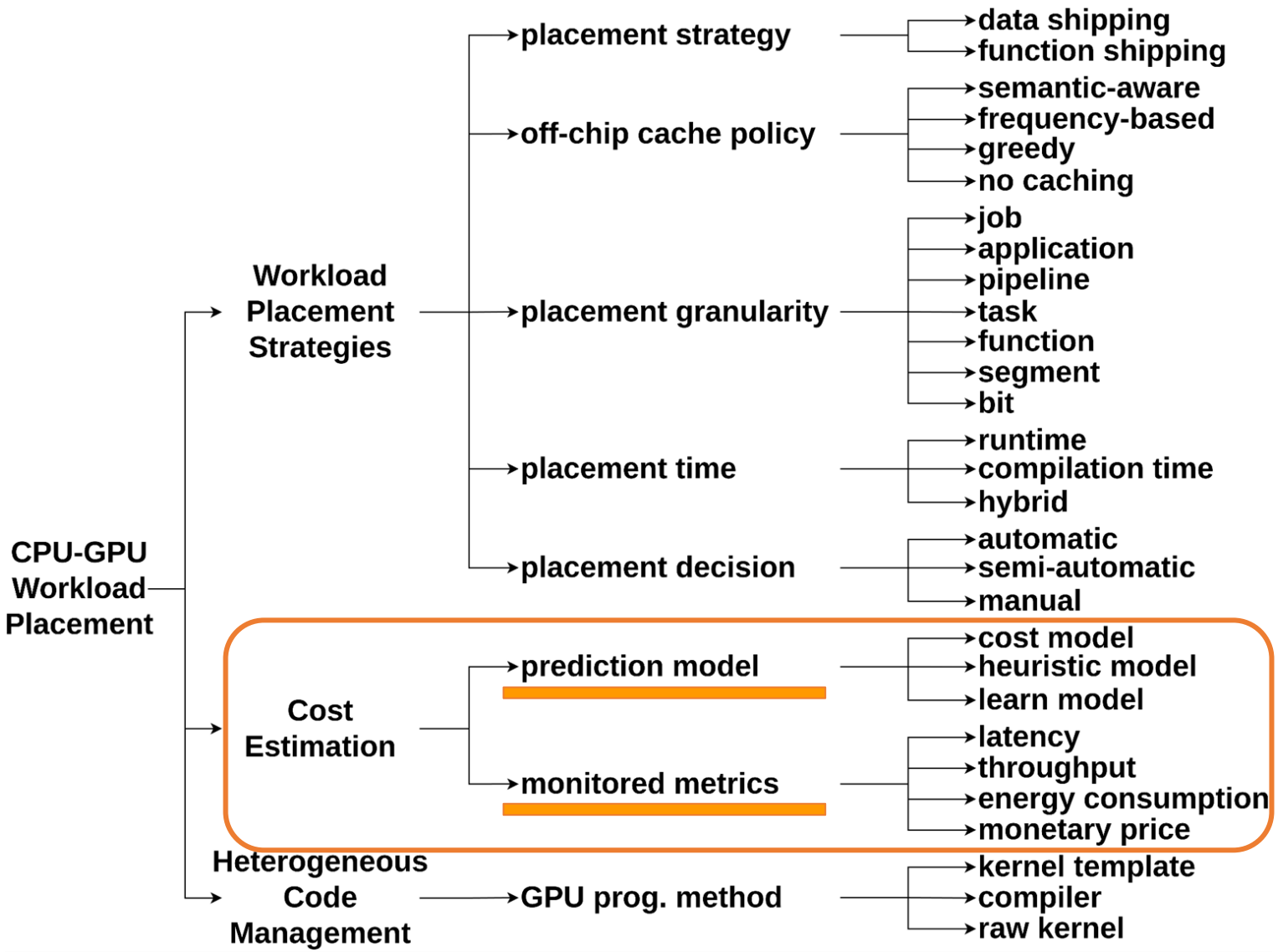
```
--python_worker_cache=true:4GB
@constraint(processors=[{'ProcessorType':'GPU'}])
@task(x={Cache: True}, y={Cache: False},
      returns=cp.array, cache_returns=True)
def mytask(x, y):
  ....
```

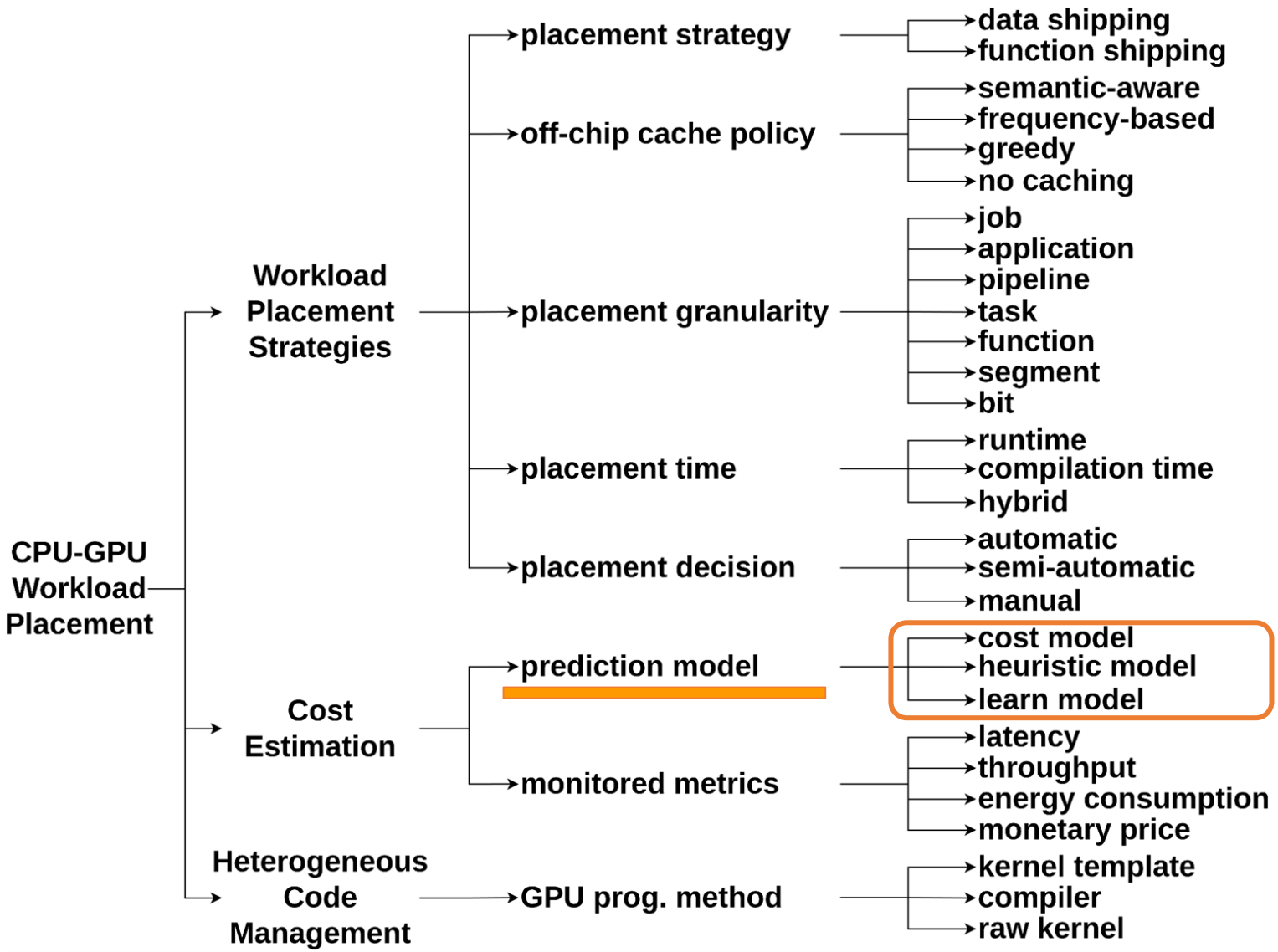
cache setup flag

CuPy array

*PyCOMPSs v. 3.2+ also enables programmers to specify CPU/GPU cache (cache size and data to cache). More info *here*

# Estimating costs for placement decisions


CPU-GPU Workload Placement
Workload Placement Strategies
placement strategy
data shipping
function shipping
off-chip cache policy
semantic-aware
frequency-based
greedy
no caching
placement granularity
job
application
pipeline
task
function
segment
bit
placement time
runtime
compilation time
hybrid
placement decision
automatic
semi-automatic
manual
Cost Estimation
prediction model
cost model
heuristic model
learn model
monitored metrics
latency
throughput
energy consumption
monetary price
Heterogeneous Code Management
GPU prog. method
kernel template
compiler
raw kernel


CPU-GPU Workload Placement
Workload Placement Strategies
placement strategy
data shipping
function shipping
off-chip cache policy
semantic-aware
frequency-based
greedy
no caching
placement granularity
job
application
pipeline
task
function
segment
bit
placement time
runtime
compilation time
hybrid
placement decision
automatic
semi-automatic
manual
Cost Estimation
prediction model
cost model
heuristic model
learn model
monitored metrics
latency
throughput
energy consumption
monetary price
Heterogeneous Code Management
GPU prog. method
kernel template
compiler
raw kernel

# Placement Prediction Model: overview

- Defines the nature of the model used to estimate performance metrics to support placement decisions
- Cost model (white box)
  - Builds a quantitative performance formulation considering hardware/algorithm internals
  - **Pros**: near-optimal placement decisions, fast estimations
  - **Cons**: ad-hoc design (complex to generalize), complex implementation, deep knowledge about the hardware/algorithm
- Heuristic model
  - Devises predefined rules, typically implemented as optimization algorithms
  - **Pros**: flexible design (generalizable)
  - **Cons**: it might result in sub-optimal placement
- Learn model (black box)
  - Builds a data-driven model to predict performance metrics based on historical data
  - **Pros**: adaptive and able to identify complex patterns, no prior knowledge about the hardware/algorithm
  - **Cons**: expensive training (e.g. training data size, training time, energy footprint)

# Placement Prediction Model: cost model (overview)

- Estimates task execution time to:
  - Support task (query operator) placement decisions
    - Crystal, Compressed Crystal, GDB
  - Take caching decisions
    - Mordred
- Estimates data partitioning for:
  - Functions (database primitives)
    - [Gowanlock et al.@DaMoN'19]
  - Deep Learning training tasks
    - CoTrain
  - Distributed scientific application tasks
    - Sigmoid, MC-MOC, Heterogeneous Linpack
  - Matrix multiplication tasks
    - [Clarke et al.@Euro-Par'12]
  - Graph processing application tasks
    - HyGraph

# Placement Prediction Model: cost model (example 1)

**e.g. estimates task execution time**: Mordred [Yogatama et al.@PVLDB'22]

- Key principles
  - Developed on top of the Crystal library (leveraging cost models for *selection*, *projection*, and *hash join*)
  - Assumes that memory bandwidth is saturated (derives cost models from the CPU/GPU memory traffic)
- Contributions
  - Extends Crystal, modeling *filter*, *probe*, *bus data transfer*, *materialization*, and *merge*
  - Uses the estimated operator execution times to compute weights for the proposed semantic-aware cache

**Algorithm 1: Update the *weighted frequency counter* for segment *S***

```
1 UpdateWeightedFreqCounter(segment S)
    # estimate query runtime when S is not cached.
2   RT_uncached = estimateQueryRuntime(cached_segments \ S)
    # estimate query runtime when S and segments correlated with S
      are cached.
3   RT_cached = estimateQueryRuntime(cached_segments ∪ S ∪
      correlated_segments)
4   weight = RT_uncached − RT_cached
5   S.weighted_freq_counter += weight
6   for C in correlated_segments do
        # evenly distribute weight to all segments correlated with S
7       C.weighted_freq_counter += weight / |correlated_segments|
```

obtained with cost models on each operator

**e.g. selection:**

```
SELECT y FROM R WHERE y < v;
```

$$runtime = \frac{4 \times N}{B_r} + \frac{4 \times \sigma \times N}{B_w}$$

- 4-byte floats
- #transferred floats
- predicate selectivity
- memory write bandwidth
- memory read bandwidth

$$data\ transfer\ time = \frac{4 \times N}{B_{pci}}$$

- 4-byte floats
- #transferred floats
- PCIe bus bandwidth

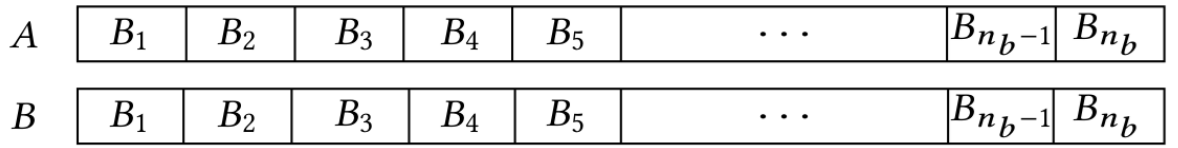
# Placement Prediction Model: cost model (example 2)

**e.g. estimates data partitioning**: [Gowanlock et al.@DaMoN'19]

- Key principles
  - Accelerates memory-bound database primitives on GPU (typically suitable for CPUs)
    - Target primitives: *batched predecessor searches*, *multiway merging*, and *partitioning*
  - Processes a primitive on both CPU and GPU over different data partitions to minimize load imbalance
- Contributions
  - Proposes cost models to determine the ideal CPU-GPU data partitioning for each primitive
  - Assumes that CPU-GPU and GPU-CPU data transfers are overlapped: $\max(2n, n) = 2n \rightarrow |A| = |B| = n$

CPU-GPU data transf. ← $2n$ | $n$ → GPU-CPU data transf.

**e.g. batched predecessor searches**
for each query element, finds the largest key index, such that the key is smaller or equal to the query




→ set of keys

→ set of queries

$$T^{CPU} = \frac{3n}{\alpha} \rightarrow T^{CPU} = \frac{3f}{\alpha}$$

$$T^{GPU} = \frac{2n}{\beta} \rightarrow T^{GPU} = \frac{2(1-f)}{\beta}$$

$$T^{CPU} = T^{GPU} \rightarrow \frac{3f}{\alpha} = \frac{2(1-f)}{\beta} \rightarrow f = \frac{2\alpha}{2\alpha + 3\beta}$$

fraction of *n* elements computed on GPU

main memory bandwidth

PCIe bandwidth

# Placement Prediction Model: heuristic model (overview/examples 1/3)

- **Heuristics to improve data locality and load balance**
  - **Locality-aware data partitioning**
    - **CoTrain**: places data partitions for tasks (*param-update*) on GPU based on the *data-reuse-distance*
    - **HetExchange**: tracks data locality on each instance of its query pipeline execution
  - **Processor rank based on load balance and data locality scores**
    - **MCL Schedulers**: uses the amount of data reuse on tasks and #data copies to find local data on a processor
    - **Parla**: uses the amount of local data and device load (on current and dependent tasks)
- **Heuristics to improve data locality**
  - **Semantic-aware cache**
    - **NeutronOrch**: computes on CPU only vertice embeddings in CPU memory (bottom layer of GNN training)
    - **HetCache**: caches densely accessed data (pages) in GPU (interconnect-bandwidth or storage-bandwidth bound queries) and sparsely accessed data for storage-bandwidth bound queries in CPU
    - **Mordred**: set of heuristics per task (query operator) to define correlated segments for frequency weight counter cache
  - **Task placement based on processor distance**
    - **CoTrain, TensorFlow, SnuCL**: places tasks according to the lowest processor distance

# Placement Prediction Model: heuristic model (overview/examples 2/3)

- **Heuristics to improve load balance**
  - **Data partitioning**
    - **GreenGPU**: partitions data to make CPU-GPU tasks finish at the same time (minimize *idling energy*)
  - **On-demand task placement based on processor status**
    - **FusionFlow, CGgraph**: opportunistically places CPU/GPU tasks on GPU/CPU when it becomes idle
    - **Caldera**: runs applications (OLAP queries) on CPU and GPU considering resource status (processor/memory utilization)
    - **SCCG**: migrates tasks to GPU/CPU if GPU memory usage is low/high (empty/full input buffer)
    - **Legion, StarPU**: queue-based scheduling with support to task stealing (processors "pull" tasks)
- **Heuristics to overcome GPU memory capacity constraints**
    - **RateupDB**: processes applications (OLAP queries) on a partition of data on CPU, if GPU memory size is not enough
    - **ZeRO-Offload**: uses mixed precision (CPU stores fp32 model states+fp16 gradients and GPU stores fp16 param.) in heterogeneous Deep Learning training
    - **A&R, [Pirk@VLDBW'12]**: processes GPU query over lossy compressed data using bitwise decomposition

# Placement Prediction Model: heuristic model (overview/examples 3/3)

- **Heuristics based on processor affinity classification**
  - **Processor affinity on data**
    - **DBD**: defines CPU-friendly, GPU-friendly and neutral blocks (placed according to processors' idle status)
    - **A&R, [Pirk@VLDBW'12]**: places approximation data on GPU and residual data on CPU
    - **[Lutz et al.@SIGMOD'20], [Ravi et al.@ICS'10]**: places grouped/individual chunks of data on GPU/CPU
  - **Processor affinity on function**
    - **GaccO**: places frequent/non-frequent applications (OLTP transactions) on GPU/CPU
    - **gSWORD, ZeRO-Offload**: places compute-intensive/memory-intensive tasks on GPU/CPU
    - **RateupDB, Caldera**: places applications (OLTP queries on CPU and OLAP queries on GPU by default)
    - **PyTorch**: places tasks (tensor operators) on GPU by default
    - **Spark-GPU**: places tasks (Spark operations) on GPU if operation is compute-intensive
    - **SCCG**: places parser, builder, and filter pipeline stages on CPU and aggregator stage on GPU by default
    - **[Mayer et al.@DIDL'17], SABER, MEGHA**: places tasks on the processor that performs best (lowest throughput/execution time)
- **Heuristic to reduce energy consumption**
    - **[Cheng et al.@DaMoN'15]**: CPUs are more energy-efficient on simple tasks (selection and aggregation operators); GPUs are more energy-efficient more complex tasks (sort and hash join operators)

# Placement Prediction Model: learn model (overview)

- **Numerical analysis over past data**
  - **Pros**: simple model maintenance, fast estimations
  - **Cons**: data size and execution time are assumed to scale linearly, limited applicability
  - **e.g. -Estimates task exec. time with interpolation**: HERO, Robust CoGaDB, [Karnagel et al.@EDBTW'15], Ocelot+HyPE, AHP
    - **-Estimates task exec. time for data partitioning with hyperplane fitting**: [Kroviakov et al.@DAMON'24]
    - **-Estimates data partitioning with curve fitting**: Rectangle Method, Qilin
    - **-Estimates application energy consumption with curve fitting**: [Cumming et al.@SC'14]
- **Supervised learning**
  - **Pros**: flexible model applicability, intuitive models, popular algorithms
  - **Cons**: limited applicability (limited predictions in very complex patterns)
  - **e.g.** GHive, GAP, Troodon, [Kerr et al.@ASPLOS'10], [Li et al.@J. Syst. Arch'21], PLB-HAC, [Wen and O'Boyle@GPGPU-PPoPP'17], SKMD, [Grewe and O'Boyle@CC'11], [Kofler et al.@ICS'13]
- **Reinforcement Learning (RL)**
  - **Pros**: self-adaptive, learns patterns in very complex scenarios (e.g. task scheduling)
  - **Cons**: expensive training time, complex models and algorithms
  - **e.g.** READYS, ADTS, Placeto, Hierarchical Planner, LDWP-IAWP, Symphony

# Placement Prediction Model: learn model (example 1)

**e.g. numerical analysis:** HyPE [Breß@Inf. Syst'13]; HOP [Karnagel et al.@Datenbank-Spektrum'14]

- **Execution time prediction to support task (query operator) placement using interpolation**
  - **HyPE optimizer**
    - **Key principles**: spline interpolation (one approximation function for all data points), depends on initial hardware calibration
    - **e.g.** Robust CoGaDB, Ocelot+HyPE
  - **HOP optimizer**
    - **Key principles**: linear interpolation (weighted average of the two neighboring data points), no hardware calibration required (initial placement heuristics)
    - **e.g.** HERO, [Karnagel et al.@EDBTW'15]

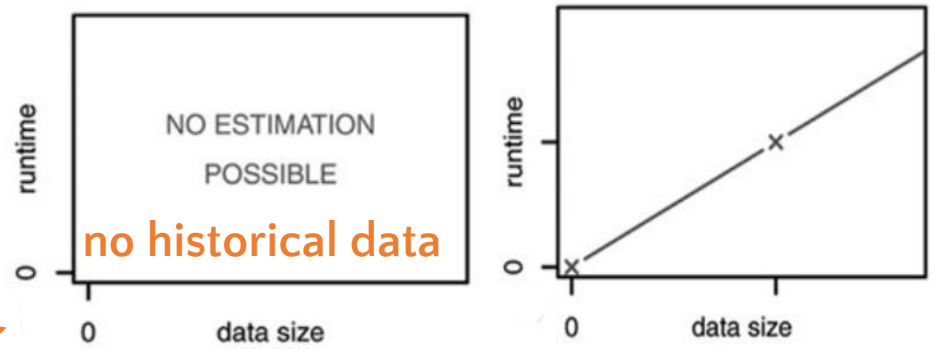
cost model

constant (hardware dependent)

$$transfer\ estimate = \sum \frac{amount\ of\ data}{transfer\ bandwidth\ for\ data}$$

$$exec.\ estimate = latency + transfer\ input + computation + transfer\ output$$

estimated with interpolation



# Placement Prediction Model: learn model (examples)

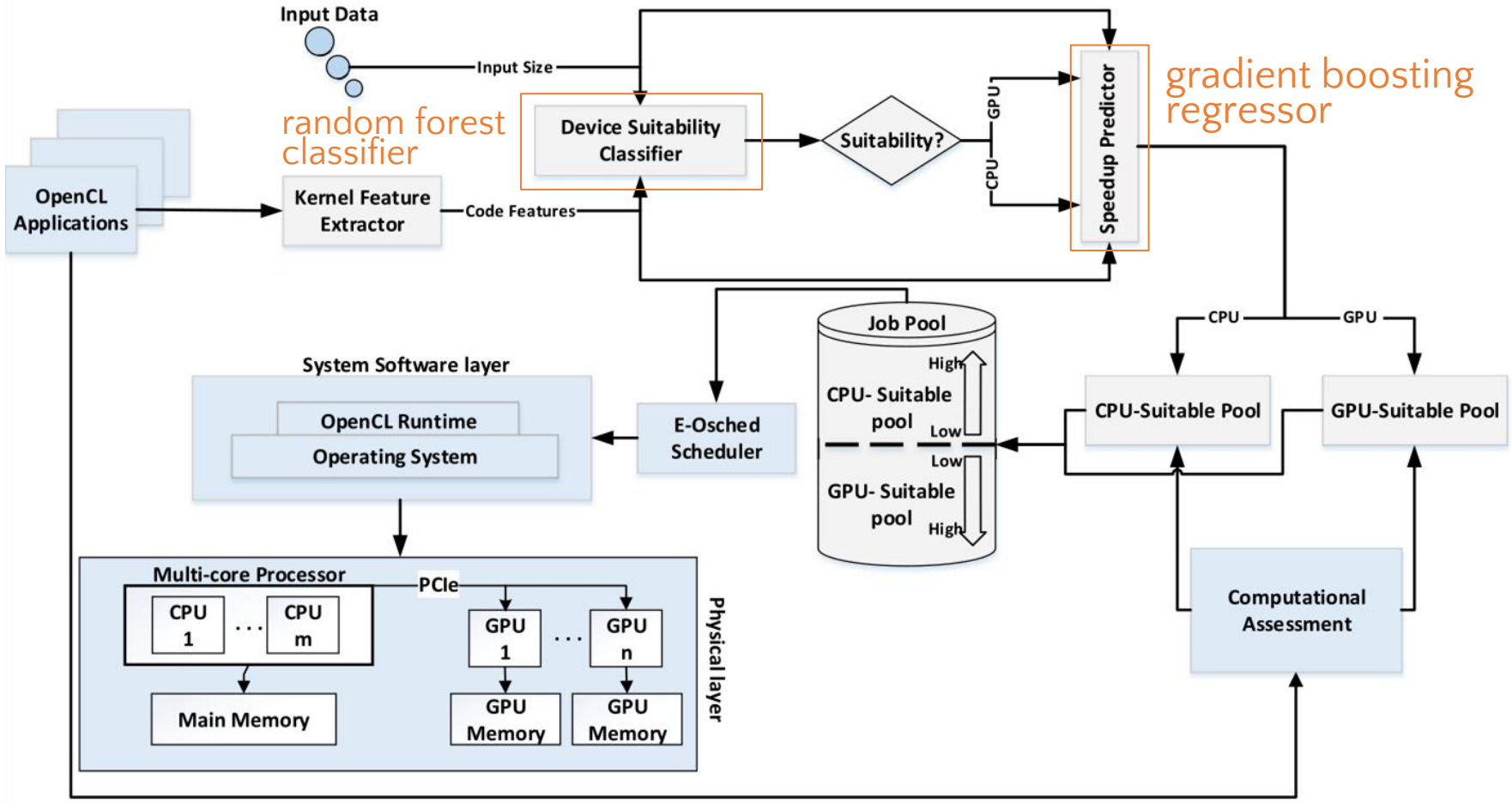
**supervised learning** (overview examples)

- Single model
  - **Execution time prediction for function placement using regression**
    - GHive, GAP
  - **Execution time prediction for data partitioning using regression**
    - [Li et al.@J. Syst. Arch'21], PLB-HAC
  - **Job power consumption prediction using regression**
    - [Sîrbu and Babaoglu@Euro-Par'16]
  - **Data partitioning prediction using**
    - **Artificial Neural Networks (ANN)**
      - [Kofler et al.@ICS'13]
    - **Support Vector Machine (SVM)**
      - [Grewe and O'Boyle@CC'11], [Kerr et al.@ASPLOS'10]
    - **Decision Tree**
      - SKMD
  - **Task processor affinity classifier using Decision Tree**
    - [Wen and O'Boyle@GPGPU-PPoPP'17]
- Multiple models
  - **Processor affinity classifier and speedup predictions using Random Forest + Gradient Boosting Regressor**
    - Troodon

# Placement Prediction Model: learn model (example 2)

**e.g. processor affinity and speedup predictions w/ supervised learning**: Troodon [Khalid et al.@JPDC'19]

- Key principle
  - Places a set of OpenCL applications (job pool) keeping load balance between CPU and GPU
- Contributions
  - A 2-step learn model:
  1. **Random Forest classif.**: CPU-GPU application affinity classifier based on past executions
  2. **Gradient Boosting regres.**: speedup predictor used to sort CPU-GPU job pools enabling processors to share applications with minimal performance degradation



# Placement Prediction Model: learn model (examples)

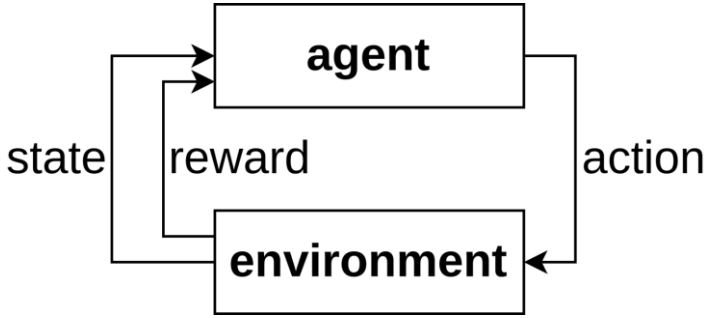
**reinforcement learning** (overview examples)

- **Task group placement for massive distributed neural networks**
  - **Hierarchical Planner**: Learns how to group tasks and predicts placements with policy gradient algorithm and Recurrent Neural Network (RNN) for policy
  - **Placeto**: predicts task group placements with policy gradient algorithm and Multilayer Perceptron (MLP) for policy
- **Task scheduling** (implicit task placement)
  - **Static task scheduling** (tasks (incl. dependencies) are known in advance)
    - **LDWP-IAWP**: Learns node-level task scheduling and processor-level function scheduling using Deep Q-network with Neural Collaborative Filtering (NCF)
    - **Symphony**: Task scheduling with Bayesian Model + RNN (Partially Observable Markov Decision Processes (POMDP))
    - **ADTS**: Task scheduling with REINFORCE algorithm using Monte-Carlo Policy-Gradient Control for policy
  - **Dynamic task scheduling** (tasks (incl. dependencies) are revealed at runtime)
    - **READYS**: Task scheduling with Actor-Critic (A2C) algorithm using Graph Convolutional Networks (GCN) for policy

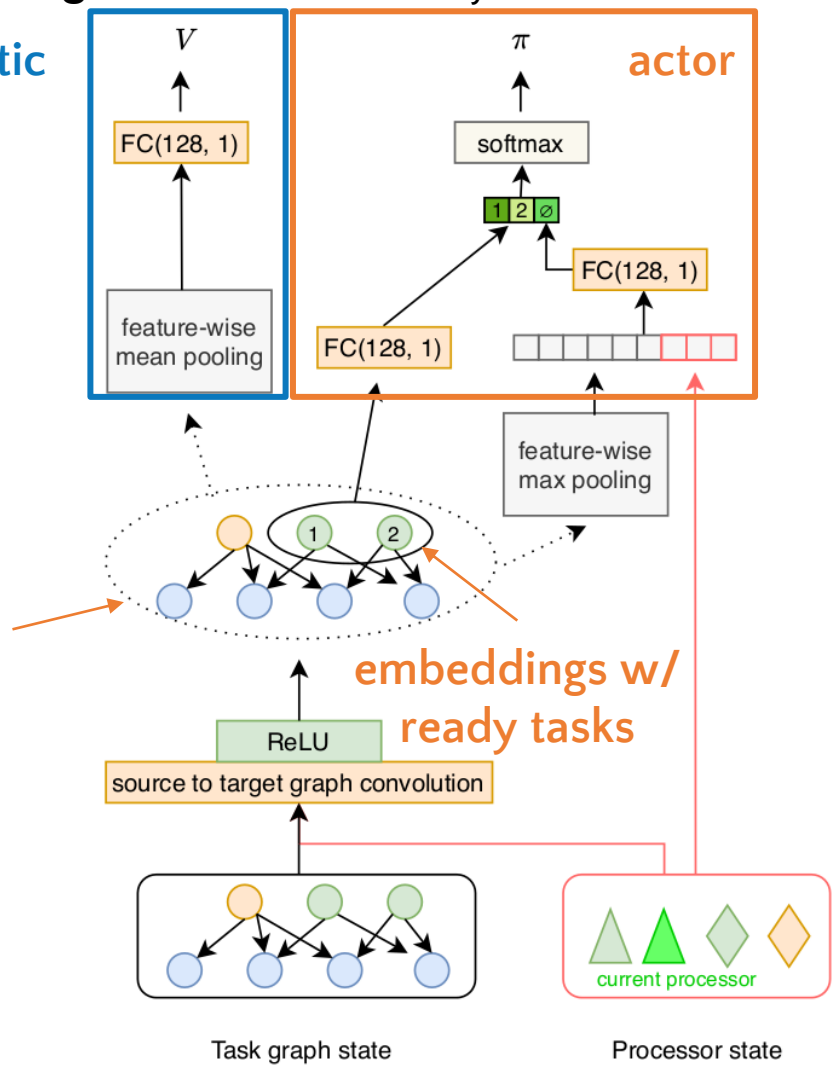
# Placement Prediction Model: learn model (example 3)

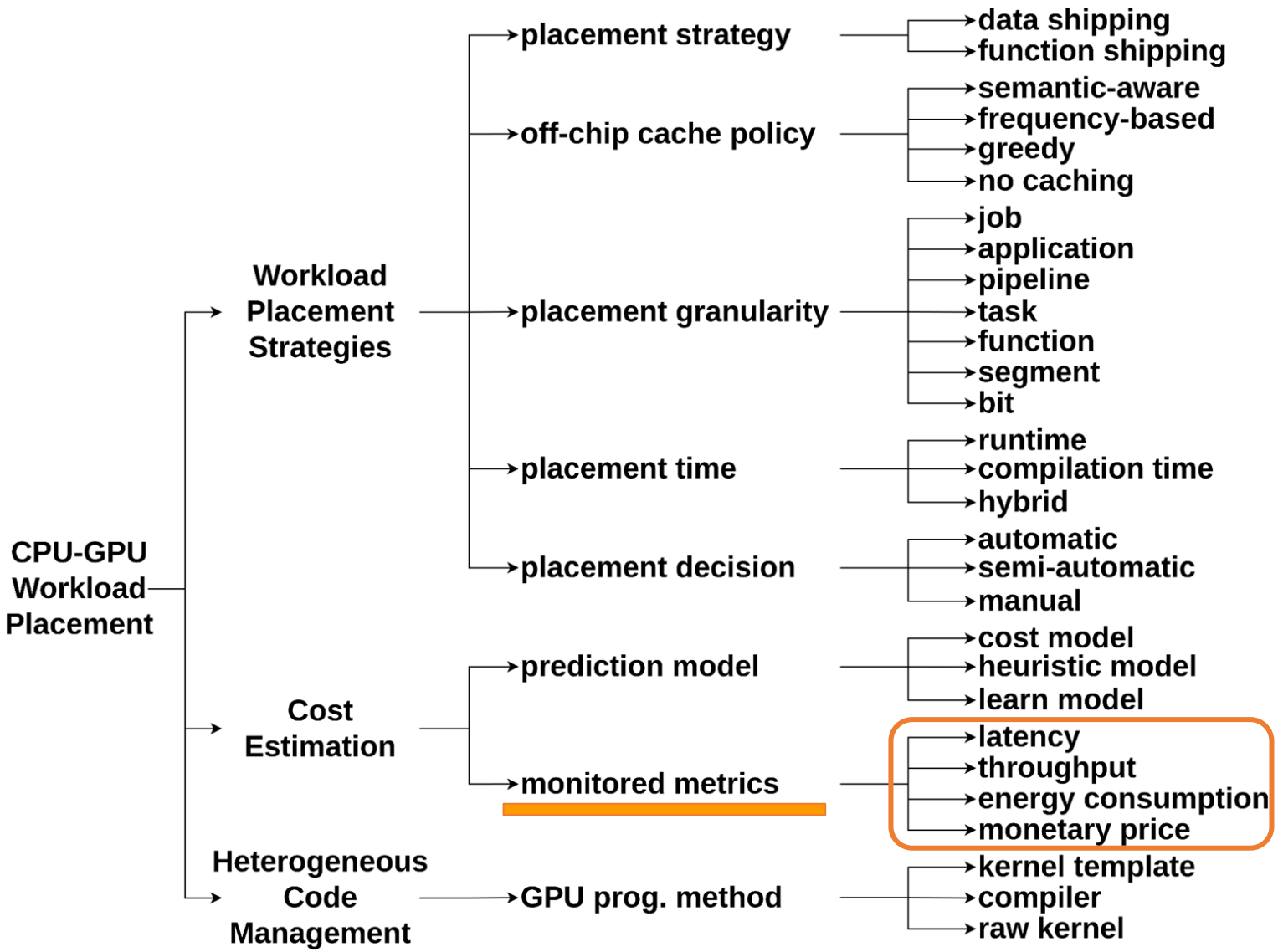
**e.g. reinforcement learning**: READYS [Grinsztajn@CLUSTER'21]

- Key principles
  - Dynamic DAG scheduling in heterogeneous system to minimize the makespan (completion time of the last task in the DAG)
    - NP-hard combinatorial optimization problem
  - Problem modeled as a Markov Decision Process




- Contributions
  - Actor-Critic (A2C) RL algorithm composed of two Graph Convolutional Networks (GCN)
    - Actor: policy network π that receives a state as input and predicts a probability distribution over the possible actions ("decisions")
    - Critic: value network to estimate the value of a state ("evaluations")




CPU-GPU Workload Placement
Workload Placement Strategies
placement strategy
data shipping
function shipping
off-chip cache policy
semantic-aware
frequency-based
greedy
no caching
placement granularity
job
application
pipeline
task
function
segment
bit
placement time
runtime
compilation time
hybrid
placement decision
automatic
semi-automatic
manual
Cost Estimation
prediction model
cost model
heuristic model
learn model
monitored metrics
latency
throughput
energy consumption
monetary price
Heterogeneous Code Management
GPU prog. method
kernel template
compiler
raw kernel

# Monitored Metrics: overview

- Defines the target metrics used to guide and/or compare CPU-GPU placement decisions
- Latency
  - Monitors the workload execution time (total or specific portions) on each processor
  - **Pros**: provides insights about placement performance, popular metric, easy to measure and interpret
  - **Cons**: limited scope
- Throughput
  - Monitors the volume of data processed per unit of time (e.g. GB/s) on each processor
  - **Pros**: provides insights about the utilization of resources (e.g. processor, memory, storage, interconnects)
  - **Cons**: complex analysis (it must consider multiple resources)
- Energy consumption
  - Monitors the energy required to run a workload on each processor
  - **Pros**: overview about the energy footprint, reveals energy saving opportunities
  - **Cons**: complex to measure (requires external power meters or specialized software)
- Monetary price
  - Monitors the cost-efficiency of each processor
  - **Pros**: offers a notion about the economical impacts of opting for a processor
  - **Cons**: complex to measure (dynamic prices and several hardware models/settings)

# Monitored Metrics: latency (overview/examples)

- For function placement
  - Chooses the processor with smallest execution time
    - **e.g.** LDWP-IAWP, Troodon, Spark-GPU, [Mayer et al.@DIDL'17], AHP, MEGHA, GDB, [Ravi et al.@CCGRID'12]
- For data partitioning
  - Partition ratio that makes concurrent CPU-GPU tasks finish at the same time
    - CoTrain, PLB-HAC, [Gowanlock et al.@DaMoN'19], HyGraph, [Clarke et al.@Euro-Par'12], MC-MOC, GreenGPU, Qilin
  - Partition that results in minimal penalty (data transfer and merge times) for concurrent CPU-GPU tasks
    - [Li et al.@J. Syst. Arch'21], SKMD

# Monitored Metrics: throughput (overview/examples)

- **For function placement**
  - **LDWP-IAWP**: Function (subtasks) speedup prediction using GPU load/store throughput as features. Places functions with smallest predicted speedups on CPUs
  - **SABER**: places applications (streaming queries) on the processor with highest measured throughput
  - **SCCG**: places aggregator tasks on both CPU and GPU based on memory usage to maximize overall throughput
- **For data partitioning**
  - **[Lutz et al.@SIGMOD'20]**: places grouped/individual chunks of data on GPU/CPU to minimize the estimated throughput execution skew between CPU and GPU tasks
  - **Heterogeneous Linpack**: partitions CPU-GPU task data proportionally to the computed throughput from the previous execution
- **For taking caching decisions**
  - **HetCache**: infers the per device task (query) processing throughput to determine if a query is interconnect-bandwidth or storage-bandwidth bound to support the semantic-aware cache

# Monitored Metrics: energy consumption (overview/examples)

- **Measurements using external power meter (++precision --practical)**
  - **Latency-driven data partitioning to minimize minimize *idling energy***
    - GreenGPU, Qilin
  - **Empirical energy consumption analysis**
    - **[Cheng et al.@DaMoN'15]**: CPU/GPU is more energy-efficient on simple/complex tasks (query operators)
    - **Extended TOTEM**: proposes guidelines for energy-efficient CPU-GPU graph processing tasks
      - GPU-only, if graph fits in GPU memory (GPU draws more power, but is faster than CPU)
      - Single/Multi-node CPU-GPU, for larger graphs, if energy/performance is the main concert
- **Measurements using software tools based on internal sensors (++precision -practical)**
    - **Sigmoid:** proposes a power monitoring tool using RAPL for CPU and NVIDIA NVML for GPU
- **Data-driven approaches (-precision ++practical)**
    - **[Sîrbu and Babaoglu@Euro-Par'16]**: predicts job power consumption using past execution time as a feature
    - **[Cumming et al.@SC'14]**: devises a methodology to estimate energy consumption using only execution time, without the need of measuring power or energy consumptions

# Monitored Metrics: monetary price (overview/example)

Crystal [Shanbhag@SIGMOD'20]

- Key principle
  - Comparison between CPU and GPU regarding purchase and renting costs
- Contributions
  - GPUs have superior performance (25x) than CPUs for data analytics
  - However, GPUs have higher monetary costs (6x) than CPUs
  - Factor of 4x improvement in cost-effectiveness of GPU over CPU
    - A similar cost-effectiveness ratio should be obtained across different CPU and GPU models

| | Purchase Cost | Renting Cost |
|---|---|---|
| CPU | $2-5K | $0.504 per hour |
| GPU | $CPU + 8.5K | $3.06 per hour |

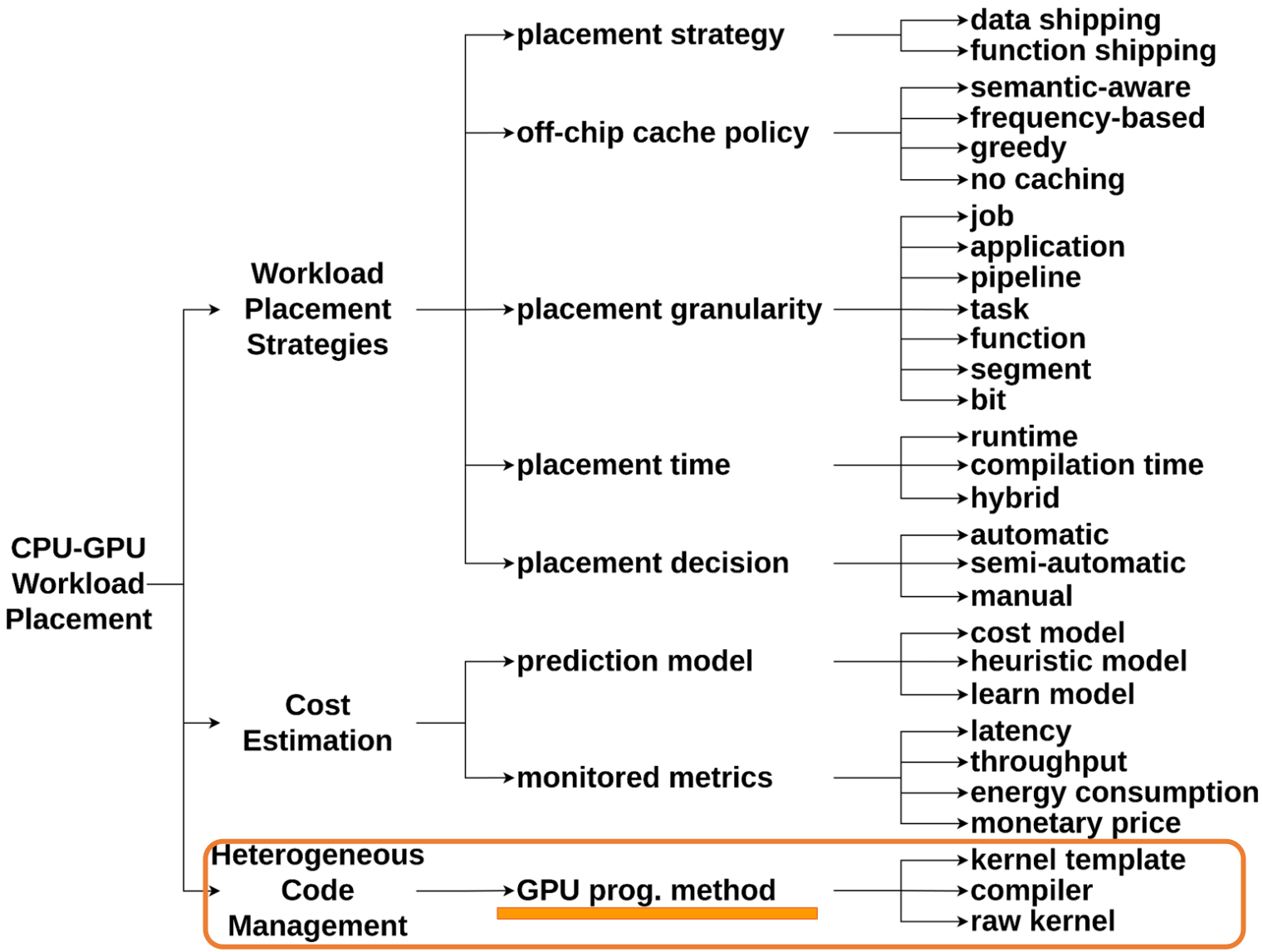
GPU/CPU cost ratio < 6x

GPU/CPU cost ratio = 6x

***Setup***

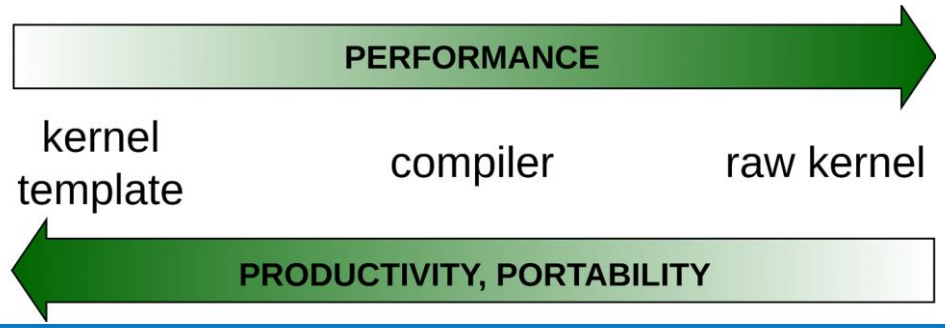
*- Skylake CPU (8 cores)*
*- 1xNVIDIA V100*
*- rental values: AWS EC2*

# Managing heterogeneous CPU-GPU code


CPU-GPU Workload Placement
Workload Placement Strategies
placement strategy
data shipping
function shipping
off-chip cache policy
semantic-aware
frequency-based
greedy
no caching
placement granularity
job
application
pipeline
task
function
segment
bit
placement time
runtime
compilation time
hybrid
placement decision
automatic
semi-automatic
manual
Cost Estimation
prediction model
cost model
heuristic model
learn model
monitored metrics
latency
throughput
energy consumption
monetary price
Heterogeneous Code Management
GPU prog. method
kernel template
compiler
raw kernel

# Heterogeneous Code Management: overview

- Defines the programming abstractions and coding paradigms to enable workload execution on CPU and GPU
- Kernel template
  - Offers pre-defined kernels via public libraries or frameworks
  - **Pros**: high productivity (easy to use), high portability (flexible design, multi-platform w/ minimal code change)
  - **Cons**: although highly optimized, it lacks capability for performance tuning
- Compiler
  - Enables the implementation of UDF kernels using high-level sequential code
  - **Pros and Cons**: balanced level of performance, productivity, and portability
- Raw kernel
  - Allows users to implement their own kernels
  - **Pros**: high performance opportunities (low-level fine tuning)
  - **Cons**: low productivity (complex implementation), low portability (ad-hoc design)

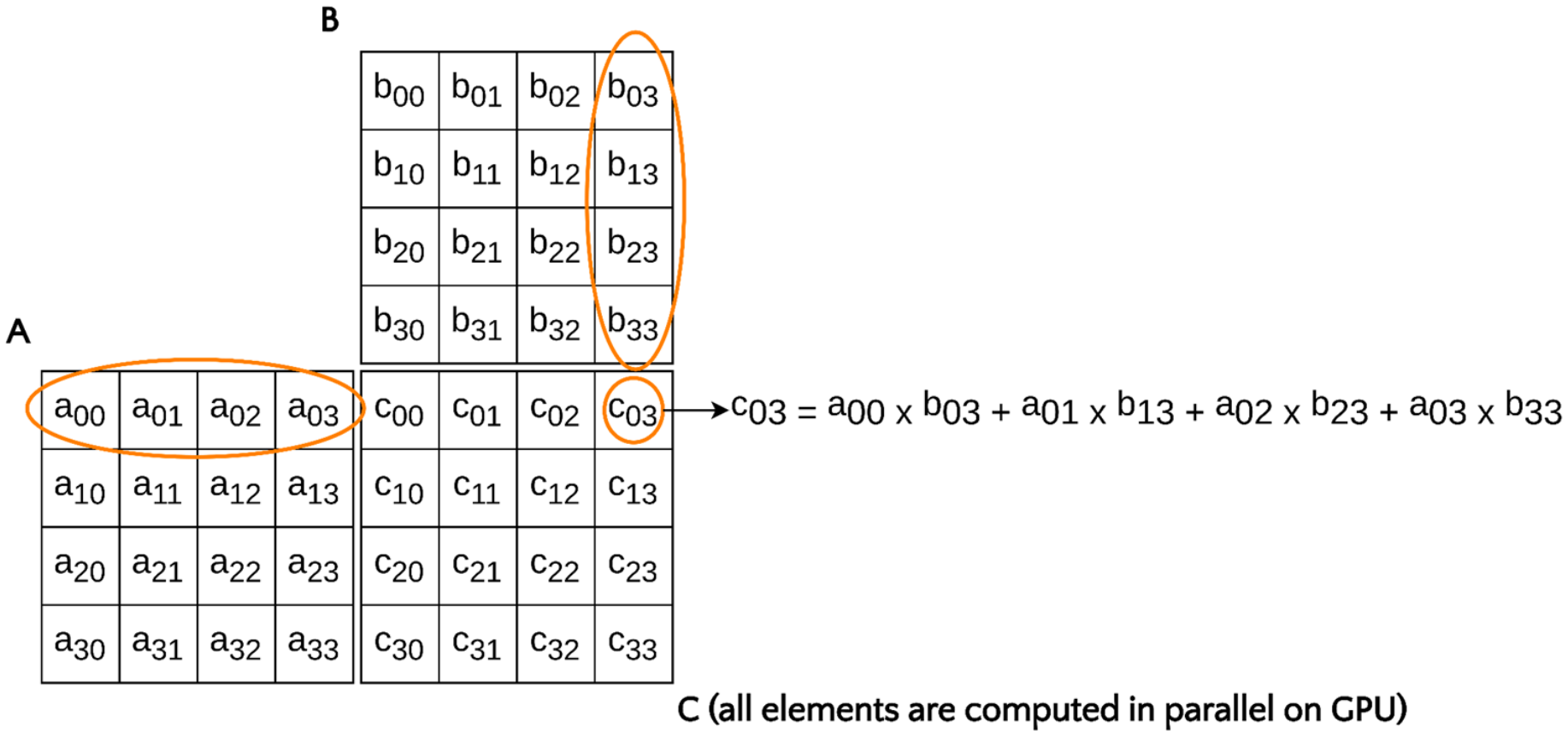


# Heterogeneous Code Management: kernel template (overview)

| Kernel Template | Field | Examples |
|---|---|---|
| CUDA-X (e.g. cuBLAS, cuDNN) | Data processing, AI, HPC | [Li et al.@J. Syst. Arch'2021], PLB-HAC, [Clarke et al.@Euro-Par'12], DAGuE, StarPU, PyTorch, TensorFlow |
| CuPy (atop CUDA-X) | Data processing, AI, HPC | [Carvalho et al.@EDBT'24], PyCOMPSs+dislib, Parla, Dask |
| RAPIDS (atop CUDA-X) | Data processing, AI, HPC | [Lee and Park@ICDEW'21], Dask |
| Crystal | Query processing | Crystal, Extended Crystal, Mordred |
| Ocelot | Query processing | Ocelot+HyPE, [Kerr et al.@ASPLOS'10] |
| Panda | Query processing | GHive |
| ACML-GPU | Math (mainly linear algebra) | Heterogeneous Linpack |
| MATE-CG | Map-Reduce applications | [Ravi et al.@CCGRID'12] |
| OpenCV | Image processing, computer vision | LDWP-IAWP |
| Thrust | Data structures | [Gowanlock et al.@DaMoN'19] |
| MAGMA | Linear algebra | READYS |
| DGL framework | Graph Neural Network | NeutronOrch |
| NVIDIA DALI | Data pre-processing for Deep Learning | FusionFlow |

# Heterogeneous Code Management: kernel template (code example)

- Matrix Multiplication (Matmul)



# Heterogeneous Code Management: kernel template (code example)

- Heterogeneous CPU-GPU code in PyCOMPSs+*dislib*+CuPy (*src:* Matmul)
  - Native support to NVIDIA and AMD GPUs




CPU task

```
@constraint(computing_units="1")
@task(returns=np.array)
def matmul_cpu(a, b):
    return a @ b
```

GPU task

```
@constraint(processors=[
                {"processorType": "CPU", "computingUnits": "1"},
                {"processorType": "GPU", "computingUnits": "1"},
            ])
@task(returns=np.array)
def matmul_gpu(a, b):
    import cupy as cp

    a_gpu, b_gpu = cp.asarray(a), cp.asarray(b)

    res = cp.asnumpy(cp.matmul(a_gpu, b_gpu))
    del a_gpu, b_gpu
    return res
```

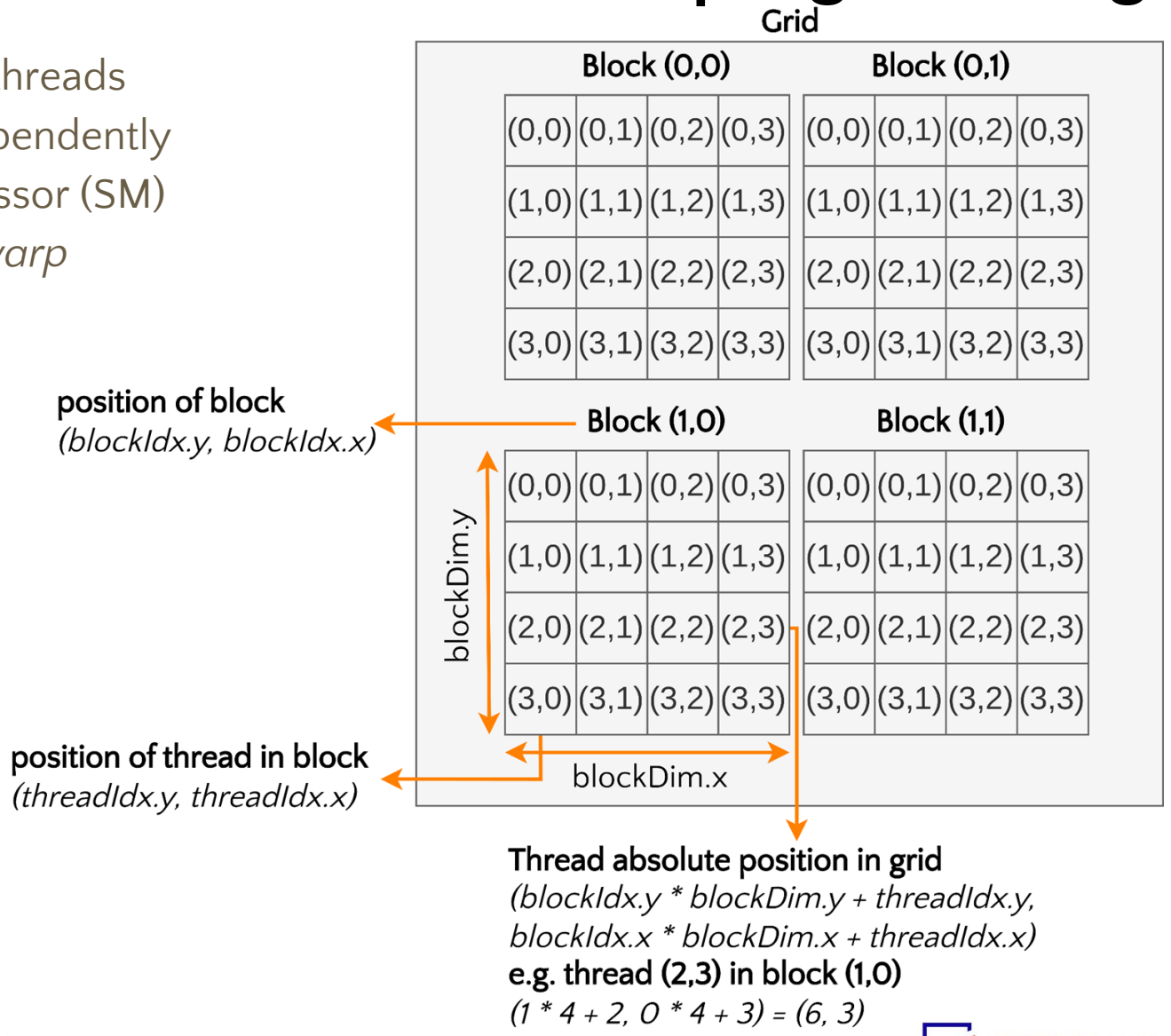
CPU-GPU comm.

GPU kernel call

GPU-CPU comm.

# Heterogeneous Code Management: basics of GPU programming

- A GPU program divides the problem into a grid of threads
- A kernel is the code executed on each thread independently
- A block is scheduled to GPU Streaming Multiprocessor (SM) in groups (of 32 threads for NVIDIA GPUs) called *warp*



# Heterogeneous Code Management: compiler (overview)

- **LLVM**: compiler infrastructure that converts high-level code into hardware-agnostic intermediate representation
  - **Works using LLVM compiler infrastructure to generate GPU code**
    - HetExchange, SKMD, [Wen and O'Boyle@GPGPU-PPoPP'17], [Kerr et al.@ASPLOS'10], [Ravi et al.@ICS'10]
  - **LLVM-based compilers**
    - **Numba**: JIT compiler for Python code
      - Parla, PyCOMPSs
    - **NVIDIA NVCC**: compiler for C, C++, and Fortran code
      - AHP
    - **Intel HDK**: JIT compiler for OLAP queries expressed as relational algebra or SQL
      - [Kroviakov et al.@DAMON'24]
- **Other compilers**
  - **JIT compiler for Java code (TornadoVM)**
    - [Xekalaki@PVLDB'22]
  - **JIT compiler for MATLAB code**
    - MEGHA
  - **Domain-Specific Language (DSL) compilers (e.g. TACO)**
    - MCL Schedulers, Extended MCL

# Heterogeneous Code Management: compiler (code example)

- GPU code in PyCOMPSs+numba (*src: Matmul*)
  - Native support to NVIDIA GPUs



```
@cuda.jit
def matmul(A, B, C):
    """Perform square matrix multiplication of C = A * B
    """
    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
```

GPU kernel

```
TPB = 16
@constraint(processors=[{'ProcessorType':'CPU', 'ComputingUnits':'1'},
                        {'ProcessorType':'GPU', 'ComputingUnits':'1'}])
@task(returns=1)
def do_matmul(a, b, c):
    gpu_a = cuda.to_device(a)
    gpu_b = cuda.to_device(b)
    gpu_c = cuda.to_device(c)

    threadsperblock = (TPB, TPB)
    blockspergrid_x = math.ceil(gpu_c.shape[0] / threadsperblock[0])
    blockspergrid_y = math.ceil(gpu_c.shape[1] / threadsperblock[1])
    blockspergrid = (blockspergrid_x, blockspergrid_y)

    matmul[blockspergrid, threadsperblock](gpu_a, gpu_b, gpu_c)
    c = gpu_c.copy_to_host()
    return c
```

CPU-GPU comm.

kernel setup

GPU kernel call

GPU-CPU comm.

# Heterogeneous Code Management: raw kernel (overview)

- **CUDA (NVIDIA)**: kernel development for NVIDIA GPUs
  - Qilin, GDB, Rectangle method, SnuCL. SCCG, GreenGPU, MC-MOC, Legion, Extended TOTEM, [Cumming et al.@SC'14], [Cheng et al.@DaMoN'15], HyGraph, Spark-GPU, Robust CoGaDB, Caldera, ADTS, GFlink, [Gowanlock et al.@DaMoN'19], Lutz et al.@SIGMOD'20, Crystal, RateupDB, READYS, Compressed Crystal, GAP, GaccO, DBD, CGgraph, gSWORD

- **OpenCL**: write kernels once and run on any device (portable on multiple processors)
  - SnuCL, [Kofler et al.@ICS'13], A&R, [Grewe and O'Boyle@CC'14], [Ocelot+HyPE], [Cheng et al.@DaMoN'15], [Karnagel et al.@EDBTW'15], Spark-GPU, SABER, [Wen and O'Boyle@GPGPU-PPoPP'17], HERO, Troodon, MCL Schedulers, Sigmoid, MCL, DBD

# Heterogeneous Code Management: raw kernel (code example)

- GPU code in CUDA (*src: Matmul*)
  - Native support to NVIDIA GPUs


```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```

GPU kernel

```
#define BLOCK_SIZE 16
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);
    // Allocate C in device memory
    Matrix d_C;
    d_C.width = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);
    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
    // Read C from device memory
    cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
```

CPU-GPU comm.

kernel setup

GPU kernel call

GPU-CPU comm.

# Open issues and research directions

# Open Issues

- **Non-linear, conflicting correlation of factors**
  - Performance relates to multiple factors: app/algorithm, dataset, hardware, resources
  - Poor combination of factors leads to CPU-GPU load imbalance and waste of resources
  - Huge design space, requiring considerable effort from developers to identify efficient execution settings
- **Different software and hardware offerings requiring specialized treatment**
  - A plethora of programming models and processors are available nowadays
  - Initial studies for HW/SF compatibility spots (e.g. [Herten@SC-W'23])
- **Expensive and energy-intensive infrastructure**
  - Optimize energy efficiency and compute performance
- **High complexity to predict performance metrics**
  - Accurate predictions is a challenge due to dynamic exec environment, skewed data
- **Shortage on GPU developers**
  - Hard to code, harder to optimize, harder to debug, less predictable
  - Need methods and abstractions

# Research Directions

- **Automated, multi-objective, multi-dimensional placement to optimize the trade-off '*processor choice vs. data transfer*'**
  - Avoid host-device round trips for memory allocation
  - Current approaches focus on optimizing a single or dependent performance metrics
  - How to balance performance, energy consumption, and monetary costs is an open question
  - Most works consider only a few of the dimensions/features listed in this tutorial
- **Automated fine-tuning of execution parameters**
  - Empirically determining and tuning parameters such as partition size, degree of parallelism, cache size, overlap of data transfer and compute, etc. is suboptimal
  - Kernel-level optimizations (e.g. MatMul-free LM [Zhu et al.@arXiv'24])
- **Adaptive cache-aware workload placement**
  - Cache is fast, but very limited in size (especially on small GPU off-chip memory)
  - Knowing when CPU-GPU cache is worth on generic workloads is an open question
- **Suitable heterogeneous CPU-GPU benchmarks and simulators to compare different strategies**
  - Build upon and extend initial efforts for benchmarks (e.g. for Python: NPBench, MLPerf, RAPIDS RAFT ANN; for C++ (OpenCL, CUDA): SHOC, Parboil, Rodinia, Valar) and simulators (e.g. Multi2Sim, gem5-gpu)

# Questions?

# Thank you!

**Marcos N. L. Carvalho**
marcos.nogueira@upc.edu

**Alkis Simitsis**
alkis@athenarc.gr

**Anna Queralt**
anna.queralt@upc.edu

**Oscar Romero**
oscar.romero@upc.edu

# References


[1] Luk, Chi-Keung, Sunpyo Hong, and Hyesoon Kim. "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping." Proceedings of the 42nd Annual IEEE/ACM international symposium on microarchitecture. 2009.
[2] Augonnet, Cédric, et al. "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures." Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings 15. Springer Berlin Heidelberg, 2009.
[3] He, Bingsheng, et al. "Relational query coprocessing on graphics processors." ACM Transactions on Database Systems (TODS) 34.4 (2009): 1-39.
[4] Kerr, Andrew, Gregory Diamos, and Sudhakar Yalamanchili. "Modeling GPU-CPU workloads and systems." Proceedings of the 3rd workshop on general-purpose computation on graphics processing units. 2010.
[5] Ravi, Vignesh T., et al. "Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations." Proceedings of the 24th ACM international conference on supercomputing. 2010.
[6] Yang, Canqun, et al. "Adaptive optimization for petascale heterogeneous CPU/GPU computing." 2010 IEEE International Conference on Cluster Computing. IEEE, 2010.
[7] Prasad, Ashwin, Jayvant Anantpur, and R. Govindarajan. "Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors." Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. 2011.
[8] Verner, Uri, Assaf Schuster, and Mark Silberstein. "Processing data streams with hard real-time constraints on heterogeneous systems." Proceedings of the international conference on Supercomputing. 2011.
[9] Grewe, Dominik, and Michael FP O'Boyle. "A static task partitioning approach for heterogeneous systems using OpenCL." Compiler Construction: 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 20. Springer Berlin Heidelberg, 2011.
[10] Ravi, Vignesh T., et al. "Scheduling concurrent applications on a cluster of cpu-gpu nodes." 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012). IEEE, 2012.
[11] Pirk, Holger. "Efficient cross-device query processing." The VLDB PhD Workshop. VLDB Endowment. 2012.
[12] Kim, Jungwon, et al. "SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters." Proceedings of the 26th ACM international conference on Supercomputing. 2012.
[13] Wang, Kaibo, et al. "Accelerating pathology image data cross-comparison on cpu-gpu hybrid systems." Proceedings of the VLDB endowment international conference on very large data bases. Vol. 5. No. 11. NIH Public Access, 2012.
[14] Ma, Kai, et al. "Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures." 2012 41st international conference on parallel processing. IEEE, 2012.
[15] Bosilca, George, et al. "DAGuE: A generic distributed DAG engine for high performance computing." Parallel Computing 38.1-2 (2012): 37-51.

# References


[16] Pienaar, Jacques A., Srimat Chakradhar, and Anand Raghunathan. "Automatic generation of software pipelines for heterogeneous parallel systems." SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE, 2012.

[17] Clarke, David, et al. "Hierarchical partitioning algorithm for scientific computing on highly heterogeneous cpu+ gpu clusters." Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings 18. Springer Berlin Heidelberg, 2012.

[18] Lu, Fengshun, et al. "Performance evaluation of hybrid programming patterns for large CPU/GPU heterogeneous clusters." Computer physics communications 183.6 (2012): 1172-1181.

[19] Bauer, Michael, et al. "Legion: Expressing locality and independence with logical regions." SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE, 2012.

[20] Gharaibeh, Abdullah, et al. "The energy case for graph processing on hybrid CPU and GPU systems." Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms. 2013.

[21] Kofler, Klaus, et al. "An automatic input-sensitive approach for heterogeneous task partitioning." Proceedings of the 27th international ACM conference on International conference on supercomputing. 2013.

[22] Lee, Janghaeng, et al. "Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems." Proceedings of the 22nd international conference on Parallel architectures and compilation techniques. IEEE, 2013.

[23] Pirk, Holger, Stefan Manegold, and Martin Kersten. "Waste not... Efficient co-processing of relational data." 2014 IEEE 30th International Conference on Data Engineering. IEEE, 2014.

[24] Cumming, Ben, et al. "Application centric energy-efficiency study of distributed multi-core and hybrid CPU-GPU systems." SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2014.

[25] Breß, Sebastian, et al. "Ocelot/hype: Optimized data processing on heterogeneous hardware." Proceedings of the VLDB Endowment 7.13 (2014): 1609-1612.

[26] Cheng, Xuntao, Bingsheng He, and Chiew Tong Lau. "Energy-efficient query processing on embedded CPU-GPU architectures." Proceedings of the 11th International Workshop on Data Management on New Hardware. 2015.

[27] Karnagel, Tomas, Dirk Habich, and Wolfgang Lehner. "Local *vs.* Global Optimization: Operator Placement Strategies in Heterogeneous Environments." EDBT/ICDT Workshops. 2015.

# References


[28] Rocklin, Matthew. "Dask: Parallel computation with blocked algorithms and task scheduling." SciPy. 2015.
[29] Abadi, Martín, et al. "{TensorFlow}: a system for {Large-Scale} machine learning." 12th USENIX symposium on operating systems design and implementation (OSDI 16). 2016.
[30] Heldens, Stijn, Ana Lucia Varbanescu, and Alexandru Iosup. "Dynamic load balancing for high-performance graph processing on hybrid cpu-gpu platforms." 2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3). IEEE, 2016.
[31] Sîrbu, Alina, and Ozalp Babaoglu. "Power consumption modeling and prediction in a hybrid CPU-GPU-MIC supercomputer." Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings 22. Springer International Publishing, 2016.
[32] Yuan, Yuan, et al. "Spark-GPU: An accelerated in-memory data processing engine on clusters." 2016 IEEE International Conference on Big Data (Big Data). IEEE, 2016.
[33] Koliousis, Alexandros, et al. "Saber: Window-based hybrid stream processing for heterogeneous architectures." Proceedings of the 2016 International Conference on Management of Data. 2016.
[34] Breß, Sebastian, Henning Funke, and Jens Teubner. "Robust query processing in co-processor-accelerated databases." Proceedings of the 2016 International Conference on Management of Data. 2016.
[35] Wen, Yuan, and Michael FP O'Boyle. "Merge or separate? Multi-job scheduling for OpenCL kernels on CPU/GPU platforms." Proceedings of the general purpose GPUs. 2017. 22-31.
[36] Appuswamy, Raja, et al. "The case for heterogeneous HTAP." 8th Biennial Conference on Innovative Data Systems Research. 2017.
[37] Mayer, Ruben, Christian Mayer, and Larissa Laich. "The tensorflow partitioning and scheduling problem: it's the critical path!." Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning. 2017.
[38] Karnagel, Tomas, Dirk Habich, and Wolfgang Lehner. "Adaptive work placement for query processing on heterogeneous computing resources." Proceedings of the VLDB Endowment 10.7 (2017): 733-744.
[39] Amela, Ramon, et al. "Executing linear algebra kernels in heterogeneous distributed infrastructures with PyCOMPSs." Oil & Gas Science and Technology–Revue d'IFP Energies nouvelles 73 (2018): 47.
[40] Mirhoseini, Azalia, et al. "A hierarchical model for device placement." International Conference on Learning Representations. 2018.
[41] Addanki, Ravichandra, et al. "Placeto: Efficient progressive device placement optimization." NIPS Machine Learning for Systems Workshop. Vol. 25. 2018.
[42] Wu, Qing, et al. "Adaptive DAG tasks scheduling with deep reinforcement learning." Algorithms and Architectures for Parallel Processing: 18th International Conference, ICA3PP 2018, Guangzhou, China, November 15-17, 2018, Proceedings, Part II 18. Springer International Publishing, 2018.

# References


[43] Moritz, Philipp, et al. "Ray: A distributed framework for emerging {AI} applications." 13th USENIX symposium on operating systems design and implementation (OSDI 18). 2018.
[44] Chen, Cen, et al. "GFlink: An in-memory computing architecture on heterogeneous CPU-GPU clusters for big data." IEEE Transactions on Parallel and Distributed Systems 29.6 (2018): 1275-1288.
[45] Gowanlock, Michael, et al. "Accelerating the unacceleratable: Hybrid CPU/GPU algorithms for memory-bound database primitives." Proceedings of the 15th International Workshop on Data Management on New Hardware. 2019.
[46] Khalid, Yasir Noman, et al. "Troodon: A machine-learning based load-balancing application scheduler for CPU–GPU system." Journal of Parallel and Distributed Computing 132 (2019): 79-94.
[47] Sant'Ana, Luis, Daniel Cordeiro, and Raphael Y. de Camargo. "PLB-HAC: dynamic load-balancing for heterogeneous accelerator clusters." Euro-Par 2019: Parallel Processing: 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26–30, 2019, Proceedings 25. Springer International Publishing, 2019.
[48] Paszke, Adam, et al. "Pytorch: An imperative style, high-performance deep learning library." Advances in neural information processing systems 32 (2019).
[49] Chrysogelos, Periklis, et al. "HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines." Proceedings of the VLDB Endowment 12.5 (2019): 544-556.
[50] Banerjee, Subho, et al. "Inductive-bias-driven reinforcement learning for efficient schedules in heterogeneous clusters." International Conference on Machine Learning. PMLR, 2020.
[51] Shanbhag, Anil, Samuel Madden, and Xiangyao Yu. "A study of the fundamental performance characteristics of GPUs and CPUs for database analytics." Proceedings of the 2020 ACM SIGMOD international conference on Management of data. 2020.
[52] Kamatar, Alok V., Ryan D. Friese, and Roberto Gioiosa. "Locality-aware scheduling for scalable heterogeneous environments." 2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS). IEEE, 2020.
[53] Lutz, Clemens, et al. "Pump up the volume: Processing large data on gpus with fast interconnects." Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2020.
[54] Zhang, Haitao, Xin Geng, and Huadong Ma. "Learning-driven interference-aware workload parallelization for streaming applications in heterogeneous cluster." IEEE Transactions on Parallel and Distributed Systems 32.1 (2020): 1-15.

# References


[55] Ren, Jie, et al. "{Zero-offload}: Democratizing {billion-scale} model training." 2021 USENIX Annual Technical Conference (USENIX ATC 21). 2021.
[56] Pérez, Borja, et al. "Sigmoid: An auto-tuned load balancing algorithm for heterogeneous systems." Journal of Parallel and Distributed Computing 157 (2021): 30-42.
[57] Li, Zexin, et al. "Efficient algorithms for task mapping on heterogeneous CPU/GPU platforms for fast completion time." Journal of Systems Architecture 114 (2021): 101936.
[58] Lee, Suyeon, and Sungyong Park. "Performance analysis of big data ETL process over CPU-GPU heterogeneous architectures." 2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW). IEEE, 2021.
[59] Lee, Rubao, et al. "The art of balance: a RateupDB™ experience of building a CPU/GPU hybrid database product." Proceedings of the VLDB Endowment 14.12 (2021): 2999-3013.
[60] Grinsztajn, Nathan, et al. "Readys: A reinforcement learning based strategy for heterogeneous dynamic scheduling." 2021 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 2021.
[61] Shanbhag, Anil, et al. "Tile-based lightweight integer compression in GPU." Proceedings of the 2022 International Conference on Management of Data. 2022.
[62] Xing, Haoyuan, Gagan Agrawal, and Rajiv Ramnath. "Gpu adaptive in-situ parallel analytics (gap)." Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. 2022.
[63] Liu, Haotian, et al. "Ghive: accelerating analytical query processing in apache hive via cpu-gpu heterogeneous computing." Proceedings of the 13th Symposium on Cloud Computing. 2022.
[64] Xekalaki, Maria, et al. "Enabling transparent acceleration of big data frameworks using heterogeneous hardware." Proceedings of the VLDB Endowment 15.13 (2022): 3869-3882.
[65] Yogatama, Bobbi W., Weiwei Gong, and Xiangyao Yu. "Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS." Proceedings of the VLDB Endowment 15.11 (2022): 2491-2503.
[66] Lee, Hochan, et al. "Parla: A python orchestration system for heterogeneous architectures." SC22: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2022.
[67] Boeschen, Nils, and Carsten Binnig. "Gacco-a gpu-accelerated oltp dbms." Proceedings of the 2022 International Conference on Management of Data. 2022.
[68] Kamatar, Alok, Ryan Friese, and Roberto Gioiosa. "A Task Based Approach for Co-Scheduling Ensemble Workloads on Heterogeneous Nodes." 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE, 2023.
[69] Park, Taehyeong, et al. "Orchestrating Large-Scale SpGEMMs using Dynamic Block Distribution and Data Transfer Minimization on Heterogeneous Systems." 2023 IEEE 39th International Conference on Data Engineering (ICDE). IEEE, 2023.
[70] Li, Zhenxing, et al. "CoTrain: Efficient Scheduling for Large-Model Training upon GPU and CPU in Parallel." Proceedings of the 52nd International Conference on Parallel Processing. 2023.

# References

[71] Nicholson, Hamish, et al. "Hetcache: Synergising nvme storage and GPU acceleration for memory-efficient analytics." 13th Annual Conference on Innovative Data Systems Research (CIDR 2023). 2023.
[72] Cui, Pengjie, et al. "CGgraph: An Ultra-fast Graph Processing System on Modern Commodity CPU-GPU Co-processor." Proceedings of the VLDB Endowment 17.6 (2024): 1405-1417.
[73] Kroviakov, Artem, et al. "Heterogeneous Intra-Pipeline Device-Parallel Aggregations." Proceedings of the 20th International Workshop on Data Management on New Hardware. 2024.
[74] Ye, Chang, et al. "gsword: Gpu-accelerated sampling for subgraph counting." Proceedings of the ACM on Management of Data 2.1 (2024): 1-26.
[75] Kim, Taeyoon, et al. "FusionFlow: Accelerating Data Preprocessing for Machine Learning with CPU-GPU Cooperation." Proceedings of the VLDB Endowment 17.4 (2024): 863-876.
[76] Ai, Xin, et al. "NeutronOrch: Rethinking Sample-Based GNN Training under CPU-GPU Heterogeneous Environments." Proceedings of the VLDB Endowment 17.8 (2024): 1995-2008.
[77] Nogueira Lobo de Carvalho, Marcos, et al. "Performance analysis of distributed GPU-accelerated task-based workflows." Proceedings 27th International Conference on Extending Database Technology (EDBT 2024): Paestum, Italy, March 25-March 28. OpenProceedings, 2024.
[78] Zhu, Rui-Jie, et al. "Scalable MatMul-free Language Modeling." arXiv preprint arXiv:2406.02528 (2024).
[79] Power, Jason, et al. "gem5-gpu: A heterogeneous cpu-gpu simulator." IEEE Computer Architecture Letters 14.1 (2014): 34-36.
[80] Herten, Andreas. "Many Cores, Many Models: GPU Programming Model vs. Vendor Compatibility Overview." Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. 2023.